

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

LOGIKA KOMPUTER



YAYASAN PRIMA AGUS TEKNIK



Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

LOGIKA KOMPUTER

The background features several snippets of Python code, primarily for image processing using OpenCV and NumPy. The code includes functions for loading images, calculating statistics, enhancing images (brightness, contrast, sharpness), and saving the results. Key elements include:

- Image Loading:** `img = cv2.imread(image_path)`
- Statistics Calculation:** `def calculate_statistics(data):` which calculates mean, median, and standard deviation.
- Image Enhancement:**
 - `enhance_image(image_path, output_path, scale_factor, brightness, contrast, sharpness)`
 - Sub-functions: `enhancer = ImageEnhance.Brightness(img)`, `enhancer = ImageEnhance.Contrast(img)`, `enhancer = ImageEnhance.Sharpness(img)`
- Kernel and Sharpening:** `kernel = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])` and `sharp_img = cv2.filter2D(img, -1, kernel)`
- CLAHE:** `clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8,8))`
- Color Conversion:** `lab = cv2.cvtColor(sharp_img, cv2.COLOR_BGR2LAB)` and `final = cv2.cvtColor(limg, cv2.COLOR_LAB2BGR)`
- Saving:** `cv2.imwrite(output_path, final)` and `enhance_image('path/to/your/image.jpg', 'path/to/save/edited_image.jpg', 1.0, 1.2, 1.3, 2.0)`



YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit ypat@stekom.ac.id

ISBN 978-634-7695-16-1 (PDF)



○

786347

695161

LOGIKA KOMPUTER

Penulis :

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

ISBN : 978-634-7695-16-1 (PDF)

Editor :

Dr. Joseph Teguh Santoso, S.Kom., M.Kom.

Penyunting :

Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

Desain Sampul dan Tata Letak :

Irdha Yuniarto, S.Ds., M.Kom

Penebit :

Yayasan Prima Agus Teknik Bekerja sama dengan
Universitas Sains & Teknologi Komputer (Universitas STEKOM)

Anggota IKAPI No: 279 / ALB / JTE / 2023

Redaksi :

Jl. Majapahit no 605 Semarang

Telp. 08122925000

Fax. 024-6710144

Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :**Universitas STEKOM**

Jl. Majapahit no 605 Semarang

Telp. 08122925000

Fax. 024-6710144

Email : info@stekom.ac.id

Hak cipta dilindungi undang-undang

Dilarang memperbanyak karya tulis ini dalam bentuk dan dengan cara
apapun tanpa ijin dari penulis

KATA PENGANTAR

Di era Revolusi Industri 4.0 dan transformasi digital yang pesat di Indonesia, logika komputer bukan lagi sekadar teori abstrak, melainkan senjata utama untuk membangun sistem cerdas, aman, dan efisien—mulai dari IoT hingga verifikasi AI. Logika komputer merupakan pondasi fundamental dalam ilmu komputer modern, yang tidak hanya mendasari desain perangkat keras dan perangkat lunak, tetapi juga menjadi kunci dalam verifikasi sistem, kecerdasan buatan, dan pemecahan masalah kompleks.

Buku *Logika Komputer* ini hadir untuk membekali pembaca dengan pemahaman mendalam tentang prinsip-prinsip logika, mulai dari dasar proposisional hingga aplikasi lanjutan seperti verifikasi program dan teori komputabilitas. Dengan pendekatan yang sistematis, buku ini menggabungkan teori, algoritma, dan latihan soal praktis, sehingga cocok untuk mahasiswa tingkat sarjana dan pascasarjana, serta para profesional yang ingin memperkuat fondasi logika dalam pengembangan sistem digital.

Buku ini terstruktur dalam 12 bab yang saling terkait, dimulai dari pengantar dasar hingga topik mutakhir. Bab 1: Pengantar Logika memperkenalkan keberadaan logika dalam kehidupan sehari-hari, sejarah singkatnya, kesalahan umum, sekilas logika matematika, serta teori pembuktian dasar, dilengkapi latihan untuk membangun intuisi awal.

Bab 2: Logika Proposisional membahas sintaksis dan semantik proposisi, bentuk normal, masalah optimasi, serta aplikasi praktisnya, dengan latihan untuk menguasai representasi boolean. Bab 3: Penalaran dalam Logika Proposisional mengeksplorasi prosedur pembuktian, sistem deduktif, resolusi, dan propagasi batasan Boolean (BCP), menyediakan alat untuk penalaran otomatis. Bab 4: Keterpuasan Proposisional fokus pada algoritma DPLL, pembelajaran klausa berbasis konflik (CDCL), penggunaan pemecah SAT, serta keterpuasan maksimum, yang krusial untuk solver modern.

Pada Bab 5: Logika Orde Pertama menguraikan sintaksis, semantik, metode pembuktian, dan bentuk normal konjungtif (CNF), memperluas logika ke domain yang lebih ekspresif. Bab 6: Unifikasi dan Resolusi membahas unifikasi, resolusi lanjutan, urutan penyederhanaan, serta alat seperti Prover9, untuk pembuktian teori otomatis.

Bab 7: Logika Orde Pertama dengan Kesetaraan menangani kesamaan istilah, sistem penulisan ulang, pembuktian induktif, resolusi dengan kesetaraan, dan pencarian model terbatas. Bab 8: Prolog Pemrograman dalam Logika memperkenalkan prinsip Prolog, tipe data, rekursi, serta ekstensi di luar klausa logika, dengan aplikasi pemrograman deklaratif. Bab 9: Logika Hoare membahas verifikasi formal, triple Hoare, otomatisasi pernyataan, dan invarian loop, untuk memastikan kebenaran program.

Bab 10: Logika Temporal mengeksplorasi pendekatan modal, logika temporal linier (LTL), tabel semantik, operator biner, serta verifikasi program konkuren. Bab 11: Masalah yang Dapat Diputuskan dan Tidak Dapat Diputuskan mengulas sejarah ketidakpastian, mesin Turing, keterputusan, komputabilitas bijeksi, dan kelengkapan Turing. Bab 12: Prosedur Pengambilan Keputusan membahas perluasan DPLL dengan teori, kesamaan fungsi, aritmatika linier, dan DPLL(T) praktis untuk masalah nyata.

Melalui buku ini, pembaca diharapkan tidak hanya memahami teori, tetapi juga mampu menerapkannya dalam proyek nyata seperti verifikasi perangkat lunak atau pengembangan AI. Terima kasih kepada semua kontributor dan pembaca setia. Semoga buku ini menjadi panduan berharga dalam perjalanan eksplorasi logika komputer.

Semangat dan Selamat Membaca...!!!

Semarang, Mei 2026
Penulis

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

DAFTAR ISI

KATA PENGANTAR.....	ii
DAFTAR ISI.....	iv
BAB 1 PENGANTAR LOGIKA.....	1
1.1 Logika Ada Di Mana-Mana	2
1.2 Sejarah Singkat Logika	5
1.3 Kesalahan Logika	8
1.4 Sekilas Tentang Logika Matematika.....	11
1.5 Teori Pembuktian.....	27
1.6 Latihan Soal	31
BAB 2 LOGIKA PROPOSISIONAL	35
2.1 Sintaksis	35
2.2 Semantik.....	38
2.3 Bentuk Normal	49
2.4 Masalah Optimasi.....	61
2.5 Menggunakan Logika Proposisional	68
2.6 Latihan Soal	76
BAB 3 PENALARAN DALAM LOGIKA PROPOSISIONAL	83
3.1 Prosedur Pembuktian	83
3.2 Sistem Deduktif	89
3.3 Resolusi.....	97
3.4 Propagasi Batasan Boolean (BCP).....	112
3.5 Latihan Soal	121
BAB 4 KETERPUASAN PROPOSISIONAL.....	124
4.1 Algoritma DPLL	124
4.2 Pembelajaran Klausa Berbasis Konflik (CDCL)	133
4.3 Penggunaan Pemecah SAT.....	143
4.4 Kepuasan Maksimum	149
4.5 Latihan Soal	162
BAB 5 LOGIKA ORDE PERTAMA	166
5.1 Sintaksis Bahasa Orde Pertama	166
5.2 Semantik.....	174
5.3 Metode Pembuktian.....	190
5.4 Bentuk Normal Konjungtif (CNF)	194
5.5 Latihan Soal	204
BAB 6 UNIFIKASI DAN RESOLUSI	208
6.1 Unifikasi	208
6.2 Resolusi.....	221

6.3	Urutan Penyederhanaan Dan Resolusi Terurut	225
6.4	Prover9: Pembukti Teorema Resolusi.....	235
6.5	Latihan Soal	242
BAB 7	LOGIKA ORDE PERTAMA DENGAN KESETARAAN.....	245
7.1	Kesamaan Istilah.....	245
7.2	Sistem Penulisan Ulang	252
7.3	Pembuktian Teorema Induktif	268
7.4	Resolusi Dengan Kesetaraan.....	279
7.5	Pencarian Model Terbatas Dalam Logika Orde Pertama	285
7.6	Latihan Soal	291
BAB 8	PROLOG PEMROGRAMAN DALAM LOGIKA	294
8.1	Prinsip Kerja Prolog	294
8.2	Tipe Data Prolog	303
8.3	Rekursi Dalam Prolog.....	309
8.4	Di Luar Klausa Dan Logika.....	315
8.5	Latihan Soal	321
BAB 9	LOGIKA HOARE.....	323
9.1	Verifikasi Formal Sistem Komputer	323
9.2	Triple Hoare	326
9.3	Otomatisasi Pembuatan Pernyataan	337
9.4	Memperoleh Invarian Loop Yang Baik.....	347
9.5	Latihan Soal	355
BAB 10	LOGIKA TEMPORAL	358
10.1	Pendekatan Dari Logika Modal.....	359
10.2	Logika Temporal Linier.....	366
10.3	Tabel Semantik Untuk LTL.....	375
10.4	Operator Temporal Biner.....	387
10.5	Verifikasi Program Konkuren	391
10.6	Latihan Soal	399
BAB 11	MASALAH YANG DAPAT DIPUTUSKAN DAN TIDAK DAPAT DIPUTUSKAN	403
11.1	Sejarah Teori Ketidakpastian	403
11.2	Mesin Turing.....	406
11.3	Keterputusan Masalah	414
11.4	Komputabilitas Bijeksi Penghitungan	427
11.5	Kelengkapan Turing	436
11.6	Latihan Soal	441
BAB 12	PROSEDUR PENGAMBILAN KEPUTUSAN	443
12.1	Memperluas DPLL Dengan Teori T.....	445
12.2	Kesamaan Dengan Fungsi Yang Tidak Diinterpretasikan	452
12.3	Aritmatika Linier	464
12.4	Membuat DPLL(T) Praktis	476

12.5 Latihan Soal	484
DAFTAR PUSTAKA	486

BAB 1

PENGANTAR LOGIKA

Apakah Anda suka memecahkan teka-teki Sudoku? Jika ya, berapa lama rata-rata waktu yang dibutuhkan untuk memecahkan teka-teki Sudoku yang muncul di surat kabar, menit atau jam? Latihan mudah di Bab 1 buku ini meminta Anda untuk menulis program kecil dalam bahasa favorit Anda yang akan memecahkan teka-teki Sudoku, apa pun tingkat kesulitannya, dalam waktu kurang dari satu detik di laptop Anda.

Pernahkah Anda memainkan Menara Hanoi? Teka-teki ini terdiri dari tiga batang dan sejumlah cakram dengan ukuran berbeda, yang dapat digeser ke batang mana pun. Teka-teki dimulai dengan cakram yang ditumpuk dalam urutan ukuran yang meningkat pada satu batang, membentuk bentuk kerucut dengan yang terkecil di atas. Tujuan teka-teki ini adalah untuk memindahkan seluruh tumpukan ke batang lain, mengikuti tiga aturan:

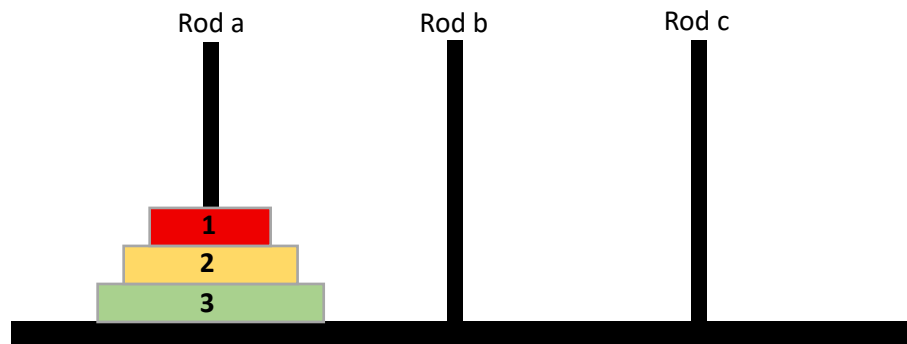
- Hanya satu cakram yang dapat dipindahkan pada satu waktu.
- Setiap gerakan terdiri dari mengambil cakram teratas dari salah satu tumpukan dan meletakkannya di atas tumpukan lain atau pada batang kosong.
- Tidak ada cakram yang lebih besar yang boleh diletakkan di atas cakram yang lebih kecil.

Pada gambar ilustrasi, ketiga batang diberi nama *a*, *b*, dan *c* (Gambar 1.1). Awalnya terdapat tiga cakram pada batang *a*, dan setiap cakram diberi nama sesuai ukurannya. Solusi teka-teki memindahkan cakram dari batang *a* ke batang *b* diberikan sebagai berikut:

```
Pindahkan cakram 1 dari a ke b
Pindahkan cakram 2 dari a ke c
Pindahkan cakram 1 dari b ke c
Pindahkan cakram 3 dari a ke b
Pindahkan cakram 1 dari c ke a
Pindahkan cakram 2 dari c ke b
Pindahkan cakram 1 dari a ke b
```

Solusi ini dihasilkan oleh program Prolog sepanjang 3 baris dan 168 karakter (termasuk koma dan titik). Program yang sama dapat menghasilkan solusi untuk berbagai jumlah cakram. Prolog adalah bahasa pemrograman berbasis logika dan diperkenalkan pada Bab 8 buku ini.

Dua contoh di atas mengilustrasikan bahwa logika bukan hanya landasan teoritis ilmu komputer tetapi juga alat pemecahan masalah. Penekanan buku ini adalah pada bagaimana membuat alat ini efisien dan praktis. Setelah pengantar yang ketat tentang konsep-konsep dasar, kita akan mempelajari berbagai algoritma berbasis logika dan memperkenalkan alat perangkat lunak yang merupakan implementasi dari algoritma-algoritma ini.



Gambar 1.1 Ilustrasi Menara Hanoi

Untuk memecahkan masalah menggunakan alat berbasis logika, Anda perlu memikirkan masalah tersebut secara logis dan menentukan masalah tersebut secara logis. Ini adalah keterampilan yang perlu dipelajari seperti keterampilan lainnya, dan memang membutuhkan pelatihan dan praktik untuk menguasai keterampilan ini. Keterampilan yang sama dapat diterapkan pada situasi abstrak seperti yang ditemui dalam pembuktian formal.

Logika adalah bawaan bagi kita semua—memang, Anda mungkin menggunakan hukum logika secara tidak sadar dalam ucapan sehari-hari dan dalam penalaran internal Anda sendiri. Karena logika bersifat bawaan, prinsip-prinsip logika yang kita pelajari harus masuk akal—jika Anda mendapati diri Anda harus menghafal salah satu prinsip dalam buku, tanpa merasakan "klik" mental atau memahami mengapa hukum itu harus bekerja, maka Anda mungkin tidak akan dapat menggunakan prinsip itu dengan benar dan efektif dalam praktik. Pada tahun 1953 Albert Einstein menulis hal berikut dalam surat kepada J. E. Switzer:

Perkembangan Sains Barat telah didasarkan pada dua pencapaian besar, penemuan sistem logika formal (dalam geometri Euklides) oleh para filsuf Yunani, dan penemuan kemungkinan untuk menemukan hubungan sebab akibat melalui eksperimen sistematis (pada masa Renaisans).

Buku ini menunjukkan penerapan dua pencapaian besar sains modern ini dalam satu konteks: mengembangkan sistem logika formal melalui eksperimen sistematis. Artinya, fokus kita adalah mengubah algoritma sistem logika menjadi alat logika, dan baik algoritma maupun alat tersebut membutuhkan eksperimen sistematis.

1.1 LOGIKA ADA DI MANA-MANA

Logika disebut sebagai kalkulus ilmu komputer, karena logika merupakan hal mendasar dalam ilmu komputer, mirip dengan kalkulus dalam ilmu fisika dan teknik. Logika digunakan di hampir setiap bidang ilmu komputer: arsitektur komputer (seperti gerbang digital, verifikasi perangkat keras), rekayasa perangkat lunak (spesifikasi dan verifikasi), bahasa pemrograman (semantik, teori tipe, tipe data abstrak, pemrograman berorientasi objek), basis data (aljabar relasional), kecerdasan buatan (pembuktian teorema otomatis, representasi pengetahuan), algoritma dan teori komputasi (kompleksitas, komputabilitas), dll.

Logika juga memainkan peran penting di bidang lain, seperti matematika, psikologi, dan filsafat. Dalam matematika, logika mencakup studi matematika tentang logika dan aplikasi logika formal ke bidang matematika lainnya. Tema-tema pemersatu dalam logika matematika meliputi studi tentang kekuatan ekspresif sistem formal dan kekuatan deduktif sistem pembuktian formal. Logika matematika selanjutnya dibagi menjadi bidang (a) teori himpunan, (b) teori model, (c) teori pembuktian, dan (d) teori komputabilitas. Bidang-bidang ini memiliki hasil dasar yang sama dalam logika.

Logika simbolik pada akhir abad kesembilan belas dan logika matematika pada abad kedua puluh juga disebut logika formal. Topik-topik yang secara tradisional ditangani oleh logika tetapi bukan bagian dari logika formal sering disebut logika filosofis, yang membahas argumen dalam bahasa alami.

Contoh. Diberikan premis bahwa (a) “semua manusia fana” dan (b) “Adam adalah manusia,” kita dapat menarik kesimpulan bahwa (c) “Adam fana,” dengan aturan inferensi *modus ponens*

Logika filosofis adalah penyelidikan, analisis kritis, dan refleksi intelektual tentang isu-isu yang muncul dalam logika. Ini adalah cabang studi tentang pertanyaan-pertanyaan mengenai referensi, prediksi, identitas, kebenaran, kuantifikasi, eksistensi, implikasi, modalitas, dan keniscayaan. Banyak dari topik-topik ini dimiliki bersama oleh logika matematika dan dibahas dalam buku ini. Oleh karena itu, mempelajari buku ini akan membantu kita berpikir, menulis, atau berbicara secara logis.

Pernyataan atau Proposisi

Logika berasal dari bahasa alami karena sebagian besar kalimat dalam bahasa alami adalah pernyataan atau proposisi. Proposisi adalah kalimat yang menyatakan penilaian atau pendapat dan memiliki nilai kebenaran. Kita akan menggunakan pernyataan dan penegasan sebagai sinonim dari proposisi. Berikut beberapa contohnya:

- Warna apel itu hijau.
- Hari ini adalah Senin atau Selasa.
- Dia adalah mahasiswa tahun kedua berusia 20 tahun.

Setiap pernyataan dapat benar atau salah. Misalnya, pernyataan pertama di atas dapat benar untuk satu apel tetapi salah untuk apel lainnya. Benar atau salah disebut nilai kebenaran suatu pernyataan. Beberapa kalimat dalam bahasa alami bukanlah pernyataan, seperti perintah atau kalimat seru. Misalnya, “Lari cepat!” teriak pelatih. Bagian pertama kalimat, “Lari cepat!”, bukanlah pernyataan; bagian kedua, “Pelatih berteriak,” adalah pernyataan. Pada kenyataannya, sebuah kalimat adalah pernyataan jika dan hanya jika kalimat tersebut memiliki nilai kebenaran.

Dalam bahasa alami, kita dapat menggabungkan atau menghubungkan pernyataan dengan kata-kata seperti “tidak” (negasi), “dan” (konjungsi), “atau” (disjungsi), “jika-maka” (implikasi), dan lain-lain. Artinya, sebuah pernyataan dapat diperoleh dari pernyataan lain melalui kata-kata ini. Dalam logika, kata-kata ini disebut operator logika, atau setara dengan penghubung logika. Sebuah pernyataan dikatakan tersusun jika dapat diungkapkan sebagai

susunan dari beberapa pernyataan yang lebih sederhana; jika tidak, pernyataan tersebut sederhana. Dalam tiga contoh pernyataan di atas, pernyataan pertama sederhana; dua pernyataan lainnya tersusun. Artinya, "hari ini adalah Senin atau Selasa" adalah susunan dari "hari ini adalah Senin" dan "hari ini adalah Selasa," menggunakan operator logika "atau." Kalimat kedua, "Dia adalah mahasiswa tahun kedua berusia 20 tahun," adalah susunan dari "dia berusia 20 tahun" dan "dia adalah mahasiswa tahun kedua," menggunakan operator logika implisit "dan." Kita sering menganggap "warna apel itu bukan hijau" sebagai pernyataan tersusun: Ini adalah negasi dari pernyataan sederhana.

Operator logika sangat penting untuk menyatakan hubungan antar pernyataan. Misalnya, pernyataan berikut merupakan komposisi dari beberapa pernyataan sederhana.

(*) Jika pajak tidak dinaikkan atau pengeluaran meningkat, maka batas utang dinaikkan.

Untuk melihat ini dengan jelas, kita perlu memperkenalkan simbol, seperti t , e , dan d , untuk menunjukkan pernyataan sederhana:

1. t : "pajak dinaikkan"
2. e : "pengeluaran meningkat"
3. d : "batas utang dinaikkan"

Untuk menyatakan bahwa pernyataan (*) merupakan komposisi, mari kita gunakan simbol-simbol ini untuk operator logika: $\neg$ untuk negasi, $\vee$ untuk "salah satu atau yang lain," dan $\rightarrow$ untuk relasi "jika-maka" (yaitu, implikasi). Maka (*) menjadi rumus berikut:

$$((\neg t) \vee e) \rightarrow d$$

Di sini, masing-masing simbol t , e , dan d disebut variabel proposisional, yang menunjukkan sebuah pernyataan, baik sederhana maupun majemuk. Secara alami, variabel proposisional ini, seperti pernyataan, hanya dapat mengambil nilai kebenaran: benar dan salah. Setiap variabel proposisional dapat diberi interpretasi atau makna. Misalnya, interpretasi t dalam contoh di atas adalah "pajak dinaikkan."

Variabel proposisional juga disebut variabel Boolean, diambil dari nama penemunya, matematikawan abad ke-19 George Boole. Logika Boolean mencakup logika apa pun di mana nilai kebenaran yang dipertimbangkan adalah benar dan salah. Buku ini secara eksklusif mempelajari logika Boolean. Studi tentang variabel proposisional dengan operator logika disebut logika proposisional, yang merupakan yang paling sederhana dalam keluarga logika Boolean. Ini dapat dibedakan dari logika probabilitas, yang bukan logika Boolean karena nilai probabilitas digunakan untuk mewakili berbagai derajat nilai kebenaran. Pernyataan yang mengandung kata sifat subyektif, seperti "apel itu enak," lebih baik diungkapkan dalam logika probabilitas, karena "enak" bersifat subyektif. Pertanyaan mengenai derajat kebenaran pernyataan subjektif diabaikan dalam logika Boolean. Misalnya, pernyataan, "apel itu enak," bisa benar menurut pandangan seseorang dan salah menurut pandangan orang lain. Terlepas dari penyederhanaan ini, atau justru karena penyederhanaan ini, logika Boolean berhasil

secara ilmiah. Seseorang bahkan tidak perlu mengetahui secara pasti nilai kebenaran benar dan salah itu.

1.2 SEJARAH SINGKAT LOGIKA

Teori-teori logika yang canggih dikembangkan di banyak budaya, termasuk Yunani, India, Cina, dan dunia Islam. Metode Yunani, khususnya logika Aristoteles (atau logika istilah), menemukan aplikasi dan penerimaan yang luas dalam sains dan matematika Barat selama ribuan tahun.

Aristoteles (384–322 SM) adalah ahli logika pertama yang mencoba analisis sistematis sintaksis logis, kata benda (atau istilah), dan kata kerja. Ia mendemonstrasikan prinsip-prinsip penalaran dengan menggunakan variabel untuk menunjukkan bentuk logis yang mendasari suatu argumen. Ia mencari hubungan ketergantungan yang menjadi ciri inferensi yang diperlukan, dan membedakan validitas hubungan ini dari kebenaran premis. Ia adalah orang pertama yang secara sistematis membahas prinsip-prinsip kontradiksi, yang menyatakan bahwa suatu proposisi tidak mungkin benar dan salah sekaligus, dan mengecualikan bagian tengah, yang menyatakan bahwa untuk setiap proposisi, proposisi ini atau negasinya adalah benar. Aristoteles memiliki pengaruh yang sangat besar dalam pemikiran Barat. Ia mengembangkan teori silogisme, di mana tiga prinsip penting diterapkan untuk pertama kalinya dalam sejarah: penggunaan simbol, perlakuan yang murni formal, dan penggunaan sistem aksiomatik. Aristoteles juga mengembangkan teori kekeliruan, sebagai teori logika non-formal.

Ahli logika Yunani kuno terkenal lainnya adalah Euklides (325–265 SM), yang meletakkan dasar-dasar geometri yang sebagian besar mendominasi bidang ini hingga awal abad kesembilan belas. Geometri Euklides adalah sistem aksiomatik, di mana semua teorema diturunkan dari sejumlah kecil aksioma sederhana. Penalaran Euklides dari asumsi ke kesimpulan tetap valid setelah penemuan geometri non-Euklides.

Para filsuf Kristen dan Islam seperti Boethius (meninggal 524), Ibn Sina (Avicenna, meninggal 1037), dan William of Ockham (meninggal 1347) mengembangkan lebih lanjut logika Aristoteles di Abad Pertengahan, mencapai puncaknya pada pertengahan abad keempat belas dengan Jean Buridan (1301–1358/62 M).

Pada abad kesembilan belas, upaya untuk memperlakukan operasi logika formal secara simbolis atau aljabar telah dilakukan oleh para matematikawan filosofis, termasuk Leibniz dan Lambert, tetapi karya-karya mereka tetap terisolasi dan kurang dikenal. Pada awal abad kesembilan belas, logika dipelajari dengan retorika, melalui silogisme, dan dengan filsafat. Logika matematika muncul pada pertengahan abad kesembilan belas sebagai bidang matematika yang independen dari studi logika tradisional. Perkembangan logika simbolik modern atau logika matematika selama periode ini oleh tokoh-tokoh seperti Boole, Frege, Russell, dan Peano merupakan yang paling signifikan dalam sejarah logika selama 2000 tahun dan dapat dikatakan sebagai salah satu peristiwa terpenting dan paling luar biasa dalam sejarah intelektual manusia.

Berpikir atau Menulis Secara Logis

Bahasa adalah alat untuk berpikir dan menulis. Bersikap logis berarti bahwa realitas dunia yang diungkapkan oleh bahasa adalah akurat. Oleh karena itu, bahasa dan logika tidak dapat dipisahkan, karena logika berkaitan dengan kebenaran. Untuk berpikir atau menulis secara logis, kita perlu memperhatikan poin-poin berikut:

- *Mencocokkan kata dengan fakta atau entitas*

Fakta adalah sesuatu yang dibuat atau dilakukan. Entitas adalah sesuatu dengan keberadaan yang berbeda dan independen. Kita akan menggunakan kata fakta untuk fakta dan entitas. Fakta memiliki status objektif. Sebuah kata dapat digunakan untuk menunjukkan fakta. Arti sebuah kata berubah jika kata tersebut digunakan untuk mewakili fakta yang berbeda, seperti halnya interpretasi variabel proposisional berubah dalam aplikasi yang berbeda. Bersikap logis mengharuskan kata tersebut secara jelas mewakili fakta dalam konteks tertentu. Banyak kata memiliki banyak arti. Misalnya, "apel" dapat berupa nama buah atau nama perusahaan. Seringkali, kita tidak dapat menemukan kata yang tepat untuk sebuah fakta. Kita mungkin membutuhkan kata baru atau menambahkan kata sifat pada sebuah kata agar kata tersebut sesuai dengan fakta. Mencampuradukkan makna suatu kata dalam konteks yang sama menyebabkan kebingungan. Dalam percakapan, kedua pihak harus memberikan makna yang sama pada suatu kata.

- *Membuat pernyataan yang benar*

Sebagian besar kalimat dalam bahasa alami adalah pernyataan. Banyak pernyataan dapat diungkapkan dalam logika proposisional. Beberapa di antaranya tidak dapat diungkapkan dengan logika proposisional, tetapi dapat diungkapkan dengan logika lain, seperti logika orde pertama. Pernyataan sederhana yang benar harus merupakan representasi dari fakta objektif. Ketika pernyataan sederhana tidak dapat secara meyakinkan benar, pernyataan tersebut memberi kita representasi yang menyimpang dari dunia objektif. Nilai kebenaran pernyataan kompleks dapat ditentukan dengan penalaran logis, yang dapat ditemukan kemudian dalam buku ini. Berbicara atau menulis kalimat yang tidak lengkap adalah kebiasaan buruk, karena kalimat-kalimat ini tidak dapat menjadi pernyataan. Kita harus berhati-hati ketika suatu pernyataan mengandung kata sifat subyektif karena kebenaran pernyataan ini bersifat subyektif. Kita harus berusaha untuk bersikap lugas dalam menulis atau berbicara untuk membantu pembaca atau audiens memahami dan menerima kebenaran pernyataan Anda.

Melanggar dua poin di atas menyebabkan kesalahan, banyak di antaranya disebut sebagai kekeliruan logika.

Contoh 1.1.1 Kata “logika” memiliki banyak arti. Dalam bagian ini, artinya adalah “studi tentang penalaran yang benar,” termasuk logika formal dan informal. Fokus buku ini adalah pada sebagian kecil logika formal yang digunakan dalam ilmu

komputer untuk membangun alat penalaran. Oleh karena itu, kata “logika” dalam judul buku ini berarti “ilmu yang berkaitan dengan prinsip dan kriteria validitas inferensi dan demonstrasi.” Di sisi lain, “logika” dalam “logika proposisional” berarti “prinsip-prinsip formal dari suatu cabang pengetahuan.” Cara penggunaan kata “logika” ini mungkin menimbulkan beberapa kebingungan, tetapi tidak akan pernah menjadi hambatan untuk mempelajari logika dalam ilmu komputer.

Jika kita mencocokkan satu kata dengan dua fakta, kebingungan dapat muncul seperti yang ditunjukkan oleh Paradoks Pengadilan, juga dikenal sebagai paradoks Protagoras, yang berasal dari Yunani kuno.

Paradoks Pengadilan Sofis terkenal Protagoras menerima seorang murid yang menjanjikan, Euathlus, dengan pemahaman bahwa murid tersebut akan membayar Protagoras untuk pengajarannya setelah ia memenangkan kasus pengadilan pertamanya. Setelah pengajaran, Euathlus memutuskan untuk tidak memasuki profesi hukum, dan karena itu Protagoras memutuskan untuk menuntut Euathlus atas jumlah yang menjadi haknya. Protagoras berpendapat bahwa jika ia memenangkan kasus tersebut, ia akan dibayar uangnya. Jika Euathlus memenangkan kasus tersebut, Protagoras tetap akan dibayar sesuai dengan kontrak awal, karena Euathlus akan memenangkan kasus pertamanya. Namun, Euathlus mengklaim bahwa jika ia menang, maka berdasarkan keputusan pengadilan ia tidak perlu membayar Protagoras. Sebaliknya, jika Protagoras menang, maka Euathlus tetap tidak akan memenangkan kasus dan oleh karena itu tidak berkewajiban untuk membayar. Pertanyaannya kemudian adalah, siapakah di antara kedua pria itu yang benar?

Cerita berlanjut dengan mengatakan bahwa para juri malu untuk mengambil keputusan dan menunda kasus tersebut tanpa batas waktu. Kebingungan paradoks ini terletak pada pencampuran "dua jenis pembayaran": satu berdasarkan kontrak dan yang lainnya berdasarkan perintah pengadilan. Jika kita membedakan kedua jenis pembayaran ini, maka paradoks tersebut tidak lagi ada.

Misalkan p menunjukkan "Euathlus memenangkan kasus pertamanya" dan $q(x)$ menunjukkan "Euathlus membayar Rp. x kepada Protagoras." Maka dua pernyataan berikut ini benar:

- **Berdasarkan kontrak:** Jika p maka $q(t)$ jika tidak $q(0)$, di mana Rp. t adalah biaya kuliah.
- **Berdasarkan perintah pengadilan:** Jika p maka $q(x)$ jika tidak $q(y)$, di mana x dapat negatif jika pengadilan meminta Protagoras untuk membayar Euathlus.

Dengan menggabungkan kedua pernyataan di atas, pernyataan "jika p maka $q(t + x)$ jika tidak $q(y)$ " adalah benar. Artinya, Euathlus perlu membayar Protagoras Rp. $(t + x)$ jika ia memenangkan kasus tersebut, dan Rp. y , jika sebaliknya. Jika $x = 0$ dan $y = t$, Protagoras akan dibayar dengan jumlah yang sama dalam kedua kasus tersebut. Euathlus tidak perlu membayar Protagoras jika $x = -t$ (ketika ia menang) atau $y = 0$ (ketika ia kalah), dua keputusan pengadilan yang tidak mungkin terjadi.

Paradoks bilangan menarik menyatakan bahwa, jika kita mengklasifikasikan setiap bilangan asli sebagai "menarik" atau "tidak menarik," maka setiap bilangan asli adalah menarik. Sebuah "bukti dengan kontradiksi" berjalan sebagai berikut: Jika ada himpunan S yang tidak kosong dari bilangan asli yang tidak menarik, maka bilangan terkecil dari S adalah menarik karena merupakan bilangan terkecil yang tidak menarik, sehingga menghasilkan kontradiksi. Paradoks ini disebabkan oleh kurangnya definisi formal tentang "bilangan menarik," sehingga suatu bilangan dapat sekaligus menarik dan tidak menarik.

Dalam logika, paradoks berfungsi sebagai contoh nyata dari inkonsistensi suatu teori dan seringkali menjadi motivasi untuk pengembangan teori baru. Dalam bahasa sehari-hari, menemukan paradoks dalam argumen lawan adalah cara yang baik untuk memenangkan debat. Terkadang, paradoks dapat menjadi ungkapan yang berguna. Frasa "perasaan yang tak terlukiskan" adalah contohnya.

Jika perasaan itu tak terlukiskan, maka frasa ini tidak menyampaikan apa pun. Tetapi jika kata "tak terlukiskan" mengkomunikasikan sesuatu tentang perasaan itu, maka kata itu memang menyampaikan sesuatu: Ini kontradiktif dalam logika tetapi berguna dalam bahasa sehari-hari. Sebagai contoh lain, "'Mustahil' tidak ada dalam kosakata kita!" Meskipun pernyataan yang salah dalam logika, itu adalah ungkapan yang ampuh dalam pidato motivasi. Pernyataan Plato "Aku tahu bahwa aku tidak tahu apa-apa" juga merupakan pernyataan salah yang bermakna.

1.3 KESALAHAN LOGIKA

Kesalahan logika adalah kesalahan logika yang dilakukan seseorang saat menulis atau berbicara. Pada bagian ini, kita akan memberikan pengantar singkat tentang kesalahan logika, yang merupakan subjek logika filosofis. Mengetahui kesalahan-kesalahan ini akan membantu kita untuk menghindarinya. Menulis secara logis berkaitan dengan, tetapi tidak sama dengan, menulis secara jelas, efisien, meyakinkan, atau informatif; idealnya, seseorang ingin melakukan semua ini sekaligus, tetapi seringkali seseorang tidak memiliki keterampilan untuk mencapai semuanya. Meskipun dengan latihan, Anda akan dapat mencapai lebih banyak tujuan penulisan Anda secara bersamaan. Keuntungan besar dari menulis secara logis adalah Anda dapat benar-benar yakin bahwa kesimpulan Anda akan benar, selama semua hipotesis Anda benar, dan langkah-langkah Anda logis.

Dalam logika filosofis, argumen memiliki arti yang mirip dengan bukti dalam logika formal, karena argumen terdiri dari premis dan kesimpulan. Kesalahan adalah penggunaan penalaran yang tidak valid atau salah dalam konstruksi argumen. Argumen yang keliru dapat menipu dengan tampak lebih baik daripada yang sebenarnya. Beberapa kekeliruan dilakukan secara sengaja untuk memanipulasi atau membujuk dengan tipu daya, sementara yang lain dilakukan secara tidak sengaja karena kecerobohan atau ketidaktahuan.

Kekeliruan umumnya dibagi menjadi "formal" dan "informal." Kekeliruan formal dapat diungkapkan dengan rapi dalam logika formal, seperti logika proposisional, sedangkan kekeliruan informal tidak dapat diungkapkan dalam logika formal.

Kekeliruan Formal

Dalam logika filosofis, kekeliruan formal juga disebut kekeliruan deduktif, kekeliruan logis, atau non sequitur (bahasa Latin untuk "tidak mengikuti"). Ini adalah pola penalaran yang menjadi tidak valid karena cacat dalam struktur logisnya.

Contoh. Diberikan premis bahwa (a) mobil saya adalah mobil, dan (b) beberapa mobil berwarna merah, kita menarik kesimpulan bahwa (c) mobil saya berwarna merah.

Ini adalah contoh tipikal dari kesimpulan yang tidak mengikuti secara logis dari premis atau didasarkan pada data yang tidak relevan. Berikut adalah beberapa kekeliruan logis yang umum.

- Menegaskan **konsekuen**: Argumen apa pun dengan struktur yang tidak valid seperti: Jika A maka B . B , oleh karena itu, A .

Contoh. Jika saya mendapat nilai B pada ujian, maka saya akan mendapat gelar. Saya mendapat gelar, jadi dapat disimpulkan bahwa saya pasti mendapat nilai B . Faktanya, saya mendapat nilai A .

- Menyangkal **antecedens**: Argumen apa pun dengan struktur yang tidak valid seperti: Jika A maka B . Bukan A , oleh karena itu bukan B .

Contoh. Jika itu anjing, maka itu mamalia. Itu bukan anjing, jadi itu pasti bukan mamalia. Faktanya, itu kucing.

- Menegaskan **disjungsi**: Argumen apa pun dengan struktur yang tidak valid: Adalah benar bahwa A atau B . Oleh karena itu, A bukan B .

Contoh. Saya sedang bekerja atau saya sedang di rumah. Saya sedang bekerja, jadi saya pasti tidak sedang di rumah. Faktanya, saya sedang bekerja di rumah.

- Menyangkal **konjungsi**: Argumen apa pun dengan struktur yang tidak valid: Bukan benar bahwa A dan B keduanya benar. Bukan A , oleh karena itu B .

Contoh. Saya tidak mungkin berada di tempat kerja dan di rumah sekaligus. Saya tidak sedang bekerja, jadi saya pasti sedang di rumah. Faktanya, saya sedang berada di taman.

- Tengah yang **tidak terdistribusi**: Argumen apa pun dengan struktur yang tidak valid: Setiap A memiliki B . C memiliki B , jadi C adalah A .

Contoh. Setiap burung memiliki paruh. Makhhluk itu memiliki paruh, jadi makhhluk itu pasti burung. Padahal, makhhluk itu adalah dinosaurus.

Kesalahan formal terjadi ketika struktur argumen tidak benar, meskipun premisnya benar. Argumen yang valid selalu memiliki struktur formal yang benar, dan jika premisnya benar, kesimpulannya pasti benar. Ketika kita menggunakan premis yang salah, kesalahan formal hilang, tetapi argumen tersebut dapat dianggap sebagai kesalahan karena kesimpulannya tidak valid.

Sebagai penerapan modus ponens, contoh berikut tidak mengandung kesalahan formal:

Jika Anda mengambil kursus perbaikan pemutar CD itu langsung setelah lulus SMA, Anda akan sukses dan berlibur di bulan sekarang.

Meskipun tidak ada kesalahan logika dalam argumen, kesimpulannya tidak valid karena premisnya bertentangan dengan fakta. Dengan premis yang salah, Anda dapat membuat kesimpulan apa pun, sehingga pernyataan yang disusun selalu benar. Namun, pernyataan yang selalu benar tidak memiliki nilai dalam penalaran.

Sebaliknya, argumen dengan kekeliruan formal masih dapat memuat semua premis yang benar:

(a) Jika seseorang memiliki berlian terbesar di dunia, maka ia kaya. (b) Raja Salomo kaya. Oleh karena itu, (c) Raja Salomo memiliki berlian terbesar di dunia.

Meskipun (a) dan (b) adalah pernyataan yang benar, (c) tidak mengikuti dari (a) dan (b) karena argumen tersebut melakukan kekeliruan formal "menegaskan konsekuensi".

Kekeliruan Informal

Terdapat banyak jenis kekeliruan informal yang menggunakan hubungan yang salah antara premis dan kesimpulan. Kekeliruan ini dapat dikelompokkan menjadi empat kelompok, dan setiap kelompok berisi beberapa jenis kekeliruan:

- Kesalahan premis yang tidak tepat
- Kesalahan generalisasi yang salah
- Kesalahan sebab yang dipertanyakan
- Kesalahan relevansi

Misalnya, penalaran melingkar, di mana orang yang bernalar memulai dengan apa yang ingin dia capai, termasuk dalam kelompok premis yang tidak tepat. Contoh penalaran melingkar diberikan di bawah ini.

Anda harus mematuhi hukum, karena melanggar hukum adalah ilegal.

Beberapa kesalahan informal tidak termasuk dalam empat kelompok di atas. Misalnya, kesalahan dilema palsu menyajikan pilihan antara dua opsi yang saling eksklusif, yang menyiratkan bahwa tidak ada pilihan lain. Satu opsi jelas lebih buruk daripada yang lain, membuat pilihan tampak jelas. Juga dikenal sebagai kesalahan "pilihan antara dua hal", dilema

palsu adalah jenis kesalahan logika informal di mana argumen yang salah digunakan untuk membujuk audiens agar setuju. Dilema palsu ada di mana-mana.

- Pilih saya atau jalani 4 tahun lagi dengan pajak yang lebih tinggi.
- Amerika: Cintai atau tinggalkan.
- Berlangganan layanan streaming kami atau terjebak dengan TV kabel.

Untuk daftar lengkap kesalahan logika informal, pembaca disarankan untuk membaca halaman Wikipedia tentang topik yang sama untuk detailnya. Memahami kesalahan logika ini dapat membantu kita lebih percaya diri dalam menganalisis argumen yang kita ikuti setiap hari, memisahkan fakta dari fiksi yang disamarkan.

1.4 SEKILAS TENTANG LOGIKA MATEMATIKA

Logika matematika dibagi menjadi empat bidang:

- Teori himpunan
- Teori komputabilitas
- Teori model
- Teori pembuktian

Setiap bidang memiliki fokus yang berbeda, meskipun banyak teknik dan hasil yang sama di antara beberapa bidang. Batasan antara bidang-bidang ini, dan garis antara logika matematika dan bidang matematika lainnya, tidak selalu jelas. Setiap bidang mengandung materi yang kaya yang dapat dipelajari dalam kursus pascasarjana di berbagai tingkatan. Sebagai buku pengantar tentang logika, kami tidak akan memberikan gambaran umum tentang logika matematika, tetapi memilih beberapa konsep dan hasil dasar, yang terkait erat dengan buku ini.

Teori Himpunan

Himpunan adalah model matematika untuk kumpulan nol atau lebih objek yang berbeda. Semua objek dalam suatu himpunan tidak sama (berbeda) dan tidak berurutan. Teori himpunan adalah cabang logika matematika yang mempelajari himpunan dan membahas operasi antar himpunan, relasi antar himpunan, dan pernyataan tentang himpunan. Meskipun semua jenis objek dapat dikumpulkan ke dalam suatu himpunan, teori himpunan paling sering diterapkan pada objek yang relevan dengan matematika. Bahasa teori himpunan dapat digunakan untuk mendefinisikan hampir semua objek matematika. Teori himpunan merupakan bidang yang sangat diminati oleh para ahli logika dan filsuf matematika dan memiliki banyak aplikasi dalam ilmu komputer.

Matematikawan Jerman Georg Cantor umumnya dianggap sebagai pendiri teori himpunan, yang sering disebut sebagai teori himpunan naif. Setelah penemuan paradoks dalam teori himpunan naif, berbagai sistem aksiomatik diusulkan pada awal abad ke-20, di antaranya teori himpunan Zermelo-Fraenkel adalah yang paling terkenal. Teori himpunan Zermelo–Fraenkel memastikan bahwa himpunan yang dirumuskan oleh serangkaian aksioma dalam logika orde pertama cukup umum untuk mewakili semua entitas dalam semesta

pembicaraan, tetapi juga mengecualikan himpunan yang merupakan anggotanya sendiri, sehingga merumuskan teori himpunan yang bebas dari paradoks.

Konsep dan Notasi Dasar

Tujuan utama presentasi ini adalah untuk menunjukkan notasi yang digunakan di seluruh buku.

Orang sering menggunakan N untuk menunjukkan himpunan bilangan asli, yaitu,

$$N = \{0, 1, 2, 3, \dots\}$$

Z untuk menyatakan himpunan bilangan bulat,

$$Z = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$$

dan R untuk menunjukkan himpunan bilangan rasional.

$$R = \{m/n \mid m, n \in N, n \neq 0\}$$

N, Z , dan R adalah himpunan tak hingga secara alami.

Misalkan U adalah kumpulan semua objek yang dipertimbangkan. U sering disebut himpunan universal, atau semesta pembicaraan, atau hanya semesta. Misalnya, U dapat berupa himpunan N dari semua bilangan asli atau populasi dunia. Setiap objek x yang dipertimbangkan adalah anggota U , ditulis sebagai $x \in U$. Suatu himpunan adalah kumpulan objek apa pun dari U . Himpunan kosong sering dilambangkan dengan $\emptyset$ atau hanya $\{\}$. Diberikan dua himpunan A dan B , berikut adalah operasi himpunan umum:

- $x \in A$: ini benar jika dan hanya jika x adalah anggota A .
- $A \cup B$: gabungan dari A dan B . Untuk setiap objek x , $x \in A \cup B$ jika dan hanya jika x berada di A atau B atau keduanya.
- $A \cap B$: irisan dari A dan B . Untuk setiap $x \in U$, $x \in A \cap B$ jika dan hanya jika $x \in A$ dan $x \in B$.
- $A - B$: selisih himpunan dari A dan B . Untuk setiap $x \in U$, $x \in A - B$ jika dan hanya jika $x \in A$ tetapi tidak $x \in B$.
- $\bar{A}$: komplemen dari A (terhadap U). $\bar{A} = U - A$. $A - B$ juga disebut komplemen dari B terhadap A : $A - B = A \cap \bar{B}$.
- $A \subseteq B$: A adalah himpunan bagian dari B . $A \subseteq B$ benar jika dan hanya jika setiap anggota A adalah anggota B . Secara alami, $A \subseteq U$ dan $B \subseteq U$ untuk himpunan semesta U .
- $A \subset B$: A adalah himpunan bagian sejati dari B . $A \subset B$ benar jika dan hanya jika $A \subseteq B$ dan $A \neq B$.
- $A = B$: $A = B$ jika dan hanya jika $A \subseteq B$ dan $B \subseteq A$, yaitu, A dan B mengandung elemen yang persis sama.

- $A \times B$: hasil perkalian Kartesius dari A dan B . Pasangan terurut (a, b) berada di $A \times B$ jika dan hanya jika $a \in A$ dan $b \in B$. Artinya, $A \times B = \{(a, b) \mid a \in A, b \in B\}$.
- A^i : hasil perkalian Kartesius dari i salinan A , di mana $i \in \mathbb{N}$, bilangan asli. Ketika $i = 0$, A^0 menunjukkan barisan kosong, bukan himpunan kosong; $A^1 = A$; $A^2 = A \times A$, $A^3 = A \times A \times A$, dst.
- $|A|$: kardinalitas atau ukuran suatu himpunan. Ketika A terbatas, $|A|$ adalah jumlah anggota dalam A . $|A|$ disebut bilangan kardinal jika A tak terhingga.
- $P(A)$: himpunan kuasa dari A . $P(A)$ adalah himpunan yang anggotanya merupakan himpunan bagian dari A . Secara formal, $P(A) = \{X \mid X \subseteq A\}$. Artinya, untuk setiap himpunan X , $X \in P(A)$ jika dan hanya jika $X \subseteq A$. Ketika $|A| = n$, $|P(A)| = 2^n$.

Teori himpunan dasar dapat dipelajari secara informal dan intuitif dan dapat diajarkan di sekolah dasar menggunakan diagram Venn. Banyak konsep matematika dapat didefinisikan secara tepat hanya menggunakan konsep teori himpunan, seperti relasi dan fungsi.

Relasi

Diberikan dua himpunan A dan B , relasi antara A dan B adalah himpunan bagian dari $A \times B$. Jika $A = B$, yang terjadi dalam banyak aplikasi, kita mengatakan relasi tersebut berada pada himpunan A . Misalnya, relasi kurang dari, $<$, pada $\mathbb{N}$ (himpunan bilangan asli), adalah

$$< = \{\langle 0,1 \rangle, \langle 0,2 \rangle, \langle 0,3 \rangle, \langle 1,2 \rangle, \langle 0,4 \rangle, \langle 1,3 \rangle, \langle 0,5 \rangle, \langle 1,4 \rangle, \langle 2,3 \rangle, \dots\}$$

yang merupakan himpunan bagian dari $\mathbb{N} \times \mathbb{N}$. Secara umum, diberikan relasi R pada himpunan S , yaitu $R \subseteq S \times S$, kita mengasumsikan bahwa $R(a, b)$ benar jika $(a, b) \in R$ dan $R(a, b)$ salah jika $(a, b) \notin R$. Tentu saja, kita terbiasa menulis $<$ sebagai operator infiks, seperti $0 < 1$, bukan operator prefiks seperti $<(0, 1)$. Misalkan $\geq$ adalah relasi pada himpunan S dan a, b, c menyatakan elemen-elemen sembarang dari S .

- Kebalikan dari $\geq$ adalah $\leq$: $a \leq b$ jika dan hanya jika $b \geq a$.
- $\geq$ bersifat refleksif jika $a \geq a$ benar.
- $\geq$ bersifat irefleksif jika $a \geq a$ salah. $\geq$ tidak bisa bersifat refleksif maupun irefleksif.
- $\geq$ bersifat antisimetris jika $a \geq b$ dan $b \geq a$ menyiratkan $a = b$.
- $\geq$ bersifat asimetris jika $b \not\geq a$ ketika $a \geq b$. $\geq$ asimetris jika dan hanya jika bersifat antisimetris dan irefleksif.
- $\geq$ bersifat transitif jika $a \geq b$ dan $b \geq c$ menyiratkan $a \geq c$.
- $\geq$ bersifat Komparatif jika setidaknya salah satu dari berikut ini benar: $a = b$, $a \geq b$, atau $b \geq a$.
- Suatu asimetris merupakan urutan parsial jika bersifat antisimetris dan transitif. Dalam literatur, "urutan parsial" sering disebut sebagai urutan parsial refleksif.
- Suatu urutan parsial irefleksif, yang sering dilambangkan dengan $>$, disebut urutan parsial ketat dan dapat didefinisikan dari suatu urutan parsial $\geq$: $a > b$ jika dan hanya jika $a \geq b$ dan $a \neq b$. Di sisi lain, suatu urutan parsial refleksif $\geq$ dapat didefinisikan dari $>$: $a \geq b$ jika dan hanya jika $a > b$ atau $a = b$. Jika suatu urutan parsial diharuskan bersifat refleksif, maka $>$ tidak dapat menjadi urutan parsial.

- Suatu asimetris merupakan urutan total jika merupakan urutan parsial yang dapat dibandingkan.
- Diberikan suatu urutan parsial, elemen minimal dari $X \subseteq S$ adalah elemen $m \in X$ sedemikian sehingga jika $x \in X$ dan $m \geq x$, maka $x = m$.
- Suatu urutan parsial $\geq$ dikatakan terdefinisi dengan baik jika terdapat elemen minimal $m \in X$ untuk setiap $X \subseteq S$ yang tidak kosong. Secara ekuivalen, $\geq$ dikatakan terdefinisi dengan baik jika dan hanya jika tidak ada urutan tak hingga dari elemen-elemen berbeda $x_1, x_2, \dots, x_i, \dots$, sedemikian sehingga

$$x_1 \geq x_2 \geq \dots \geq x_i \geq \dots$$

- Suatu himpunan $\geq$ disebut himpunan terurut baik jika $\geq$ merupakan himpunan total terurut baik.

Sebagai contoh, relasi umum $\geq$ pada bilangan bulat terurut baik pada himpunan N bilangan asli tetapi tidak terurut baik pada himpunan Z bilangan bulat.

Fungsi

Suatu fungsi f adalah relasi $R \subset A \times B$ dengan batasan bahwa untuk setiap $a \in A$, terdapat paling banyak satu $(a, b) \in R$ untuk beberapa $b \in B$, ditulis $f(a) = b$. Secara ekuivalen, batasan tersebut dapat dinyatakan sebagai: Untuk setiap $a \in A$ dan $b, c \in B$, jika $(a, b) \in R$ dan $(a, c) \in R$, maka $b = c$.

- Fungsi f sering dilambangkan dengan $f: A \rightarrow B$ untuk menunjukkan bahwa f adalah relasi $A \times B$, dan kita mengatakan f adalah fungsi dari A ke B .
- Diberikan $f: A \rightarrow B$, jika $f(a) = b$, kita mengatakan $f(a)$ terdefinisi dan b adalah output dari f pada input a . Jelas, A adalah himpunan nilai input yang mungkin untuk f dan B adalah himpunan nilai output yang mungkin dari f .
- Pemetaan adalah sinonim dari fungsi. Jika $f(a) = b$, kita katakan a dipetakan ke b dan b adalah bayangan a di bawah f .
- Diberikan $f: A \rightarrow B$, $dom(f) = \{x \mid f(x) \text{ terdefinisi}\}$ disebut domain f dan $ran(f) = \{f(x) \mid x \in dom(f)\}$ adalah jangkauan f . Jelas, $dom(f) \in A$ dan $ran(f) \in B$.
- Jika suatu fungsi g mengambil dua nilai sebagai input, misalnya $g(a, a') = b$, kita menyatakan g dengan $g: A \times A' \rightarrow B$ untuk menunjukkan bahwa $g \in A \times A' \times B$. Sesuai konvensi, kita menulis $g(a, a') = b$ alih-alih $g((a, a')) = b$, di mana $a \in A$, $a' \in A'$, dan $b \in B$. Praktik ini digeneralisasikan ke fungsi dengan lebih dari dua input.
- Fungsi $f: A \rightarrow B$ dikatakan total jika $f(a)$ didefinisikan untuk setiap $a \in A$. Jika tidak, f adalah parsial. Jelas, f adalah total jika dan hanya jika $dom(f) = A$. Misalnya, fungsi pembagian pada bilangan bulat bukanlah total karena hasil bagi tidak didefinisikan ketika pembaginya nol. Orang umumnya menganggap bahwa suatu fungsi adalah total kecuali disebutkan secara eksplisit.

- Ketika A dan B sama-sama terbatas, himpunan semua fungsi dari A ke B juga terbatas. Misalkan $F = \{f \mid f : A \rightarrow B, f \text{ adalah total}\}$, maka $|F| = nm$ jika $|A| = m$ dan $|B| = n$. Ini karena ada m pilihan untuk $a \in A$ dan ada n pilihan untuk setiap $f(a)$. Misalnya, jika $B = \{0, 1\}$ dan $A = B \times B$, maka $|B| = 2$, $|A| = 4$ dan jumlah total fungsi dari A ke B adalah $2^4 = 16$. Dengan kata lain, ada 16 fungsi Boolean biner total yang berbeda. Jumlah semua fungsi dari A ke B sama dengan jumlah semua fungsi total dari A ke $B \cup \{u\}$, di mana $u \notin B$ dan $f(a) = u$ berarti " $f(a)$ tidak terdefinisi untuk $f : A \rightarrow B$ ".
- Fungsi $f : A \rightarrow B$ dikatakan *surjektif* jika f adalah fungsi total dan $\text{ran}(f) = B$. Artinya, untuk setiap $b \in B$, terdapat $a \in A$ sedemikian sehingga $f(a) = b$. Fungsi surjektif juga disebut *onto*.
- Fungsi $f : A \rightarrow B$ dikatakan *injektif*, juga disebut *satu – ke – satu*, jika untuk setiap $x, y \in A$ dan $w, z \in B$, jika $x \neq y$, $f(x) = w$, dan $f(y) = z$, maka $w \neq z$. Jika f injektif, maka invers dari f , yang dilambangkan dengan f^{-1} , adalah fungsi injektif dari B ke A : $f^{-1}(y) = x$ jika dan hanya jika $f(x) = y$.
- Fungsi $f : A \rightarrow B$ dikatakan *bijektif* jika f bersifat injektif dan surjektif. Fungsi bijektif juga disebut korespondensi satu-ke-satu atau bijeksi antara A dan B . Dalam hal ini, kita mengatakan A dan B memiliki ukuran yang sama dan menulis $|A| = |B|$.

Dalam definisi di atas, fungsi surjektif harus bersifat total. Jika persyaratan ini dihilangkan, maka sebuah bijeksi harus bersifat total, karena jika sebuah bijeksi adalah fungsi parsial, inversnya tidak dapat berupa bijeksi dan bijeksi semacam itu tidak berguna. Misalnya, definisikan $f : P(N) \rightarrow N$ dengan $f(\{x\}) = x$ untuk semua $x \in N$, maka f bersifat injektif dan surjektif (jika tidak diharuskan bersifat total), dan invers dari f tidak bersifat surjektif.

Proposisi 1.3.1 $f : A \rightarrow B$ bersifat bijektif jika dan hanya jika f dan invers dari f keduanya merupakan fungsi total.

Pembuktiannya diserahkan sebagai latihan.

Himpunan Terhitung

Misalkan S adalah himpunan apa pun. Jika S berhingga, $|S|$ adalah bilangan dalam N , dan kita dapat membandingkan bilangan-bilangan ini dengan mudah. Bagaimana cara membandingkan ukuran himpunan tak hingga? Secara intuitif, jika X adalah himpunan bagian yang tepat dari S , maka $|X| < |S|$. Namun, intuisi ini hanya benar untuk himpunan berhingga. Misalnya, misalkan E adalah himpunan semua bilangan asli genap, maka $E \subset N$. Namun, salah jika mengatakan $|E| < |N|$. Misalkan $f : N \rightarrow E$ didefinisikan oleh $f(i) = 2i$. Maka f bersifat bijektif; N dan E memiliki ukuran yang sama, yaitu $|N| = |E|$ meskipun $E \subset N$.

Paradoks Hotel Hilbert David Hilbert membayangkan sebuah hotel dengan kamar bernomor $0, 1, 2$, dan seterusnya tanpa batas atas dan setiap kamar terisi. Masih ada sejumlah pengunjung baru yang tak terhingga jumlahnya dan masing-masing mengharapkan kamar mereka sendiri. Manajer kemudian memerintahkan semua tamu di kamar i untuk pindah secara bersamaan ke kamar $2i$. Dengan demikian, semua kamar bernomor ganjil menjadi

tersedia untuk pengunjung baru yang masing-masing dapat memiliki kamar mereka sendiri di hotel ini.

Paradoks ini menunjukkan bahwa pemikiran kita sangat berbeda ketika berurusan dengan himpunan tak terhingga. Himpunan bagian yang tepat dapat memiliki ukuran yang sama dengan himpunan bagian yang lebih besar. Cantor memperkenalkan konsep berikut untuk perbandingan himpunan tak hingga.

Definisi 1.3.2 Suatu himpunan S dapat dihitung jika S berhingga atau $|S| = |\mathbb{N}|$. Ketika $|S| = |\mathbb{N}|$, bijeksi $r: S \rightarrow \mathbb{N}$ disebut fungsi rank dari S dan $r(x)$ adalah rank dari $x \in S$. S tidak dapat dihitung jika S tidak dapat dihitung. S dapat dihitung tak hingga jika S dapat dihitung dan tak hingga.

Proposisi 1.3.3 (a) Semua himpunan tak hingga terhitung memiliki ukuran yang sama, dan setiap himpunan tak hingga terhitung memiliki fungsi rank.
(b) Jika r adalah fungsi rank dari himpunan tak hingga terhitung S , maka r menginduksi urutan $\succsim$ total yang terdefinisi dengan baik pada S : $x \succsim y$ jika dan hanya jika $r(x) \geq r(y)$.

Buktinya diserahkan sebagai latihan.

Diberikan suatu himpunan S , kita dapat menulis $S = \{a_0, a_1, a_2, \dots\}$ jika S adalah himpunan tak hingga terhitung. Dalam notasi ini, diasumsikan fungsi rank $r: S \rightarrow \mathbb{N}$ sedemikian sehingga $r(a_i) = i$ untuk setiap $i \in \mathbb{N}$. Karena $\mathbb{N} - \{0\}$ juga merupakan himpunan tak hingga terhitung, kita juga dapat menulis $S = \{a_1, a_2, a_3, \dots\}$. Jadi, himpunan bilangan asli dapat mengandung 0 atau tidak, dan rank dapat dimulai dengan 0 atau 1. Jika S tidak terhitung, salah jika menulis $S = \{a_0, a_1, a_2, \dots\}$. Saat ini, himpunan terhitung membentuk dasar matematika diskrit. Pertama-tama kita periksa himpunan mana yang terhitung.

Proposisi 1.3.4 Himpunan-himpunan berikut ini terhitung:

1. Setiap himpunan hingga
2. $\mathbb{N} - \{k\}$, di mana $k \in \mathbb{N}$
3. $A \times B$, jika A dan B keduanya merupakan himpunan terhitung
4. $\mathbb{Z}$, himpunan semua bilangan bulat
5. $\{0, 1\}^*$, himpunan semua string biner dengan panjang hingga
6. Σ^* , himpunan semua string dengan panjang hingga yang dibangun di atas simbol-simbol dari Σ , sebuah alfabet terbatas

Bukti Beberapa bukti berupa sketsa.

1. Ini berasal dari definisi.
2. Bijeksi $f: \mathbb{N} \rightarrow (\mathbb{N} - \{k\})$ dapat didefinisikan sebagai berikut: $f(i) = i$ jika $i < k$, jika tidak $i + 1$. Maka $f^{-1}(x) = x$ jika $x < k$, jika tidak $x - 1$. Baik f maupun f^{-1} adalah fungsi total.

3. $A \times B$ dapat dihitung ketika A atau B terbatas, dan pembuktiannya diserahkan sebagai latihan. Ketika A dan B keduanya tak hingga terhitung, tanpa mengurangi keumuman, kita mengasumsikan $A = B = N$. Misalkan $g(k) = k(k+1)/2$, yang merupakan jumlah dari $k+1$ bilangan asli pertama, dan $f: N \times N \rightarrow$ adalah fungsi:

$$f(i, j) = g(i+j) + j$$

Invers dari f ada: Untuk setiap $k \in N$, misalkan $m = h(k)$ adalah bilangan bulat sedemikian sehingga $g(m) \leq k < g(m+1)$. Maka $f^{-1}(k) = (i, j)$, di mana $i = g(m) + m - k = g(h(k)) + h(k) - k$ dan $j = k - g(m) = k - g(h(k))$. Mudah untuk memeriksa bahwa $i+j = m$ dan $f(i, j) = k$. Misalnya, jika $k = 12$, maka $m = 4$, dan $i = j = 2$. Tabel 1.1 menunjukkan beberapa nilai pertama dari $k = f(i, j)$. Karena f dan f^{-1} keduanya merupakan fungsi total, maka f bersifat bijektif. Jadi, $N \times N$ dan N memiliki ukuran yang sama.

4. Untuk menunjukkan bahwa Z dapat dihitung, kita mendefinisikan sebuah bijeksi f dari Z ke N : $f(x) =$ jika $x \geq 0$ maka $2x$ jika tidak maka $-2x - 1$. Fungsi ini memetakan 0 ke 0 , bilangan bulat positif ke bilangan asli genap, dan bilangan bulat negatif ke bilangan asli ganjil. f bersifat bijektif karena baik f maupun f^{-1} adalah fungsi total: Untuk setiap $n \in N$, $f^{-1}(n) =$ jika n genap maka $n/2$ jika tidak maka $(-n-1)/2$. Perhatikan bahwa $\geq$ pada Z tidak terdefinisi dengan baik dan tidak dapat digunakan untuk mendaftar Z dari yang terkecil hingga yang lebih besar. Dengan menggunakan f sebagai fungsi peringkat Z , kita mendaftar Z berdasarkan peringkat setiap elemen sebagai berikut: $0, -1, 1, -2, 2, -3, 3, \dots$
5. Misalkan $S = \{0, 1\}^*$ adalah himpunan semua string biner dengan panjang terbatas. Untuk menunjukkan bahwa S dapat dihitung, mari kita temukan sebuah bijeksi dari S ke N . Untuk setiap string biner $s \in S$, misalkan panjang s adalah n , yaitu $|s| = n$. Terdapat $1 + 2 + 2^2 + \dots + 2^{n-1} = 2^n - 1$ string dalam S yang panjangnya kurang dari $|s|$. Misalkan $v(s)$ adalah nilai desimal dari s : $v(\epsilon) = 0$, di mana ϵ adalah string kosong, $v(x0) = 2v(x)$ dan $v(x1) = 2v(x) + 1$, maka terdapat $v(s)$ string dengan panjang n yang nilai desimalnya kurang dari $v(s)$. Definisikan $\gamma: S \rightarrow N$ dengan $\gamma(s) = 2^{|s|} + v(s) - 1$, maka γ haruslah merupakan bijeksi, karena baik γ maupun $\gamma^{-1}: N \rightarrow \{0, 1\}^*$ adalah fungsi total: Untuk setiap $x \in N$, misalkan $n = \lfloor \log_2(x+1) \rfloor$, maka $\gamma^{-1}(x) = s$ sedemikian sehingga $|s| = n$ dan $v(s) = x - 2^n + 1$.
6. Misalkan $\Sigma = \{a_0, a_1, \dots, a_{k-1}\}$. Jika $k = 1$, maka $\Sigma^* = \{a_0^i \mid i \in N\}$ dapat dihitung. Jika $k \geq 2$, kita generalisasikan bukti pada masalah sebelumnya dari $\{0, 1\}^*$ ke Σ^* . Misalkan $\theta: \Sigma \rightarrow N$ adalah $\theta(a_i) = i$ untuk $0 \leq i \leq k-1$. Setiap string $s \in \Sigma^*$ adalah ϵ atau xa , di mana $x \in \Sigma^*$ dan $a \in \Sigma$. Nilai desimal dari $s \in \Sigma^*$, $v(s)$, dihitung sebagai berikut: $v(\epsilon) = 0$ dan $v(xa) = v(x) * k + \theta(a)$. Jika $|s| = n$, terdapat $(k^n - 1)/(k - 1)$ string Σ^* yang panjangnya kurang dari n , dan terdapat $v(s)$ string dengan panjang n yang nilai desimalnya kurang dari $v(s)$. Definisikan $\gamma: \Sigma^* \rightarrow N$

dengan $\gamma(s) = (k^{|s|} - 1)/(k - 1) + v(s)$, maka mudah untuk memeriksa bahwa baik γ maupun γ^{-1} (sebuah latihan) adalah fungsi total.

Tabel 1.1 $k = f(i, j) = (i + j)(i + j + 1)/2 + j$

i/j	0	1	2	3	4
0	0	2	5	9	14
1	1	4	8	13	19
2	3	7	12	18	25
3	6	11	17	24	32
4	10	16	23	31	40

Tiga contoh terakhir adalah contoh himpunan tak hingga terhitung. Σ^* serta $\{0, 1\}^*$ dalam dua contoh terakhir memainkan peran penting dalam teori komputasi. Jika kita mengurutkan string Σ^* berdasarkan γ , yaitu, mendaftarkannya sebagai $\gamma^{-1}(0), \gamma^{-1}(1), \gamma^{-1}(2)$, dan seterusnya, urutan ini akan memilih (a) panjang yang lebih pendek terlebih dahulu; (b) untuk string dengan panjang yang sama, nilai desimal yang lebih kecil terlebih dahulu. Dalam urutan ini, $s \in \Sigma^*$ didahului oleh $(k^{|s|} - 1)/(k - 1)$ string yang panjangnya kurang dari $|s|$, ditambah $v(s)$ string dengan panjang $|s|$ dan nilainya kurang dari $v(s)$. Oleh karena itu, rank dari s dalam urutan ini tepat $\gamma(s) = (k^{|s|} - 1)/(k - 1) + v(s)$.

Definisi 1.3.5 Misalkan $\gamma: \Sigma^* \rightarrow N$ adalah fungsi rank yang didefinisikan dalam Proposisi 1.3.4(6). Urutan yang diinduksi oleh γ disebut urutan kanonik dari Σ^* .

Sebagai contoh, $\{0, 1\}^*$ diurutkan berdasarkan urutan kanonik sebagai berikut:

$$\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, \dots\}$$

Urutan kanonik Σ^* didefinisikan secara unik oleh fungsi peringkat γ . Kita dapat berbicara secara tidak berbeda tentang bilangan asli dan string karena $w \in \Sigma^*$ dapat dipandang sebagai $\gamma(w) \in N$ dan $n \in N$ sebagai $\gamma^{-1}(n) \in \Sigma^*$

Contoh 1.3.6 Misalkan $\Sigma = \{a, b\}$, maka beberapa string pertama dari Σ^* diberikan di bawah ini dengan γ , di mana a^i menunjukkan i salinan dari a :

$x \in \Sigma^*$	ϵ	a	b	a^2	ab	ba	b^2	a^3	a^2b	aba	ab^2	ba^2	bab	b^2a	b^3	a^4	a^3b	$\dots$
$\gamma(x)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	$\dots$

Σ^* menjadi string biner jika kita mengganti a dengan 0 dan b dengan 1,

Himpunan Tak Terhitung

Pada tahun 1874, dalam artikel teori himpunan pertamanya, Georg Cantor memperkenalkan istilah "himpunan terhitung" dan membuktikan bahwa himpunan bilangan

real tidak terhitung, sehingga menunjukkan bahwa tidak semua himpunan tak terhingga dapat dihitung. Pada tahun 1878, ia menggunakan bukti bijektif untuk mengidentifikasi himpunan tak terhingga dengan ukuran yang sama, membedakan himpunan yang terhitung dari himpunan yang tak terhitung dengan metode diagonal yang terkenal.

Proposisi 1.3.7 Himpunan-himpunan berikut ini tidak terhitung:

1. B : himpunan string biner dengan panjang tak terhingga
2. R_1 : himpunan bilangan real non-negatif kurang dari 1
3. $P(N)$: himpunan kuasa dari bilangan asli
4. $D_F = \{f \mid f: N \rightarrow \{0, 1\}\}$ adalah himpunan total, yaitu himpunan fungsi total dari bilangan asli ke nilai Boolean, yang disebut fungsi keputusan atau fungsi karakteristik.

Bukti Untuk himpunan B dari string biner dengan panjang tak terbatas, kita akan menggunakan metode diagonal Cantor untuk menunjukkan bahwa B tidak terhitung dengan bukti sanggahan. Asumsikan bahwa B terhitung, maka ada bijeksi $f: N \rightarrow B$ dan kita tulis $B = \{s_0, s_1, s_2, \dots\}$, di mana $s_i = f(i)$. Misalkan simbol ke- j dari s_i adalah s_{ij} . Sekarang, kita membangun string biner x dengan panjang tak terbatas sebagai berikut: Simbol ke- j dari x , yaitu x_j , adalah komplemen dari s_{jj} . Artinya, jika $s_{jj} = 0$, maka $x_j = 1$; jika $s_{jj} = 1$, maka $x_j = 0$. Jelas, $x \in B$. Misalkan $x = s_k$ untuk beberapa $k \in N$. Namun, x berbeda dari s_k pada simbol ke- k , yaitu $x_k = \overline{s_{kk}}$, komplemen dari s_{kk} , di mana $\overline{0} = 1$ dan $\overline{1} = 0$. Kontradiksi muncul dari asumsi bahwa B dapat dihitung. Tabel 1.2 mengilustrasikan gagasan metode diagonal Cantor.

Tabel 1.2 Ilustrasi metode diagonal Cantor: Jika B dapat diurutkan sebagai $s_0, s_1, s_2, \dots$, maka $x = \overline{s_{00}}, \overline{s_{11}}, \overline{s_{22}}, \dots$ adalah string biner tetapi tidak dapat berada di B , di mana $\overline{0} = 1$ dan $\overline{1} = 0$

$s_0 =$	0	0	0	0	0	0	0	0	0	0	...
$s_1 =$	0	0	0	0	0	0	0	0	1	0	...
$s_2 =$	0	0	0	0	0	1	0	0	0	0	...
$s_3 =$	0	0	0	1	0	0	0	0	0	0	...
$s_7 =$	0	0	0	1	0	1	0	0	0	0	...
$s_6 =$	0	0	0	0	1	0	0	0	0	0	...
$s_4 =$	0	0	0	1	0	1	0	1	0	0	...
$s_5 =$	0	0	1	0	0	0	0	1	0	0	...
$s_8 =$	0	1	0	0	1	0	0	1	0	0	...
$s_9 =$	0	1	0	0	1	1	0	1	0	1	...
...											
x	1	1	1	0	1	1	1	0	1	0	...

Setelah kita mengetahui bahwa B tidak dapat dihitung, jika kita memiliki bijeksi dari B ke R_1 , maka R_1 harus tidak dapat dihitung. Fungsi $f: B \rightarrow R_1$ didefinisikan oleh $f(s) = 0.s$, karena setiap angka R_1 dapat direpresentasikan oleh bilangan biner, di mana $s \in B$. Jelas bahwa $f^{-1}(0.s) = s$, dengan asumsi urutan tak hingga dari 0 ditambahkan ke ujung kanan s jika s adalah urutan terbatas. Karena f dan f^{-1} adalah fungsi total, maka f memang merupakan bijeksi.

Untuk menunjukkan bahwa $P(N)$ tidak terhitung, misalkan $g: B \rightarrow P(N)$ adalah $g(s) = \{j \mid \text{simbol ke-}(j+1) \text{ dari } s \text{ adalah } 1\}$. Misalnya, jika $s = 1101000, \dots$ (sisanya adalah nol), maka $g(s) = \{0, 1, 3\}$. Jika $g(s) = A \in P(N)$, maka $g^{-1}(A) = \lambda(0)\lambda(1)\lambda(2) \dots \in B$, di mana, untuk setiap $i \in N$, $\lambda(i) = 1$ jika $i \in A$ dan 0 jika $i \notin A$. Jelas, g bersifat bijektif karena baik g maupun g^{-1} adalah fungsi total. Dalam literatur, $g^{-1}(A)$ disebut barisan karakteristik dari A .

Demikian pula, untuk menunjukkan bahwa himpunan D_F dari fungsi keputusan tidak terhitung, misalkan $h: B \rightarrow D_F$ adalah $h(s) = f$, di mana, untuk setiap $j \in N$, $f(j) = \text{simbol ke-}(j+1) \text{ dari } s$. Karena s adalah string biner dengan panjang tak terbatas, $f: N \rightarrow \{0, 1\}$ terdefinisi dengan baik. Misalnya, jika s adalah string yang seluruhnya terdiri dari nol, maka $f(j) = 0$ untuk semua j . Jika $h(s) = f$, maka $h^{-1}(f) = f(0)f(1)f(2) \dots \in B$. Jelas, h bersifat bijektif karena baik h maupun h^{-1} adalah fungsi total.

Fungsi keputusan total $f = h(s)$, $s \in B$, juga disebut fungsi karakteristik, karena $s = g^{-1}(A)$, $A \in P(N)$, adalah barisan karakteristik dari $A \subset N$, di mana h dan g didefinisikan dalam bukti di atas.

Hasil bahwa $P(N)$ tidak terhitung sedangkan N terhitung dapat digeneralisasikan: Untuk setiap himpunan S , himpunan kuasa dari S , yaitu $P(S)$, memiliki ukuran yang lebih besar daripada S . Hasil ini menyiratkan bahwa gagasan himpunan semua himpunan adalah gagasan yang tidak konsisten. Jika S adalah himpunan semua himpunan, maka $P(S)$ akan sekaligus lebih besar daripada S dan merupakan himpunan bagian dari S . Cantor memilih simbol $\aleph_0$ untuk ukuran N , $|N|$, sebuah bilangan kardinal. Simbol $\aleph_0$ dibaca aleph-nol, di mana $\aleph$ adalah huruf pertama dari alfabet Ibrani. Ukuran bilangan real sering dilambangkan dengan $\aleph_1$, atau c untuk kontinum bilangan real, karena himpunan bilangan real lebih besar daripada N . Cantor menunjukkan bahwa ada tak terhingga banyak bilangan kardinal tak terhingga, dan tidak ada bilangan kardinal terbesar.

$$\begin{aligned}\aleph_0 &= |N| \\ \aleph_1 &= |P(N)| &= 2^{\aleph_0} > \aleph_0 \\ \aleph_2 &= |P(P(N))| &= 2^{\aleph_1} > \aleph_1 \\ \aleph_3 &= |P(P(P(N)))| &= 2^{\aleph_2} > \aleph_2 \\ &\dots\end{aligned}$$

Contoh inkonsisten yang terkenal dari teori himpunan naif adalah apa yang disebut paradoks Russell, yang ditemukan oleh Bertrand Russell pada tahun 1901:

Paradoks Russell Misalkan $T = \{S \mid S \notin S\}$, maka $T \in T \equiv T \notin T$.

Artinya, jika " $T \in T$ " salah, maka " $T \in T$ " benar menurut definisi T karena kondisi " $S \notin S$ " benar dan $T = S \in T$. Jika " $T \in T$ " benar, maka " $T \notin T$ " juga benar karena setiap anggota T memenuhi $S \notin S$. Paradoks ini menunjukkan bahwa beberapa upaya formalisasi teori himpunan naif yang dibuat oleh Cantor dkk. menghasilkan kontradiksi. Paradoks Russell menunjukkan bahwa konsep "suatu himpunan mengandung dirinya sendiri" tidak valid. Jika " $S \in S$ " selalu salah, maka $S \notin S$ selalu benar dan $T = \{S \mid S \notin S\}$ adalah "himpunan dari semua himpunan," yang juga tidak ada. Versi awam dari paradoks Russell disebut paradoks tukang cukur.

Paradoks Tukang Cukur: Di sebuah desa, hanya ada satu tukang cukur yang mencukur semua orang, dan hanya mereka, yang tidak mencukur diri sendiri. Pertanyaannya adalah, apakah tukang cukur itu mencukur dirinya sendiri?

Menjawab pertanyaan ini menghasilkan kontradiksi. Tukang cukur tidak dapat mencukur dirinya sendiri karena ia hanya mencukur mereka yang tidak mencukur diri sendiri. Sebaliknya, jika tukang cukur tidak mencukur dirinya sendiri, maka ia termasuk dalam kelompok orang yang akan dicukur oleh tukang cukur, dan dengan demikian, sebagai tukang cukur, ia harus mencukur dirinya sendiri.

Penemuan paradoks dalam teori himpunan naif menyebabkan para matematikawan bertanya-tanya apakah matematika itu sendiri tidak konsisten, dan mencari bukti konsistensi. Ernst Zermelo (1904) memberikan bukti bahwa setiap himpunan dapat diurutkan dengan baik, menggunakan aksioma pilihan, yang memicu perdebatan dan penelitian yang sengit di kalangan matematikawan. Pada tahun 1908, Zermelo memberikan serangkaian aksioma pertama untuk teori himpunan. Aksioma-aksioma ini, bersama dengan aksioma penggantian yang diusulkan oleh Abraham Fraenkel, sekarang disebut teori himpunan Zermelo–Fraenkel (ZF). Selain ZF, banyak teori himpunan lain telah diusulkan sejak saat itu, untuk menghilangkan paradoks dari teori himpunan naif.

Teori Komputabilitas

Teori komputabilitas, yang dulunya disebut teori rekursi, adalah cabang dari logika matematika, ilmu komputer, dan teori komputasi yang berawal pada tahun 1930-an dengan studi tentang fungsi yang dapat dihitung. Bidang ini kemudian berkembang mencakup studi tentang komputabilitas umum. Di bidang-bidang ini, teori komputabilitas tumpang tindih dengan teori himpunan dan teori pembuktian.

Rekursi

Alasan utama untuk nama "teori rekursi" adalah karena rekursi digunakan untuk membangun objek dan fungsi.

Contoh 1.3.8 Diberikan simbol konstanta 0 dari himpunan T dan simbol fungsi $suc : T \rightarrow T$, objek-objek dari himpunan T dapat dibangun secara rekursif sebagai berikut:

1. 0 adalah objek dari T .
2. Jika n adalah objek dari T , maka $suc(n)$ juga merupakan objek dari T .

3. Tidak ada yang lain yang akan menjadi objek dari T .

Jika kita misalkan $suc^0(0) = 0, suc^1(0) = suc(0), suc^2(0) = suc(suc(0))$, dan seterusnya, maka T dapat dinyatakan sebagai:

$$T = \{0, suc(0), suc^1(0), suc^2(0), suc^3(0), \dots, suc^i(0), \dots\}$$

Jelas, ada bijeksi f antara T dan N : $f(suc^i(0)) = i$, di mana N adalah himpunan bilangan asli. Artinya, untuk setiap $i \in N$, $f^{-1}(i) = suc^i(0)$. Dengan memberikan suc arti penerus bilangan asli, para matematikawan menggunakan T sebagai definisi N .

Dalam ZF (teori himpunan Zermelo–Fraenkel), N dapat didefinisikan oleh aksioma pasangan dan tak hingga:

$$\begin{aligned} 0 &= \{\} &= \emptyset \\ 1 &= \{0\} &= \{\emptyset\} \\ 2 &= \{0, 1\} &= \{\emptyset, \{\emptyset\}\} \\ 3 &= \{0, 1, 2\} &= \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ \dots & & \dots \end{aligned}$$

Anggota N disebut ordinal von Neumann di ZF.

Fungsi dapat didefinisikan secara rekursif dengan cara yang analog. Misalkan pre, add, sub , dan mul adalah fungsi pendahulu, penjumlahan, pengurangan, dan perkalian pada himpunan bilangan asli:

$$\begin{aligned} pre(0) &= 0 \\ pre(s(x)) &= x \\ add(0, y) &= y \\ add(s(x), y) &= s(add(x, y)) \\ sub(x, 0) &= x \\ sub(x, s(y)) &= sub(pre(x), y) \\ mul(0, y) &= 0 \\ mul(s(x), y) &= add(mul(x, y), y) \end{aligned}$$

Perhatikan bahwa pengurangan yang didefinisikan di atas berbeda dari pengurangan pada bilangan bulat: Kita memiliki $sub(s^i(0), s^j(0)) = 0$ ketika $i \leq j$. Fungsi ini memiliki sifat bahwa $sub(x, y) = 0$ jika dan hanya jika $x \leq y$.

Fungsi-fungsi di atas adalah contoh dari apa yang disebut fungsi rekursif primitif, yang merupakan fungsi total dan anggota dari himpunan yang disebut fungsi rekursif umum. Fungsi rekursif primitif dapat dihitung oleh program komputer yang menggunakan batas atas (ditentukan sebelum memasuki loop) untuk jumlah iterasi setiap loop. Himpunan fungsi rekursif umum (juga disebut fungsi rekursif parsial) setara dengan fungsi-fungsi yang dapat dihitung oleh mesin Turing.

Bentuk Backus–Naur (BNF)

Dalam ilmu komputer, bentuk Backus–Naur (BNF) adalah teknik notasi untuk tata bahasa bebas konteks, yang sering digunakan untuk mendeskripsikan sintaks bahasa yang digunakan dalam komputasi, seperti bahasa pemrograman komputer, format dokumen, set instruksi, dan protokol komunikasi. John Backus dan Peter Naur adalah orang pertama yang menggunakan notasi ini untuk mendeskripsikan Algol, sebuah bahasa pemrograman awal. BNF diterapkan di mana pun deskripsi formal bahasa dibutuhkan. Kita akan menggunakan BNF untuk mendeskripsikan sintaks logika sebagai sebuah bahasa.

BNF dapat diperluas untuk mendefinisikan objek yang dibangun secara rekursif dengan gaya ringkas. Misalnya, himpunan bilangan asli dinyatakan dalam BNF sebagai berikut, di mana *suc* menunjukkan fungsi penerus:

$$\langle N \rangle ::= 0 \mid \textit{suc}(\langle N \rangle)$$

Simbol $::=$ dapat dipahami sebagai “didefinisikan oleh,” dan “ $\mid$ ” memisahkan bagian-bagian alternatif dari sebuah definisi. Rumus di atas dapat dibaca sebagai berikut: anggota N adalah 0, atau *suc* yang berlaku untuk (anggota lain) dari N ; tidak ada yang lain yang akan ada di N . Dari contoh ini, kita melihat bahwa untuk mendefinisikan himpunan N , kita menggunakan beberapa item di sebelah kanan $::=$. Untuk definisi rekursif, kita memiliki kasus dasar (seperti 0) dan kasus rekursif (seperti $\textit{suc}(\langle N \rangle)$), di mana $\langle N \rangle$ menunjukkan anggota dari N .

Simbol seperti $\langle N \rangle$ disebut variabel dalam tata bahasa bebas konteks dan simbol seperti 0, *suc*, (, dan) disebut terminal. Tata bahasa bebas konteks menentukan himpunan string terminal apa yang dapat diturunkan untuk setiap variabel. Untuk contoh di atas, himpunan string yang diturunkan dari BNF adalah

$$N = \{0, \textit{suc}(0), \textit{suc}'(0), \textit{suc}^3(0), \dots, \textit{suc}^i(0) \dots\}$$

Himpunan ini sama dengan objek yang didefinisikan dalam Contoh 1.3.8, tetapi BNF memberi kita definisi formal dan ringkas.

Contoh 1.3.9 Untuk mendefinisikan himpunan Σ^* dari semua string (dengan panjang terbatas) di atas alfabet Σ , kita dapat menggunakan BNF. Misalnya, jika $\Sigma = \{a, b, c\}$, maka Σ^* adalah himpunan semua string dengan panjang terbatas:

$$\begin{aligned} \langle \textit{symbol} \rangle &::= a \mid b \mid c \\ \langle \Sigma^* \rangle &::= \epsilon \mid \langle \textit{symbol} \rangle \langle \Sigma^* \rangle \end{aligned}$$

Artinya, string kosong $\epsilon \in \Sigma^*$; jika $w \in \Sigma^*$, untuk setiap simbol $s \in \Sigma$, $sw \in \Sigma^*$; tidak ada yang lain yang akan ada di Σ^* . Dengan demikian, Σ^* berisi tepat semua string yang dibangun di atas Σ :

$$\Sigma^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \dots\}$$

Dengan cara ini, kita dapat membangun Σ^* untuk alfabet terbatas Σ .

Karena ada korespondensi satu-ke-satu antara string dan urutan, urutan juga dapat didefinisikan secara rekursif dengan cara yang analog. Bahkan, semua objek yang dibangun secara rekursif dapat didefinisikan dengan cara ini.

Fungsi yang Dapat Dihitung

Salah satu pertanyaan mendasar yang dibahas oleh teori komputabilitas adalah sebagai berikut:

Apa artinya fungsi pada bilangan asli dapat dihitung?

Ada dua jawaban berbeda untuk pertanyaan sederhana ini:

- Suatu fungsi dapat dihitung jika ada model komputasi (juga disebut perangkat komputasi, atau mesin), untuk menghitungnya.
- Suatu fungsi dapat dihitung jika ada algoritma yang menghitung fungsi tersebut.

Membedakan antara dua arti "fungsi yang dapat dihitung" sangat penting dalam teori komputabilitas. Untuk menghindari kebingungan, kita akan berpegang pada arti pertama dari "fungsi yang dapat dihitung" dan menyebut fungsi dari arti kedua sebagai "fungsi yang dapat dihitung secara total."

Para ahli logika dan ilmuwan komputer memperkenalkan berbagai model komputasi. Model komputasi ini dapat berupa fungsi rekursif umum Gödel, mesin Turing, kalkulus λ Church, teori rekursi Kleene, sistem kanonik Post, tata bahasa Chomsky, mesin penghitung Minsky, atau model von Neumann.

Menurut tesis Church–Turing (juga dikenal sebagai tesis komputabilitas, dugaan Church–Turing), jika suatu fungsi pada bilangan asli dapat dihitung oleh suatu model komputasi, maka fungsi tersebut dapat dihitung oleh mesin Turing. Dengan asumsi tesis Church–Turing benar, kita akan menggunakan mesin Turing sebagai model komputasi kita. Mesin Turing dapat berulang tanpa henti pada beberapa input dan fungsi yang dihitung karenanya bersifat parsial karena perulangan ini. Jadi, yang dimaksud dengan "fungsi yang dapat dihitung" adalah fungsi total atau parsial yang dihitung oleh mesin Turing.

Secara historis, algoritma didefinisikan sebagai urutan instruksi langkah demi langkah yang terbatas dan telah dibahas oleh banyak orang dalam berbagai model komputasi. Algoritma dapat berupa program, prosedur, atau mesin Turing. Dalam studi algoritma saat ini, cakupan algoritma dipersempit menjadi program-program yang harus berhenti pada setiap input. Artinya, dengan algoritma, kita maksudkan bahwa proses komputasi akan berhenti dalam sejumlah langkah yang terbatas. Untuk arti lama dari "algoritma," kita akan menggunakan "prosedur" sebagai gantinya.

Dengan demikian, "fungsi komputasi total" lebih ketat daripada "fungsi komputasi" dalam arti pertama: Ini mensyaratkan adanya model komputasi untuk fungsi tersebut dan model komputasi tersebut harus berhenti pada setiap input fungsi. Dalam hal ini, fungsi yang dihitung bersifat total. Artinya, "fungsi komputasi total" dan "fungsi total dan komputasi" memiliki arti yang sama, begitu pula "fungsi komputasi total." Dengan asumsi tesis Church–Turing benar, setiap fungsi komputasi dapat dihitung oleh mesin Turing. Setiap fungsi yang

dapat dihitung secara total dapat dihitung oleh mesin Turing yang berhenti pada setiap input, dan mesin Turing ini akan disebut "algoritma."

Dalam teori fungsi rekursif, "fungsi yang dapat dihitung" setara dengan fungsi rekursif umum, dan "fungsi yang dapat dihitung secara total" adalah himpunan bagian dari semua fungsi rekursif primitif. Perhatikan bahwa fungsi pembagian " / " pada bilangan asli dapat bersifat total dengan mengizinkan $x/0 = 0$.

Definisi 1.3.10 Suatu masalah keputusan adalah fungsi yang menerima input apa pun dari masalah tersebut dan mengembalikan 0 (salah atau tidak) atau 1 (benar atau ya) sebagai output.

Suatu masalah keputusan dapat dihitung jika ada mesin Turing untuk menghitung fungsinya. Suatu masalah keputusan dapat diputuskan jika ada mesin Turing yang menghitungnya dan berhenti pada setiap input. Mesin Turing ini disebut penentu.

Sekarang, fungsi dibagi menjadi tiga kelompok terpisah: (a) dapat dihitung sepenuhnya, (b) dapat dihitung tetapi tidak dapat dihitung sepenuhnya, dan (c) tidak dapat dihitung. Masalah keputusan juga dibagi menjadi tiga kelompok terpisah: (a) dapat diputuskan, (b) dapat dihitung tetapi tidak dapat diputuskan, dan (c) tidak dapat dihitung. Menurut tesis Church–Turing, tidak ada model komputasi yang tersedia untuk fungsi yang tidak dapat dihitung atau masalah keputusan yang tidak dapat dihitung.

Pada Bagian sebelumnya, kita melihat bahwa himpunan fungsi keputusan pada bilangan asli tidak terhitung. Fungsi-fungsi ini termasuk dalam irisan masalah keputusan dan fungsi umum pada bilangan asli. Karena himpunan string biner dengan panjang terbatas dapat dihitung dan terdapat bijeksi antara string biner dan mesin Turing, maka himpunan mesin Turing dapat dihitung. Dengan kata lain, kita hanya memiliki himpunan mesin Turing yang dapat dihitung, tetapi masalah keputusan tidak terhitung. Akibatnya, terdapat sejumlah besar masalah keputusan yang tidak dapat dihitung.

Salah satu masalah yang tidak dapat diputuskan yang penting adalah memutuskan apakah mesin Turing M akan berhenti pada input w . Jika kita mengkodekan M dan w sebagai bilangan bulat positif tunggal yang dilambangkan dengan $\langle M, w \rangle$, dan mendefinisikan fungsi $f(\langle M, w \rangle) = 1$ jika M berhenti pada w , dan 0, jika tidak, maka f adalah fungsi keputusan yang tidak dapat diputuskan.

Sebagai lawan dari tesis Church–Turing, konsep kelengkapan Turing berkaitan dengan apakah model komputasi dapat mensimulasikan mesin Turing. Model komputasi dikatakan lengkap Turing jika dapat mensimulasikan mesin Turing apa pun. Menurut tesis Church–Turing, mesin Turing secara teoritis mampu melakukan semua tugas yang dilakukan oleh komputer; di sisi lain, hampir semua komputer digital lengkap Turing jika batasan memori terbatas diabaikan. Beberapa logika juga lengkap Turing karena dapat digunakan untuk mensimulasikan mesin Turing apa pun. Akibatnya, beberapa masalah seperti masalah penghentian untuk logika ini tidak dapat diputuskan. Teori komputabilitas akan membantu kita untuk memutuskan

apakah ada atau tidak prosedur pengambilan keputusan untuk suatu logika, dan kita akan melihat beberapa contohnya di Bab 11.

Teori Model

Teori model adalah studi tentang hubungan antara teori formal dan modelnya. Teori formal adalah kumpulan kalimat dalam bahasa formal yang mengekspresikan pernyataan. Model adalah struktur matematika (misalnya, grup, medan, aljabar, graf, logika) di mana pernyataan teori tersebut berlaku. Aspek yang diteliti meliputi konstruksi, jumlah dan ukuran model suatu teori, hubungan model yang berbeda satu sama lain, dan interaksinya dengan bahasa formal itu sendiri. Secara khusus, para ahli teori model juga meneliti himpunan yang dapat didefinisikan dalam model suatu teori, dan hubungan himpunan yang dapat didefinisikan tersebut satu sama lain. Setiap bahasa formal memiliki sintaksis dan semantiknya. Model adalah struktur semantik yang terkait dengan struktur sintaksis suatu bahasa formal. Mengikuti pendekatan ini, setiap logika formal didefinisikan di dalam suatu bahasa formal.

Sintaksis dan Semantik

Sintaksis suatu bahasa formal menentukan bagaimana berbagai komponen bahasa, seperti simbol, kata, dan kalimat, didefinisikan. Misalnya, bahasa untuk logika proposisional hanya menggunakan variabel proposisional dan operator logika sebagai simbolnya, dan rumus yang terbentuk dengan baik yang dibangun di atas simbol-simbol ini sebagai kalimatnya. Dalam teori model, sekumpulan kalimat dalam bahasa formal adalah salah satu komponen yang membentuk suatu teori. Bahasa untuk logika sering kali mengandung dua simbol konstan, yaitu 1 dan 0, atau T dan $\perp$, yang masing-masing diinterpretasikan sebagai benar dan salah. Simbol-simbol ini biasanya dianggap sebagai operator logika khusus yang tidak mengambil argumen, bukan variabel proposisional.

Semantik suatu bahasa menentukan makna dari berbagai komponen bahasa tersebut. Misalnya, jika kita menggunakan simbol q untuk menyatakan pernyataan "Hari ini adalah Senin," maka makna q adalah "Hari ini adalah Senin." q dapat digunakan untuk menyatakan seribu pernyataan yang berbeda, seperti seribu Hamlet di mata seribu orang. Di sisi lain, makna formal dari rumus $q \vee r$ adalah nilai kebenaran dari $q \vee r$, yang dapat diputuskan secara unik ketika nilai kebenaran q dan r diberikan. Dalam teori model, makna formal suatu kalimat dieksplorasi: Teori ini memeriksa elemen semantik (makna dan kebenaran) melalui elemen sintaksis (rumus dan bukti) dari bahasa yang sesuai.

Dalam teori model, semantik dan model adalah sinonim. Model suatu teori adalah interpretasi yang memenuhi kalimat-kalimat dari teori tersebut. Aljabar universal sering digunakan sebagai model. Dalam definisi ringkas, yang berasal dari tahun 1973,

$$\text{Teori Model} = \text{Aljabar Universal} + \text{logika} \quad (1.1)$$

Aljabar universal, juga disebut aljabar kategorikal, adalah bidang matematika yang mempelajari struktur aljabar dan model atau aljabarnya.

Aljabar Boolean

Dalam aljabar universal, aljabar yang paling relevan terkait dengan logika yang dibahas dalam buku ini adalah aljabar Boolean. Aljabar Boolean tidak selalu menjadi topik teori model, tetapi sering dikutip sebagai contoh dalam teori model.

Banyak konsep sintaksis aljabar Boolean dapat diterapkan pada logika proposisional hanya dengan sedikit perubahan dalam notasi dan terminologi, sementara semantik logika proposisional didefinisikan melalui aljabar Boolean sedemikian rupa sehingga tautologi (teorema) logika proposisional sesuai dengan teorema persamaan aljabar Boolean.

Dalam aljabar Boolean, nilai variabel adalah nilai kebenaran benar dan salah, biasanya dilambangkan dengan 1 dan 0. Operasi utama aljabar Boolean adalah perkalian “.” (konjungsi), penjumlahan “+” (disjungsi), dan invers i (negasi). Dengan demikian, ini adalah formalisme untuk menggambarkan operasi logika dengan cara yang sama seperti aljabar dasar menggambarkan operasi numerik, seperti penjumlahan dan perkalian. Bahkan, aljabar Boolean adalah himpunan apa pun dengan operasi biner $+$ dan $\cdot$ dan operasi uner i di dalamnya yang memenuhi hukum (persamaan) Boolean, yang mendefinisikan sifat-sifat operasi logika.

Aljabar Boolean diperkenalkan oleh George Boole dalam buku pertamanya, *The Mathematical Analysis of Logic* (1847). Kalimat yang dapat diekspresikan dalam logika proposisional memiliki ekspresi yang setara dalam aljabar Boolean. Dengan demikian, aljabar Boolean terkadang digunakan untuk menunjukkan logika proposisional yang dilakukan dengan cara ini. Namun, aljabar Boolean tidak cukup untuk menangkap rumus logika yang menggunakan kuantifikasi, seperti yang ada dalam logika orde pertama.

1.5 TEORI PEMBUKTIAN

Secara setara dengan teori model, teori pembuktian terletak di area interdisipliner antara matematika, filsafat, dan ilmu komputer. Dalam teori pembuktian, bukti dipelajari sebagai objek matematika formal. Bukti biasanya didefinisikan secara rekursif sebagai struktur data seperti daftar (atau pohon) rumus. Dalam sistem logika, aksioma dan aturan inferensi disediakan untuk memandu konstruksi bukti. Akibatnya, teori pembuktian bersifat sintaksis, berbeda dengan teori model yang bersifat semantik.

Aksioma dan Teorema

Dalam logika formal, aksioma adalah sekumpulan kalimat yang diasumsikan benar. Biasanya, aksioma secara default berisi definisi semua operator logika. Misalnya, operator negasi $\neg$ didefinisikan oleh $\neg(T) = \perp$ dan $\neg(\perp) = T$.

Secara intuitif, himpunan teorema adalah rumus-rumus yang dapat dibuktikan kebenarannya dari aksioma oleh sistem pembuktian yang diberikan. Misalnya, untuk variabel proposisional p apa pun, $\neg(\neg(p)) = p$ adalah sebuah teorema. Dengan menggunakan metode analisis kasus, jika nilai kebenaran p adalah $\top$, maka $\neg(\neg(\top)) = \neg(\perp) = \top$; jika nilai kebenaran p adalah $\perp$, maka $\neg(\neg(\perp)) = \neg(\top) = \perp$. Mudah untuk melihat bahwa himpunan aksioma yang berbeda atau sistem pembuktian yang berbeda akan menghasilkan himpunan teorema yang berbeda.

Pendekatan yang lebih baik untuk mendefinisikan teorema adalah dengan menggunakan konsep model. Diberikan himpunan aksioma, teorema adalah rumus apa pun yang akan menerima model aksioma apa pun sebagai modelnya. Menurut definisi ini, jika himpunan aksioma tidak memiliki model, maka setiap rumus akan menjadi teorema. Nantinya, kita akan melihat bahwa definisi teorema ini mengarah pada dua sifat penting dari suatu sistem pembuktian: kelengkapan (setiap teorema dapat dibuktikan) dan kebenaran (segala sesuatu yang dibuktikan adalah teorema).

Terdapat dua sifat penting yang berkaitan dengan himpunan aksioma:

- **Konsistensi** Himpunan aksioma dikatakan konsisten jika setiap rumus dalam himpunan aksioma dapat benar pada saat yang bersamaan.
- **Independensi** Himpunan aksioma dikatakan independen jika tidak ada aksioma yang merupakan teorema dari aksioma lainnya. Himpunan aksioma yang independen juga disebut himpunan aksioma minimum.

Prosedur Pembuktian

Diberikan himpunan aksioma A dan rumus B , kita membutuhkan prosedur P yang dapat menjawab pertanyaan “apakah B merupakan teorema dari A atau bukan”: Jika $P(A, B)$ mengembalikan “ya,” kita katakan B terbukti; jika $P(A, B)$ mengembalikan “tidak,” kita katakan B tidak terbukti; jika tidak mengembalikan apa pun, kita tidak tahu apakah B merupakan teorema atau bukan. P disebut prosedur pembuktian. Prosedur ini dapat dilakukan secara manual atau dieksekusi pada komputer.

Ada empat sifat penting yang berkaitan dengan prosedur pembuktian P :

- **Konsistensi** $P(A, B)$ konsisten jika dan hanya jika A tidak konsisten, atau tidak mungkin bagi $P(A, B)$ untuk mengembalikan nilai benar dan salah untuk B apa pun. Konsep ini mengasumsikan bahwa prosedur pembuktian dapat mengembalikan lebih dari satu keluaran pada masukan yang sama.
- **Kebenaran** $P(A, B)$ benar jika dan hanya jika setiap kali $P(A, B)$ mengembalikan nilai benar, B merupakan teorema dari A ; Setiap kali $P(A, B)$ mengembalikan nilai false, maka B bukanlah teorema dari A .
- **Kelengkapan** $P(A, B)$ lengkap jika dan hanya jika untuk setiap teorema B dari A , $P(A, B)$ mengembalikan nilai true.
- **Terminasi** $P(A, B)$ terminasi jika dan hanya jika $P(A, B)$ berhenti untuk setiap himpunan aksioma A dan setiap rumus B .

Kesadaran adalah sifat yang lebih kuat daripada konsistensi karena suatu prosedur pembuktian mungkin konsisten tetapi tidak sah. Misalnya, dalam logika proposisional, jika $P(\{p \vee q\}, p)$ mengembalikan nilai true tetapi $P(\{p \vee q\}, \neg p)$ tidak mengembalikan nilai true, maka P konsisten. Namun, P tidak sah, karena p bukanlah teorema dari $p \vee q$ dalam logika proposisional.

Proposisi 1.3.11

- (a) Jika suatu prosedur pembuktian $P(A, B)$ sah, maka $P(A, B)$ konsisten.
- (b) Jika prosedur pembuktian $P(A, B)$ sah dan berakhir, maka $P(A, B)$ lengkap.

Bukti

- (a) Jika A tidak konsisten, maka $P(A, B)$ konsisten menurut definisi. Sekarang anggap A konsisten. Jika $P(A, B)$ tidak konsisten, maka ada rumus B sedemikian sehingga $P(A, B)$ mengembalikan nilai benar dan salah. Dalam hal ini, $P(A, B)$ tidak dapat sah, karena B tidak dapat menjadi teorema dan bukan teorema pada saat yang sama ketika A konsisten.
- (b) Untuk setiap teorema B dari aksioma A , $P(A, B)$ harus mengembalikan nilai benar karena jika $P(A, B)$ berhenti dengan nilai salah, itu merupakan kontradiksi terhadap kesahihan $P(A, B)$.

Bukti dari proposisi di atas menunjukkan bahwa, dengan syarat kebenaran, penghentian P menyiratkan kelengkapan P . Dalam hal ini, P disebut prosedur keputusan untuk logika jika P baik benar maupun berakhir. Dengan hasil di atas, setiap prosedur keputusan adalah benar dan lengkap. Secara umum, prosedur keputusan biasanya mengacu pada algoritma yang selalu berhenti dengan jawaban yang benar untuk masalah keputusan, seperti " A valid atau tidak." Beberapa algoritma selalu berhenti dengan salah satu hasil: "ya," "tidak," atau "tidak diketahui"; algoritma tersebut tidak dapat disebut "prosedur keputusan," karena algoritma tersebut dapat mengembalikan "tidak diketahui" pada beberapa teorema, sehingga tidak lengkap.

Dengan syarat kebenaran, kelengkapan prosedur pembuktian P tidak menyiratkan penghentian P , karena prosedur pembuktian yang benar dan lengkap dapat berulang pada formula yang bukan teorema dan dengan demikian tidak dapat menjadi prosedur keputusan menurut definisi. Dalam literatur, orang menyebut prosedur pembuktian yang benar dan lengkap sebagai prosedur semi-keputusan untuk logika, karena prosedur tersebut akan berhenti pada formula apa pun yang merupakan teorema.

Untuk logika apa pun, kita selalu mencari prosedur pengambilan keputusan yang sah dan berakhir. Misalnya, logika proposisional memiliki prosedur pengambilan keputusan yang berbeda untuk masalah pengambilan keputusan yang berbeda. Jika prosedur pengambilan keputusan tidak ada, kita mencari prosedur semi-pengambilan keputusan. Untuk beberapa logika, bahkan prosedur semi-pengambilan keputusan pun tidak dapat ada. Komentar ini akan lebih bermakna setelah kita mendefinisikan logika yang dimaksud secara formal.

Aturan Inferensi

Dalam teori pembuktian struktural, yang merupakan bidang utama teori pembuktian, aturan inferensi digunakan untuk membangun bukti. Dalam logika, aturan inferensi adalah pola yang mengambil rumus sebagai premis dan mengembalikan rumus sebagai kesimpulan. Misalnya, aturan modus ponens (MP) mengambil dua premis, satu dalam bentuk " $\text{Jika } p \text{ maka } q$ " (yaitu, $p \rightarrow q$) dan yang lain dalam bentuk p , dan mengembalikan kesimpulan q . Suatu aturan dikatakan sah jika setiap kali premis benar di bawah interpretasi apa pun, maka kesimpulannya juga benar.

Sistem inferensi S terdiri dari himpunan A aksioma dan himpunan R aturan inferensi. Kebenaran S berasal dari kebenaran setiap aturan inferensi dalam R . Bukti formula F_n dalam $S = (A, R)$ adalah urutan formula $F_1, F_2, \dots, F_n$ sedemikian sehingga setiap F_i berada di A atau

dapat dihasilkan oleh aturan inferensi R , menggunakan formula sebelum F_i dalam urutan tersebut sebagai premis.

Contoh 1.3.12 Misalkan $S = (A, R)$, $A = \{p \rightarrow (q \rightarrow r), p, q\}$, dan $R = \{MP\}$. Bukti r dari A menggunakan R diberikan di bawah ini:

1. $p \rightarrow (q \rightarrow r)$	<i>axiom</i>
2. p	<i>axiom</i>
3. $q \rightarrow r$	$MP, 1, 2$
4. q	<i>axiom</i>
5. r	$MP, 3, 4$

di mana “ $MP, 1, 2$ ” berarti rumus $q \rightarrow r$ diturunkan oleh MP (modus ponens) dari rumus 1 dan 2 sebagai premis.

Dengan demikian, bukti formal adalah urutan rumus, dimulai dengan aksioma dan dilanjutkan dengan teorema yang diturunkan dari anggota sebelumnya dalam urutan tersebut melalui aturan inferensi.

Salah satu keuntungan yang jelas dari bukti semacam itu adalah bahwa bukti-bukti ini dapat diperiksa baik secara manual maupun otomatis oleh komputer. Memeriksa bukti formal biasanya sederhana, sedangkan menemukan bukti (pembuktian teorema otomatis) umumnya sulit. Sebaliknya, memeriksa bukti informal dalam literatur matematika membutuhkan waktu berminggu-minggu untuk ditinjau oleh rekan sejawat dan mungkin masih mengandung kesalahan. Bukti informal dalam praktik matematika sehari-hari tidak seperti bukti dalam teori pembuktian. Bukti informal seperti sketsa tingkat tinggi yang memungkinkan seorang ahli untuk merekonstruksi bukti formal setidaknya secara prinsip, jika diberi cukup waktu dan kesabaran. Bagi sebagian besar matematikawan, menulis bukti formal sepenuhnya terlalu bertele-tele dan panjang lebar untuk digunakan secara umum.

Dalam teori pembuktian, bukti biasanya disajikan sebagai struktur data yang didefinisikan secara rekursif seperti daftar (seperti yang ditunjukkan di sini), atau pohon, yang dibangun sesuai dengan aksioma dan aturan inferensi dari sistem logika. Dengan demikian, teori pembuktian bersifat sintaksis, berbeda dengan teori model yang bersifat semantik.

Setiap sistem inferensi dapat diubah menjadi prosedur pembuktian jika strategi yang adil diterapkan untuk mengontrol penggunaan aturan inferensi. Strategi yang adil memastikan bahwa jika suatu aturan inferensi dapat diterapkan pada suatu titik untuk menghasilkan rumus baru, rumus baru ini pada akhirnya akan diturunkan. Jika setiap aturan inferensi valid, maka prosedur pembuktian yang sesuai juga valid. Upaya ekstra umumnya diperlukan untuk menunjukkan bahwa prosedur pembuktian tersebut lengkap atau berhenti.

Selain teori pembuktian struktural, beberapa bidang utama teori pembuktian meliputi analisis ordinal, logika pembuktian, matematika terbalik, penambangan bukti, pembuktian teorema otomatis, dan kompleksitas bukti.

Salah satu tujuan utama buku ini adalah untuk menyajikan algoritma yang diciptakan untuk logika dan perangkat lunak yang dibangun berdasarkan algoritma tersebut. Itulah

mengapa kita akan menginvestasikan upaya yang tidak proporsional (menurut standar buku teks logika konvensional) dalam aspek algoritmik logika. Teori himpunan dan teori model membantu kita memahami dan menentukan masalah; teori pembuktian membantu kita membuat algoritma; dan teori komputabilitas membantu kita mengetahui masalah mana yang memiliki algoritma, dan masalah mana yang sama sekali tidak memiliki algoritma.

1.6 LATIHAN SOAL

1. Tentukan apakah kalimat-kalimat berikut merupakan pernyataan atau bukan. Jika ya, gunakan variabel proporsional (yaitu, simbol) untuk pernyataan sederhana dan penghubung logika untuk mengungkapkannya.
 - (a) Jika saya memenangkan lotre, saya akan miskin.
 - (b) Pria itu bertanya, "tutup pintu di belakangmu."
 - (c) Hari ini adalah hari pertama sekolah.
 - (d) Hari ini cerah atau berawan.
2. Pinocchio, boneka animasi, dihukum untuk setiap kebohongan yang dia ucapkan dengan mengalami pertumbuhan hidung lebih lanjut. Paradoks Pinocchio muncul ketika Pinocchio berkata "Hidungku sekarang tumbuh." Jelaskan mengapa ini merupakan paradoks.
3. Paradoks Yablo diberikan sebagai urutan pernyataan tak terbatas yang terurut, yang masing-masing menyatakan bahwa "pernyataan berikutnya salah." Jelaskan mengapa ini merupakan paradoks. Tentukan apakah paradoks ini bergantung pada referensi diri atau tidak.
4. Jawablah pertanyaan-pertanyaan dalam contoh-contoh berikut mengenai kesalahan logika.
 - (a) Selama pertengkaran mereka, Anthony mengatakan kepada Marie bahwa dia membencinya dan berharap dia tertabrak mobil. Kemudian malam itu, Anthony menerima telepon dari seorang teman yang memberitahunya bahwa Marie berada di rumah sakit karena tertabrak mobil. Anthony segera menyalahkan dirinya sendiri dan beralasan bahwa jika dia tidak membuat komentar itu selama pertengkaran mereka, Marie tidak akan tertabrak. Kesalahan logika apa yang telah dilakukan Anthony?
 - (b) Louise mencalonkan diri sebagai ketua kelas. Dalam pidato kampanyenya, dia mengatakan, "Lawan saya tidak pantas menang. Dia seorang perokok dan dia selingkuh dari pacarnya tahun lalu." Kesalahan logika apa yang telah dilakukan Louise?
 - (c) Bill: "Saya pikir hukuman mati itu salah." Adam: "Tidak, bukan begitu. Jika itu salah, itu tidak akan legal." Kesalahan logika apa yang menjadi contoh dari ini?
 - (d) Ricky sedang menonton televisi ketika dia melihat iklan krim kaki. Penyiar iklan berkata, "Ini adalah krim kaki terbaik di pasaran karena tidak ada pembuat krim kaki lain yang mampu membuktikan sebaliknya!" Kesalahan logika apa yang telah dilakukan penyiar?

- (e) Maria telah bekerja di pekerjaannya saat ini selama lebih dari 30 tahun dengan upah yang sama. Dia sangat menginginkan kenaikan gaji, jadi dia mendekati bosnya untuk memintanya. Dia berkata, "Anda adalah salah satu orang paling baik yang saya kenal. Anda pintar dan tampan, dan saya sangat menyukai sepatu Anda." Jenis kesalahan logika apa ini?
 - (f) Ibu Jeff khawatir ketika dia mengetahui bahwa dia bolos kelas suatu hari. Dia mengatakan kepadanya bahwa dia khawatir karena dia bolos satu kelas, dia akan mulai bolos lebih sering. Kemudian dia akan putus sekolah sama sekali, tidak pernah lulus atau masuk perguruan tinggi, dan akhirnya menganggur dan tinggal di rumah selama sisa hidupnya. Jenis kekeliruan apa yang telah dilakukan ibu Jeff?
 - (g) Dana sedang berusaha mengumpulkan uang untuk perpustakaan universitasnya. Dalam pidatonya kepada dewan pengawas, dia berkata, "Kita harus menaikkan biaya kuliah untuk menutupi biaya buku-buku baru. Jika tidak, perpustakaan akan terpaksa ditutup." Ini merupakan contoh dari kekeliruan apa?
 - (h) Jeff sedang bersiap untuk membuat iklan untuk minuman energi baru. Dia mengunjungi sekolah menengah setempat dan mensurvei siswa di kelas bahasa Inggris tentang preferensi minuman mereka. Sebagian besar siswa mengatakan mereka lebih menyukai minuman rasa anggur, jadi Jeff memberi tahu atasannya bahwa anggur adalah rasa yang paling disukai oleh siswa sekolah menengah. Kesalahan penalaran apa yang telah dilakukan Jeff?
5. Dalam tiga soal terakhir dari latihan sebelumnya, harap identifikasi kalimat mana yang merupakan pernyataan dan mana yang bukan. Untuk pernyataan sederhana dalam soal-soal ini, harap perkenalkan variabel proposisional (yaitu, simbol) untuk menunjuknya. Gunakan variabel dan operator logika ini untuk menunjuk pernyataan yang tersusun.
 6. Jika adalah urutan parsial, tunjukkan bahwa kebalikan dari adalah urutan parsial.
 7. Misalkan $A = \{0, 1\}$; berikan solusi $P(A)$ dan $P(P(A))$.
 8. Tunjukkan bahwa semua himpunan tak hingga terhitung memiliki ukuran yang sama, dan setiap himpunan tak hingga terhitung memiliki fungsi rank.
 9. Tunjukkan bahwa jika r adalah fungsi rank dari himpunan tak hingga terhitung S , maka r menginduksi $\succsim$ urutan baik pada S : $x \succsim y$ jika dan hanya jika $r(x) \geq r(y)$.
 10. Misalkan a, b adalah bilangan real dan $a < b$. Definisikan fungsi $f: [0, 1] \rightarrow [a, b]$ dengan $f(x) = a + x(b - a)$, di mana $[a, b]$ menunjukkan interval $\{x \mid a \leq x < b\}$. Tunjukkan bahwa f adalah bijektif.
 11. Jawablah pertanyaan-pertanyaan berikut dengan justifikasi:
 - (a) Apakah himpunan semua bilangan bulat ganjil terhitung?
 - (b) Apakah himpunan bilangan real dalam interval $[0, 0,1]$ terhitung?
 - (c) Apakah himpunan sudut dalam interval $[0^\circ, 90^\circ]$ terhitung?
 - (d) Apakah himpunan semua titik pada bidang dengan koordinat rasional terhitung?
 - (e) Apakah himpunan semua program Java terhitung?
 - (f) Apakah himpunan semua kata yang menggunakan alfabet Inggris terhitung?

- (g) Apakah himpunan pasir di bumi terhitung?
- (h) Apakah himpunan atom di tata surya terhitung?
- 12. Buktikan sifat-sifat berikut:
 - (a) Fungsi $f: A \rightarrow B$ bersifat injektif jika dan hanya jika inversnya terdefinisi dengan baik dan injektif.
 - (b) Fungsi injektif $f: A \rightarrow B$ bersifat total jika dan hanya jika inversnya bersifat surjektif (dengan asumsi fungsi surjektif tidak diharuskan total).
- 13. Buktikan Proposisi 1.3.1: $f: A \rightarrow B$ adalah bijektif jika dan hanya jika f dan f^{-1} keduanya adalah fungsi total.
- 14. Buktikan bahwa himpunan R bilangan rasional adalah terhitung.
- 15. Buktikan bahwa himpunan $N \times \{a, b, c\}$ adalah terhitung, di mana N adalah himpunan bilangan asli. Buktikan Anda tidak boleh menggunakan sifat bahwa himpunan terhitung tertutup terhadap hasil perkalian Kartesius.
- 16. Tentukan dengan bukti apakah himpunan N^k terhitung atau tidak, di mana N adalah himpunan bilangan asli dan $k \in N$.
- 17. Tentukan dengan bukti apakah himpunan $S_k = \{A \mid A \subset N, |A| = k\}$ terhitung atau tidak, di mana N adalah himpunan bilangan asli dan $k \in N$.
- 18. Buktikan bahwa jika A dan B keduanya terhitung dan $A \cap B = \emptyset$, maka $A \cup B$ adalah terhitung.
- 19. Misalkan masing-masing $A_0, A_1, \dots, A_i, \dots$ adalah tak hingga terhitung dan untuk semua $i, j \in N, A_i \cap A_j = \emptyset$ untuk $i \neq j$. Buktikan bahwa $S = \bigcup_{i=0}^{\infty} A_i$ adalah tak hingga terhitung.
- 20. Tentukan dengan bukti apakah himpunan semua fungsi total $f: \{0, 1\} \rightarrow N$ terhitung atau tidak.
- 21. Berikan sebuah bijeksi $f: N \rightarrow \{0, 1\}^*$ dan tunjukkan cara menghitung $f(x)$ untuk setiap $x \in N$. Berapakah kompleksitas waktu algoritma Anda untuk $f(x)$?
- 22. Untuk fungsi γ dalam Definisi 1.3.5, definisikan γ^{-1} secara eksplisit sebagai fungsi total dari N ke Σ^* .
- 23. Buktikan bahwa himpunan $\{a, b, c\}^*$ dari semua string (dengan panjang terbatas) pada $\{a, b, c\}$ adalah terhitung.
- 24. Buktikan bahwa himpunan semua bahasa formal tidak terhitung, di mana bahasa formal adalah himpunan bagian dari Σ^* dan $\Sigma \neq \emptyset$.
- 25. Buktikan bahwa pernyataan-pernyataan berikut ini secara logis ekuivalen:
 - (a) Setiap himpunan bagian dari himpunan terhitung adalah terhitung.
 - (b) Setiap himpunan bagian dari N adalah terhitung.
 - (c) Jika terdapat fungsi injektif dari himpunan S ke N , maka S adalah terhitung.
 - (d) Jika terdapat fungsi surjektif dari N ke S , maka S adalah terhitung.
- 26. **Paradoks tukang cukur:** Di sebuah desa, hanya ada satu tukang cukur yang mencukur semua orang, dan hanya mereka, yang tidak mencukur diri sendiri. Pertanyaannya adalah, apakah tukang cukur itu mencukur dirinya sendiri? Misalkan relasi $s(x, y)$ adalah “ x mencukur y ” dan b menunjukkan tukang cukur.

- (a) Definisikan A sebagai himpunan orang-orang yang dicukur oleh tukang cukur menggunakan $s(x, y)$ dan b .
 - (b) Definisikan B sebagai himpunan orang-orang yang tidak mencukur dirinya sendiri.
 - (c) Gunakan A dan B untuk menunjukkan bahwa jawaban atas pertanyaan “apakah tukang cukur mencukur dirinya sendiri?” mengarah pada kontradiksi.
27. Misalkan himpunan bilangan asli dibangun oleh konstanta 0 dan fungsi penerus s . Berikan definisi $<$ dan $\leq$ pada bilangan asli. Tidak ada fungsi selain 0 dan s yang diizinkan untuk digunakan saat mendefinisikan $<$ atau $\leq$.
28. Berikan definisi fungsi pangkat, $pow(x, y)$, yang menghitung xy , pada bilangan asli yang dibangun oleh 0 dan fungsi penerus s . Anda hanya diizinkan untuk menggunakan fungsi $add(penjumlahan)$ dan $mul(perkalian)$.
29. Berikan BNF (Bentuk Backus–Naur) untuk himpunan pohon biner, di mana setiap simpul yang tidak kosong berisi bilangan asli. Anda dapat menggunakan $null: Tree$ untuk pohon kosong dan $node: Nat, Tree, Tree' \rightarrow Tree$ untuk simpul pohon.

BAB 2

LOGIKA PROPOSISIONAL

Mengikuti teori model yang disajikan pada Bab 1, kita akan menyajikan logika proposisional sebagai sebuah bahasa. Pertama-tama kita akan memperkenalkan sintaksis bahasa ini dan kemudian semantiknya.

2.1 SINTAKSIS

Sintaksis logika proposisional menentukan simbol apa yang diperbolehkan dan dalam bentuk apa simbol-simbol ini digunakan. Logika proposisional menggunakan variabel proposisional dan operator logika sebagai satu-satunya simbol. Pada bab pertama, kita telah mengatakan bahwa variabel proposisional adalah simbol untuk menyatakan pernyataan dan hanya mengambil nilai benar dan salah. Untuk kenyamanan, kita akan menggunakan 1 untuk benar dan 0 untuk salah. Kita akan memperlakukan $\top$ dan $\perp$ sebagai dua operator logika nullary sedemikian rupa sehingga nilai benar $\top$ adalah 1 dan nilai benar $\perp$ adalah 0. Perhatikan bahwa $\top$ dan $\perp$ adalah simbol sintaksis sedangkan 1 dan 0 adalah simbol semantik.

Operator Logika

Para matematikawan menggunakan kata-kata “*Not (tidak)*”, “*And (dan)*”, “*atau*” dan sebagainya, untuk operator yang mengubah atau menggabungkan proposisi. Arti dari operator logika ini dapat ditentukan sebagai fungsi yang menerima nilai Boolean dan mengembalikan nilai Boolean. Fungsi-fungsi ini disebut fungsi Boolean. Misalnya, jika p adalah sebuah proposisi, maka $\neg p$ juga merupakan proposisi dan nilai kebenaran dari proposisi $\neg p$ ditentukan oleh nilai kebenaran p sesuai dengan arti dari $\neg$: $\neg(1) = 0$ dan $\neg(0) = 1$. Artinya, jika nilai p adalah 1, maka nilai $\neg(p)$ adalah 0; jika nilai p adalah 0, maka nilai $\neg(p)$ adalah 1. Perhatikan bahwa simbol $\neg$ yang sama digunakan baik sebagai operator Boolean (sisi sintaksis) maupun sebagai fungsi Boolean (sisi semantik).

Tabel 2.1 Arti dari $\wedge$, $\vee$, $\rightarrow$, $\oplus$, dan $\leftrightarrow$

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$q \rightarrow p$	$(q \rightarrow p) \wedge (p \rightarrow q)$	$p \oplus q$	$p \leftrightarrow q$
1	1	1	1	1	1	1	0	1
1	0	0	1	0	1	0	1	0
0	1	0	1	1	0	0	1	0
0	0	0	0	1	1	1	1	1

Secara umum, definisi fungsi Boolean ditampilkan oleh sebuah tabel, yang disebut tabel kebenaran, di mana keluaran fungsi Boolean diberikan untuk setiap kemungkinan penugasan nilai kebenaran ke semua variabel masukan.

Sebagai contoh, arti dari $\wedge$, $\vee$, dan $\rightarrow$ didefinisikan oleh Tabel 2.1. Tabel tersebut memiliki empat baris, karena ada empat pasang nilai Boolean untuk kedua variabel. Menurut tabel tersebut, nilai kebenaran dari $p \wedge q$ adalah 1 ketika p dan q keduanya bernilai 1. Nilai

kebenaran dari $p \vee q$ adalah 1 ketika p atau q , atau keduanya bernilai benar. Ini tidak selalu merupakan arti yang dimaksudkan dari "atau" dalam dialog sehari-hari, tetapi ini adalah definisi standar dalam logika. Jadi, jika seorang ahli logika mengatakan, "Anda boleh makan kue, atau Anda boleh makan es krim," ia bermaksud bahwa Anda dapat makan keduanya.

Jika Anda ingin mengecualikan kemungkinan makan kue dan es krim sekaligus, Anda harus menggabungkannya dengan operator eksklusif-atau, $\oplus$, yang juga didefinisikan dalam Tabel 2.1. Disjungsi eksklusif $\oplus$ sesuai dengan penjumlahan modulo 2 dan oleh karena itu diberi simbol $+$, di mana nilainya adalah 0 dan 1. Tabel kebenaran untuk implikasi juga diberikan pada Tabel 2.1. Operator ini layak diingat, karena sebagian besar pernyataan matematika berbentuk "jika-maka". Sekarang mari kita cari tahu kebenaran dari contoh berikut: Jika gajah terbang, saya akan berada di Mars. Berapakah nilai kebenaran dari pernyataan ini? Pernyataan ini dapat direpresentasikan oleh $p \rightarrow q$, di mana p adalah "gajah dapat terbang" dan q adalah "saya akan berada di Mars."

Gajah tidak terbang, jadi nilai kebenaran p adalah 0 dan kita sampai pada dua baris terakhir Tabel 2.1: Di kedua baris tersebut, $p \rightarrow q$ adalah 1. Pada Bagian selanjutnya tentang kekeliruan informal, kita telah menunjukkan bahwa dengan premis yang salah, Anda dapat membuat kesimpulan apa pun, sehingga pernyataan gabungan selalu benar. Namun, pernyataan yang selalu benar tidak memiliki nilai dalam penalaran, dan tidak menyiratkan hubungan sebab-akibat antara premis dan kesimpulan dari suatu implikasi. Dalam logika, penting untuk menerima fakta bahwa implikasi logis mengabaikan hubungan sebab-akibat.

Dalam Tabel 2.1, kami juga memberikan definisi $\leftrightarrow$ (jika dan hanya jika). Jika p menyatakan "Tom mengenakan celana jeans biru" dan q menyatakan "Sam mengenakan celana jeans biru," maka rumus $p \leftrightarrow q$ menyatakan bahwa "Tom dan Sam selalu mengenakan celana jeans biru pada waktu yang sama." Ini karena kalimat tersebut menyiratkan bahwa "Tom mengenakan celana jeans biru ketika Sam mengenyakannya" dan "Sam mengenakan celana jeans biru ketika Tom mengenyakannya." Jadi, kita memiliki $(p \rightarrow q) \wedge (q \rightarrow p)$, yang memiliki nilai kebenaran yang sama dengan $p \leftrightarrow q$ dalam setiap interpretasi. Bahkan, $p \leftrightarrow q$ benar jika dan hanya jika p dan q memiliki nilai kebenaran yang sama. Perhatikan bahwa $p \oplus q$ dan $p \leftrightarrow q$ selalu memiliki nilai yang berlawanan, yaitu, $\neg(p \oplus q)$ dan $p \leftrightarrow q$ selalu memiliki nilai yang sama.

Misalkan $B = \{0, 1\}$; arti dari setiap operator logika adalah fungsi Boolean $f: B^n \rightarrow B$. Ada dua operator nullary (ketika $n = 0$): $\top$ menunjukkan 1 dan $\perp$ menunjukkan 0. Ada empat operator unary (ketika $n = 1$): $f_1(x) = 0$; $f_2(x) = 1$; $f_3(x) = x$; dan $f_4(x) = \neg x$. Ada 16 operator biner (ketika $n = 2$), dan $\vee$, $\wedge$, $\rightarrow$, $\oplus$, dan $\leftrightarrow$ adalah beberapa di antaranya. Kita dapat menggunakan operator ketika $n > 2$. Misalnya, misalkan $ite: B^3 \rightarrow B$ adalah operator *if – then – else*, jika kita menggunakan tabel kebenaran untuk arti $ite(p, q, r)$, maka tabel kebenaran akan memiliki delapan baris. Secara umum, ada 2^{2^n} fungsi Boolean n -ary, karena jumlah fungsi $f: A \rightarrow B$ adalah $|B|^{|A|}$ dan untuk fungsi Boolean n -ary, $|B| = 2$ dan $|A| = |B^n| = |B|^n = 2^n$.

Rumus

Tidak setiap kombinasi variabel proposisional dan operator logika masuk akal. Misalnya, $p () q \vee \neg$ adalah rangkaian simbol yang tidak bermakna. Rumus proposisional adalah himpunan rangkaian yang terbentuk dengan baik yang dibangun oleh variabel proposisional dan operator logika berdasarkan seperangkat aturan yang ketat.

Misalkan op adalah operator biner dan V adalah variabel proposisional yang digunakan dalam suatu aplikasi, maka Rumus untuk aplikasi ini didefinisikan oleh tata bahasa BNF berikut:

$$\begin{aligned}\langle op \rangle &::= \wedge \mid \vee \mid \rightarrow \mid \oplus \mid \leftrightarrow \\ \langle V \rangle &::= p \mid q \mid r \mid s \mid t \\ \langle Rumus \rangle &::= \top \mid \perp \mid \langle V \rangle \mid \neg \langle Rumus \rangle \mid (\langle Rumus \rangle \langle op \rangle \langle Rumus \rangle)\end{aligned}$$

Kita dapat menambah/menghapus operator atau variabel proposisional sesuka hati dalam definisi di atas. Menurut definisi tersebut, setiap anggota Formula adalah konstanta $\top$ atau $\perp$, variabel proposisional VP , atau negasi dari sebuah formula, atau operator biner yang berlaku untuk dua formula. Berikut beberapa contohnya:

$$\top, p, \neg q, (p \wedge \neg q), (r \rightarrow (p \wedge \neg q)), ((p \wedge \neg q) \vee r)$$

Sepanjang bab ini, kita akan menggunakan $p, q, \dots$, untuk variabel proposisional, dan $A, B, \dots$, untuk rumus.

Untuk setiap operator biner, kita memperkenalkan sepasang tanda kurung, dan beberapa di antaranya mungkin tidak diperlukan. Kita akan menggunakan prioritas, yang merupakan urutan total atas semua operator, untuk menghapusnya. Prioritas tipikal diberikan sebagai daftar berikut, di mana operator dengan prioritas lebih tinggi didahulukan. Perhatikan bahwa kita menetapkan prioritas yang sama untuk $\oplus$ dan $\leftrightarrow$.

$$\neg, \wedge, \vee, \rightarrow \{ \oplus, \leftrightarrow \}$$

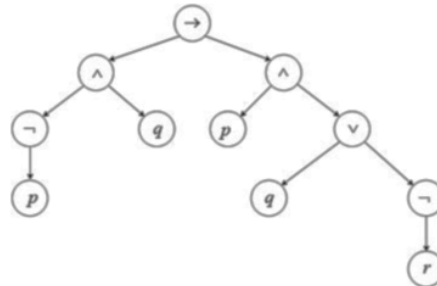
Dengan demikian $(r \rightarrow (p \wedge \neg q))$ dapat dituliskan sebagai $r \rightarrow p \wedge \neg q$, dan kita tidak akan menganggapnya sebagai $(r \rightarrow p) \wedge \neg q$, karena $\wedge$ memiliki prioritas lebih tinggi daripada $\rightarrow$. Demikian pula, $((p \wedge \neg q) \vee r)$ dituliskan sebagai $p \wedge \neg q \vee r$ tanpa ambiguitas.

Ketika ekspresi yang terbentuk dengan baik, seperti rumus yang didefinisikan di atas, direpresentasikan oleh sebuah string (urutan simbol), kita menggunakan tanda kurung dan koma untuk mengidentifikasi struktur ekspresi tersebut. Struktur seperti itu dapat dengan mudah direpresentasikan oleh sebuah pohon, di mana kita tidak memerlukan tanda kurung dan koma.

Definisi 2.1.1 Sebuah pohon untuk rumus A didefinisikan secara rekursif sebagai berikut:

1. Jika A adalah p , $\top$, atau $\perp$, pohon tersebut adalah sebuah simpul tunggal dengan A sebagai labelnya.

2. Jika A adalah $\neg B$, maka simpul akar diberi label $\neg$ dan memiliki satu cabang yang menunjuk ke pohon rumus B .
3. Jika A adalah $B \text{ op } C$, maka simpul akar diberi label op dan memiliki dua cabang yang menunjuk ke pohon rumus B dan C , masing-masing.



Gambar 2.1 Pohon rumus untuk $(\neg p \wedge q) \rightarrow (p \wedge (q \vee \neg r))$

Sebagai contoh, pohon rumus $(\neg p \wedge q) \rightarrow (p \wedge (q \vee \neg r))$ ditampilkan pada Gambar 2.1. Representasi pohon dari suatu rumus tidak memerlukan tanda kurung.

Suatu rumus dan pohon rumusnya adalah dua representasi dari objek yang sama. Kita bebas memilih salah satu representasi untuk kemudahan pembahasan.

Definisi 2.1.2 Diberikan suatu rumus A , kita katakan B adalah subrumus A jika $B = A$ atau $A = (A_1 \text{ op } A_2)$, di mana op adalah operator biner, atau $A = \neg A_1$ dan B adalah subrumus dari A_1 atau A_2 . Jika B adalah subformula dari A dan $B \neq A$, maka B adalah subformula yang tepat dari A .

Sebagai contoh, $\neg p \wedge \neg(q \vee r)$ mengandung $p, q, r, \neg p, q \vee r$, dan $\neg(q \vee r)$ sebagai subformula yang tepat. Secara intuitif, subformula hanyalah sebuah formula yang diwakili oleh subpohon apa pun dalam pohon formula.

Definisi 2.1.3 Diberikan subformula B dari A dan formula lain C , kita menggunakan $A[B \leftarrow C]$ untuk menunjukkan formula yang diperoleh dengan mengganti semua kemunculan B dengan C dalam A . Jika B adalah variabel proposisional, maka $A[B \leftarrow C]$ disebut sebuah contoh dari A .

Sebagai contoh, jika A adalah $\neg p \vee \neg(q \vee r)$, B adalah $\neg(q \vee r)$, dan C adalah $\neg q \wedge \neg r$, maka $A[B \leftarrow C]$ adalah $\neg p \vee (\neg q \wedge \neg r)$. $A[p \leftarrow p \wedge q] = \neg(p \wedge q) \vee \neg(q \vee r)$ adalah sebuah contoh dari A .

2.2 SEMANTIK

Judul bagian sebelumnya adalah sintaksis, meskipun kita telah membahas nilai Boolean, tabel kebenaran, dan makna operator logika, yang sebenarnya merupakan konsep semantik dari logika. Sekarang, mari kita fokus pada makna variabel proposisional dan rumus.

Makna informal dari variabel proposisional adalah pernyataan yang diwakili oleh simbol tersebut. Dalam aplikasi yang berbeda, simbol yang sama dapat mewakili pernyataan yang berbeda dan dengan demikian memiliki makna yang berbeda. Namun, setiap pernyataan hanya dapat memiliki dua nilai kebenaran, yaitu benar atau salah. Kita akan menggunakan nilai kebenaran variabel proposisional ini, yang diperluas ke rumus yang mengandungnya, sebagai semantik formal dari logika proposisional.

Interpretasi

Suatu variabel proposisional dapat diberi nilai kebenaran, baik benar atau salah. Penetapan nilai kebenaran ini dianggap sebagai semantik variabel tersebut. Untuk suatu rumus, penetapan nilai kebenaran untuk setiap variabel proposisional dikatakan sebagai interpretasi dari rumus tersebut. Jika A adalah rumus yang dibangun di atas himpunan V_P variabel proposisional, maka interpretasi dari A adalah fungsi $\sigma : V_P \rightarrow \{1, 0\}$. Mudah untuk memeriksa bahwa ada $2^{|V_P|}$ interpretasi yang berbeda.

Misalkan σ adalah interpretasi atas $V_P = \{p, q\}$ sedemikian sehingga $\sigma(p) = 1$ dan $\sigma(q) = 0$. Kita dapat menulis $\sigma = \{p \rightarrow 1, q \rightarrow 0\}$. Kita juga dapat menggunakan $(1, 0)$ untuk σ yang sama, dengan asumsi V_P adalah daftar (p, q) dan $\sigma(p, q) = (1, 0)$.

Representasi alternatif dari interpretasi σ adalah dengan menggunakan himpunan bagian dari V_P . Diberikan σ , misalkan

$$X_\sigma = \{x \in V_P \mid \sigma(x) = 1\}$$

Maka terdapat hubungan satu-ke-satu antara σ dan X_σ . Jadi, kita dapat menggunakan X_σ untuk mewakili σ . Misalnya, jika $V_P = \{p, q\}$, maka keempat interpretasi atas V_P adalah himpunan kuasa dari V_P :

$$P(V_P) = \{\emptyset, \{p\}, \{q\}, \{p, q\}\}$$

Representasi alternatif lain dari σ adalah dengan menggunakan himpunan literal. Sebuah literal dapat berupa variabel proposisional atau negasi dari variabel proposisional. Misalkan

$$Y_\sigma = \{x \in V_P \mid \sigma(x) = 1\} \cup \{\neg y \mid y \in V_P, \sigma(y) = 0\}$$

Maka Y_σ adalah himpunan literal sedemikian rupa sehingga setiap variabel V_P muncul di Y_σ tepat sekali. Misalnya, misalkan $V_P = \{p, q\}$ dan $X_\sigma = \emptyset$, maka $Y_\sigma = \{\neg p, \neg q\}$; jika $X_\sigma = \{p\}$, $Y_\sigma = \{p, \neg q\}$. Artinya, dengan menambahkan negasi dari variabel yang hilang di X_σ , kita memperoleh representasi Y_σ untuk σ . Menggunakan himpunan literal untuk merepresentasikan interpretasi memiliki keuntungan tambahan: Ia dapat merepresentasikan interpretasi parsial di mana beberapa variabel proposisional tidak memiliki nilai kebenaran.

Untuk representasi interpretasi yang ringkas, kita dapat menulis himpunan literal sebagai konjungsi dari literal-literal ini, seringkali dengan menghilangkan operator konjungsi $\wedge$. Misalnya, $\{p, \neg q\}$ dapat ditulis sebagai $p \wedge \neg q$, atau hanya $p\bar{q}$.

Diberikan σ , kita dapat memeriksa nilai kebenaran rumus di bawah σ , yang dilambangkan dengan $\sigma(A)$. Notasi ini berarti bahwa kita memperlakukan σ sebagai fungsi dari rumus ke $\{0, 1\}$, sebagai perluasan unik dari $\sigma: V_p \rightarrow \{0, 1\}$. Di sisi lain, notasi A_σ digunakan untuk menunjukkan rumus yang diperoleh dengan mengganti setiap variabel proposisional p dengan $\sigma(p)$ jika kita menganggap 1 sebagai T dan 0 sebagai $\perp$.

Contoh 2.2.1 Diberikan rumus $A = (p \wedge \neg q) \vee r$, dan interpretasi $\sigma = \{p \rightarrow 1, q \rightarrow 1, r \rightarrow 0\}$, kita dapat menulis ulang σ sebagai $\{p, q, \neg r\}$. Dengan mengganti p dengan 1, q dengan 1, dan r dengan 0 dalam A , kita memiliki $A_\sigma = 1 \wedge \neg 1 \vee 0$. Dengan menerapkan arti dari $\neg$, $\wedge$, dan $\vee$, kita memperoleh 0. Dalam hal ini, kita mengatakan bahwa rumus tersebut dievaluasi menjadi 0 di bawah σ , yang dilambangkan dengan $\sigma(A) = 0$. Dengan interpretasi lain $\sigma' = \{p, q, r\}$, rumus yang sama akan dievaluasi menjadi 1, yaitu, $\sigma'(A) = 1$.

Ingatlah bahwa kita menggunakan $A[B \leftarrow C]$ untuk menunjukkan contoh A di mana setiap kemunculan variabel proposisional B digantikan oleh C . Interpretasi σ adalah kasus khusus dari substitusi rumus di mana B adalah variabel dan C adalah 1 atau 0. Misalnya, jika $\sigma = \{p, \neg q\}$, maka $A_\sigma = A[p \leftarrow 1][q \leftarrow 0]$. Sebenarnya, A_σ bukanlah formula proposisional: Ini adalah makna dari A di bawah σ . Oleh karena itu, kita mengasumsikan $T\sigma = 1$ dan $\perp\sigma = 0$ untuk setiap σ , di mana T dan $\perp$ adalah formula proposisional, dan 1 dan 0 adalah nilai kebenarannya masing-masing.

Untuk mendapatkan $\sigma(A)$, kita dapat menggunakan algoritma eval.

Algoritma 2.2.2 Algoritma eval akan mengambil formula A dan interpretasi σ dan mengembalikan nilai Boolean.

```

proc eval( $A, \sigma$ )
  if  $A = T$  return 1
  if  $A = \perp$  return 0
  if  $A \in V_p$  return  $\sigma(A)$ 
  if  $A$  is  $\neg B$  return  $\neg$ eval( $B, \sigma$ )
  else  $A$  is  $(B \text{ op } C)$  return eval( $B, \sigma$ ) op eval( $C, \sigma$ )

```

Contoh 2.2.3 Misalkan $\sigma = \{p, \neg q\}$. Maka eksekusi eval($p \rightarrow p \wedge \neg q, \sigma$) akan mengembalikan 1:

```

eval( $p \rightarrow p \wedge \neg q, \sigma$ ) calls
  eval( $p, \sigma$ ), which returns 1; and
  eval( $p \wedge \neg q, \sigma$ ), which calls
    eval( $p, \sigma$ ), which returns 1; and
    eval( $\neg q, \sigma$ ), which calls
      eval( $q, \sigma$ ), which returns 0; and

```

returns $\neg 0$, i.e., 1;
 returns $1 \wedge 1$, i.e., 1;
 returns $1 \rightarrow 1$, i.e., 1.

Yang dilakukan *eval* adalah menelusuri pohon formula A dari bawah ke atas: Jika simpul diberi label dengan variabel, gunakan σ untuk mendapatkan nilai kebenaran; jika tidak, hitung nilai kebenaran simpul ini di bawah σ menggunakan operator pada simpul tersebut dengan nilai kebenaran dari anak-anaknya. Proses menjalankan eval persis sama dengan yang kita lakukan saat membuat tabel kebenaran untuk $p \rightarrow p \wedge \neg q$. Nilai kebenaran di bawah p dan q dalam tabel kebenaran memberi kita semua interpretasi σ dan nilai kebenaran dari $A = \neg q, p \wedge \neg q$, atau $p \rightarrow p \wedge \neg q$ adalah nilai dari $eval(A, \sigma)$.

p	q	$\neg q$	$p \wedge \neg q$	$p \rightarrow p \wedge \neg q$
0	0	1	0	1
0	1	0	0	1
1	0	1	1	1
1	1	0	0	1

Teorema 2.2.4 Algoritma $eval(A, \sigma)$ mengembalikan 1 jika dan hanya jika $\sigma(A) = 1$, dan berjalan dalam waktu $O(|A|)$, di mana $|A|$ menunjukkan jumlah simbol, tidak termasuk tanda kurung, dalam A .

$|A|$ juga merupakan jumlah node dalam pohon formula A . Buktinya diserahkan sebagai latihan.

Korolari 2.2.5 Untuk setiap formula A dan B , dan setiap interpretasi σ , persamaan berikut berlaku.

$$\begin{aligned}\sigma(A \vee B) &= \sigma(A) \vee \sigma(B) \\ \sigma(A \wedge B) &= \sigma(A) \wedge \sigma(B) \\ \sigma(\neg A) &= \neg \sigma(A)\end{aligned}$$

Nilai kebenaran suatu formula dalam logika proposisional mencerminkan dua prinsip dasar logika Boolean: prinsip bivalensi, yang hanya memperbolehkan dua nilai kebenaran, dan prinsip ekstensibilitas bahwa nilai kebenaran suatu formula umum hanya bergantung pada nilai kebenaran bagian-bagiannya, bukan pada makna informalnya.

Model, Keterpuasan, dan Validitas

Interpretasi memainkan peran yang sangat penting dalam logika proposisional dan memperkenalkan banyak konsep penting.

Definisi 2.2.6 Diberikan suatu formula A dan interpretasi σ , σ adalah model dari A jika $eval(A, \sigma) = 1$. Jika A memiliki model, maka A dapat dipenuhi.

Diberikan suatu himpunan V_P variabel proposisional, kita akan menggunakan All_P untuk menunjukkan himpunan semua interpretasi atas V_P . Misalnya, jika $V_P = \{p, q\}$, maka $All_P = \{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}, \{\neg p, \neg q\}\}$, atau disingkat sebagai $All_P = \{pq, p\bar{q}, \bar{p}q, \bar{p}\bar{q}\}$. Secara umum, $|All_P| = 2^{|V_P|}$. Kita dapat menggunakan *eval* untuk mencari model A dengan memeriksa setiap interpretasi dalam All_P .

Definisi 2.2.7 Diberikan suatu rumus A , misalkan $M(A)$ adalah himpunan semua model dari A . Jika $M(A) = \emptyset$, maka A tidak dapat dipenuhi. Jika $M(A) = All_P$, yaitu, setiap interpretasi adalah model dari A , maka A valid, atau tautologi. Kita tulis $\models A$ jika A valid.

Valid berbeda dengan berguna atau efisien. Misalnya, pernyataan $p \vee \neg p$ valid tetapi tidak berguna maupun efisien (rumus T lebih tepat).

Jika ada tabel kebenaran untuk suatu rumus A , $M(A)$ dapat dengan mudah diperoleh dari tabel kebenaran, karena nilai kebenaran di bawah variabel A adalah semua interpretasi dan model A diberikan pada baris-baris tabel di mana nilai kebenaran A adalah 1.

Contoh 2.2.8 Misalkan $V_P = \{p, q\}$. Jika A adalah $\neg p \rightarrow \neg q$, maka $M(A) = \{pq, p\bar{q}, \bar{p}\bar{q}\}$. Jika B adalah $p \vee (\neg q \rightarrow \neg p)$, maka $M(B) = All_P$. Dari $M(A)$ dan $M(B)$, kita tahu bahwa A dan B keduanya dapat dipenuhi, dan B juga valid, yaitu, $\models B$ adalah benar.

Suatu rumus yang valid adalah rumus yang selalu benar dalam setiap interpretasi, tidak peduli nilai kebenaran apa pun yang mungkin diambil oleh variabelnya, yaitu, setiap interpretasi adalah modelnya. Contoh paling sederhana adalah T, atau $p \vee \neg p$. Ada banyak rumus yang ingin kita ketahui apakah valid, dan ini dilakukan dengan apa yang disebut pembuktian teorema, baik secara manual maupun otomatis.

Kita dapat menganggap rumus yang valid sebagai rumus yang menangkap kebenaran logika fundamental. Misalnya, sifat transitivitas implikasi menyatakan bahwa jika satu pernyataan menyiratkan pernyataan kedua, dan pernyataan kedua menyiratkan pernyataan ketiga, maka pernyataan pertama menyiratkan pernyataan ketiga. Dalam logika, transitivitas dapat dinyatakan sebagai rumus berikut:

$$(p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r)$$

Validitas rumus di atas menegaskan kebenaran sifat implikasi ini. Ada banyak sifat seperti ini untuk operator logika lainnya, seperti komutativitas dan asosiatifitas $\wedge$ dan $\vee$, dan semuanya dapat dinyatakan dengan tautologi.

Teorema 2.2.9 (Substitusi) Jika A valid, maka setiap contoh A juga valid. Artinya, jika p adalah variabel proposisional dalam A dan B adalah rumus apa pun, maka $A[p \sim B]$ valid.

Bukti Untuk setiap interpretasi σ dari $A[p \sim B]$, misalkan $\sigma(p) = \sigma(B)$. Dengan menerapkan σ pada pohon rumus A dan $A[p \sim B]$, maka nilai kebenaran dari semua simpul A harus identik dengan nilai kebenaran dari simpul yang sesuai di $A[p \sim B]$. Karena A valid, simpul akar A harus bernilai 1 di bawah σ , sehingga simpul akar $A[p \sim B]$ harus memiliki nilai kebenaran yang sama, yaitu 1. Karena σ bersifat sembarang, $A[p \sim B]$ harus valid.

Signifikansi teorema di atas adalah bahwa jika kita memiliki tautologi untuk satu variabel, tautologi tersebut berlaku ketika variabel tersebut digantikan oleh rumus apa pun. Misalnya, dari $p \vee \neg p$, kita memiliki $A \vee \neg A$ untuk setiap A . Di sisi lain, ketika kita mencoba membuktikan tautologi yang melibatkan simbol $A, B, \dots$, kita dapat memperlakukan setiap simbol ini sebagai variabel proposisional. Misalnya, untuk membuktikan $(A \wedge B) \leftrightarrow (B \wedge A)$, kita dapat membuktikan $(p \wedge q) \leftrightarrow (q \wedge p)$ sebagai gantinya.

Suatu rumus yang tidak dapat dipenuhi adalah rumus yang tidak memiliki model apa pun, yaitu, tidak ada interpretasi yang menjadi modelnya. Contoh paling sederhana adalah $\perp$, atau $p \wedge \neg p$. Validitas dan ketidakpuasan adalah konsep ganda, seperti yang dinyatakan oleh proposisi berikut.

Proposisi 2.2.10 Suatu rumus A valid jika dan hanya jika $\neg A$ tidak dapat dipenuhi.

Dari sudut pandang pembuktian teorema, membuktikan validitas suatu rumus sama sulitnya (atau semudahnya) dengan membuktikan ketidakpuasannya.

Teorema berikut menunjukkan hubungan antara logika dan teori himpunan.

Teorema 2.2.11 Untuk setiap rumus A dan B di atas V_p ,

- (a) $M(A \vee B) = M(A) \cup M(B)$.
- (b) $M(A \wedge B) = M(A) \cap M(B)$.
- (c) $M(\neg A) = All_p - M(A)$.

Bukti Untuk menunjukkan $M(A \vee B) = M(A) \cup M(B)$, kita perlu menunjukkan bahwa, untuk interpretasi apa pun σ , $\sigma \in M(A \vee B)$ jika dan hanya jika $\sigma \in M(A) \cup M(B)$. Berdasarkan definisi, $\sigma \in M(A \vee B)$ jika dan hanya jika $\sigma(A \vee B) = 1$. Berdasarkan Korolari 2.2.5, $\sigma(A \vee B) = \sigma(A) \vee \sigma(B)$. Jadi $\sigma(A \vee B) = 1$ jika dan hanya jika $\sigma(A) = 1$ atau $\sigma(B) = 1$. $\sigma(A) = 1$ atau $\sigma(B) = 1$ jika dan hanya jika $\sigma(A) \vee \sigma(B) = 1$. $\sigma(A) \vee \sigma(B) = 1$ jika dan hanya jika $\sigma \in M(A) \cup M(B)$.

Bukti (b) dan (c) diserahkan sebagai latihan.

Contoh 2.2.12 Misalkan $V_p = \{p, q\}$. $M(p) = \{pq, p\bar{q}\}$, $M(q) = \{pq, \bar{p}q\}$; $M(\neg p) = All_p - M(p) = \{\bar{p}\bar{q}, \bar{p}q\}$, dan $M(\neg p \vee q) = M(\neg p) \cup M(q) = \{pq, \bar{p}q, \bar{p}\bar{q}\}$.

Kesetaraan

Dalam bahasa alami, satu pernyataan dapat diungkapkan dalam berbagai bentuk dengan makna yang sama. Misalnya, “Jika saya memenangkan lotere, saya pasti kaya.” Makna ini juga dapat diungkapkan “Karena saya tidak kaya, saya tidak memenangkan lotere.” Dengan memperkenalkan p untuk “Saya memenangkan lotere” dan q untuk “Saya kaya,” pernyataan pertama menjadi $p \rightarrow q$ dan pernyataan kedua menjadi $\neg q \rightarrow \neg p$. Ternyata $M(p \rightarrow q) = M(\neg q \rightarrow \neg p)$, seperti yang ditunjukkan Tabel 2.2 bahwa kedua rumus tersebut memiliki himpunan model yang sama. Dalam logika, rumus-rumus ini dianggap setara.

Rumus $\neg q \rightarrow \neg p$ disebut kontraposisif dari implikasi $p \rightarrow q$. Tabel kebenaran menunjukkan bahwa implikasi dan kontraposisifnya setara: Keduanya hanyalah cara berbeda untuk mengatakan hal yang sama. Sebaliknya, kebalikan dari $p \rightarrow q$ adalah rumus $q \rightarrow p$, yang memiliki himpunan model yang berbeda (seperti yang ditunjukkan pada Tabel 2.2).

Tabel 2.2 Kesetaraan $p \rightarrow q$, $\neg p \vee q$, dan kontraposisif $\neg q \rightarrow \neg p$

p	q	$\neg p$	$\neg q$	$p \rightarrow q$	$\neg p \vee q$	$\neg q \rightarrow \neg p$	$q \rightarrow p$
1	1	0	0	1	1	1	1
1	0	0	1	0	0	0	1
0	1	1	0	1	1	1	0
0	0	1	1	1	1	1	1

Definisi 2.2.13 Diberikan dua rumus A dan B , A dan B secara logis ekuivalen jika $M(A) = M(B)$, yang dilambangkan dengan $A \equiv B$.

$A \equiv B$ berarti bahwa, untuk setiap interpretasi σ , $\sigma(A) = \sigma(B)$, sehingga $\sigma \in M(A)$ jika dan hanya jika $\sigma \in M(B)$. Relasi $\equiv$ atas rumus adalah relasi ekuivalensi karena $\equiv$ bersifat refleksif, simetris, dan transitif. Relasi $\equiv$ juga merupakan relasi kongruensi karena $A \equiv C$ dan $B \equiv D$ menyiratkan bahwa $\neg A \equiv \neg C$ dan $A \circ B \equiv C \circ D$ untuk setiap operator biner $\circ$. Sifat ini memungkinkan kita untuk memperoleh rumus ekuivalen dengan mengganti sebagian rumus dengan rumus ekuivalen lainnya. Relasi $\equiv$ memainkan peran yang sangat penting dalam logika karena digunakan untuk menyederhanakan rumus, untuk mengubah rumus menjadi bentuk normal yang setara, atau untuk memberikan definisi alternatif untuk operator logika.

Dalam aritmatika, kita cukup menulis $s = t$ untuk menyatakan bahwa suku s, t mewakili fungsi yang sama. Misalnya, “ $(x + y)^2 = x^2 + 2xy + y^2$ ” menyatakan nilai yang sama dari suku-suku di kedua sisi “ $=$ ”. Namun, dalam logika proposisional, kita menggunakan tanda kesamaan seperti $A = B$ hanya untuk identitas sintaksis rumus A dan B . Oleh karena itu, kesetaraan rumus harus dinyatakan secara berbeda, seperti $A \equiv B$.

Teorema 2.2.14 Untuk setiap rumus A, B , dan C , di mana B adalah subrumus dari A , dan $B \equiv C$, maka $A \equiv A[B \sim C]$.

Bukti Untuk interpretasi σ apa pun, $\sigma(B) = \sigma(C)$, karena $B \equiv C$. Terapkan σ pada pohon rumus A dan $A[B \sim C]$ dan bandingkan nilai kebenaran dari semua simpul yang sesuai dari kedua pohon tersebut, dengan mengabaikan subpohon sebenarnya dari B dan C . Karena $\sigma(B) = \sigma(C)$, maka keduanya harus memiliki nilai kebenaran yang sama, yaitu, $\sigma(A) = \sigma(A[B \sim C])$.

Relasi kesetaraan banyak digunakan untuk menyederhanakan rumus dan memiliki kepentingan praktis yang nyata dalam ilmu komputer. Penyederhanaan rumus dalam perangkat lunak dapat membuat program lebih mudah dibaca dan dipahami. Program yang disederhanakan juga dapat berjalan lebih cepat, karena membutuhkan lebih sedikit operator. Dalam desain perangkat keras, penyederhanaan rumus dapat mengurangi jumlah gerbang logika pada sebuah chip karena sirkuit digital dapat diekspresikan oleh rumus logika.

Meminimalkan rumus logika sesuai dengan mengurangi jumlah gerbang dalam sirkuit. Keuntungan dari minimisasi gerbang berpotensi sangat besar: Sebuah chip dengan lebih sedikit gerbang lebih kecil, mengkonsumsi lebih sedikit daya, memiliki tingkat cacat yang lebih rendah, dan lebih murah untuk diproduksi.

Misalkan sebuah formula A mengandung k variabel proposisional, maka A dapat dipandang sebagai salah satu fungsi Boolean $f: \{1, 0\}^k \rightarrow \{1, 0\}$. Misalnya, $\neg p \vee q$ mengandung dua variabel dan dapat dianggap sebagai fungsi Boolean $f(p, q)$. Tabel kebenaran (Tabel 2.2) mengungkapkan bahwa $f(p, q)$, yaitu $\neg p \vee q$, selalu memiliki nilai kebenaran yang sama dengan $p \rightarrow q$, sehingga f dan $\rightarrow$ adalah fungsi yang sama. Akibatnya, kita dapat menggunakan $\neg p \vee q$ sebagai definisi dari $p \rightarrow q$. Sebagai contoh lain, fungsi *if – then – else, ite*: $\{1, 0\}^3 \rightarrow \{1, 0\}$, dapat didefinisikan dengan $ite(1, B, C) = B$ dan $ite(0, B, C) = C$, alih-alih menggunakan tabel kebenaran delapan baris. Proposisi berikut mencantumkan banyak pasangan rumus yang setara.

Proposisi 2.2.15 Untuk rumus A, B , dan C ,

$A \vee A \equiv A$	$A \wedge A \equiv A$
$A \vee B \equiv B \vee A$	$A \wedge B \equiv B \wedge A$
$(A \vee B) \vee C \equiv A \vee (B \vee C)$	$(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$
$A \vee \perp \equiv A$	$A \wedge \top \equiv A$
$A \vee \neg A \equiv \top$	$A \wedge \neg A \equiv \perp$
$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$	$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
$\neg \top \equiv \perp; \quad \neg \perp \equiv \top;$	$\neg \neg A \equiv A$
$\neg(A \vee B) \equiv \neg A \wedge \neg B$	$\neg(A \wedge B) \equiv \neg A \vee \neg B$
$A \rightarrow B \equiv \neg A \vee B$	$A \rightarrow B \equiv \neg B \rightarrow \neg A$
$A \oplus B \equiv (A \vee B) \wedge (\neg A \vee \neg B)$	$A \leftrightarrow B \equiv (A \vee \neg B) \wedge (\neg A \vee B)$

Beberapa kesetaraan di atas memiliki nama khusus. Misalnya, $\neg\neg A \equiv A$ disebut negasi ganda. $\neg(A \vee B) \equiv \neg A \wedge \neg B$ dan $\neg(A \wedge B) \equiv \neg A \vee \neg B$ disebut hukum de Morgan. $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ dan $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$ disebut hukum distributif.

Salah satu cara untuk menunjukkan bahwa dua rumus setara adalah dengan membuktikan validitas suatu rumus, seperti yang diberikan oleh proposisi berikut:

Proposisi 2.2.16 Untuk setiap rumus A dan B , $A \equiv B$ jika dan hanya jika $\models A \leftrightarrow B$.

Bukti $A \equiv B$ berarti, untuk setiap interpretasi σ , $\sigma(A) = \sigma(B)$, yang berarti $\sigma(A \leftrightarrow B) = 1$, yaitu, setiap σ adalah model dari $A \leftrightarrow B$, yaitu, $A \leftrightarrow B$ valid.

Oleh karena itu, $A \equiv T$ jika dan hanya jika $\models A$ ketika $B = T$. Proposisi di atas menunjukkan hubungan antara $\equiv$, yang merupakan notasi semantik, dan $\leftrightarrow$, yang merupakan simbol sintaksis. Selain itu, Teorema 2.2.9 memungkinkan kita untuk membuktikan kesetaraan menggunakan metode tabel kebenaran, memperlakukan A, B , dan C sebagai variabel proposisional. Misalnya, dari $(p \rightarrow q) \equiv (\neg p \vee q)$, kita tahu $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$ valid; dari validitas $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$, kita tahu $(A \rightarrow B) \leftrightarrow (\neg A \vee B)$ valid untuk setiap rumus A dan B , sehingga $(A \rightarrow B) \equiv (\neg A \vee B)$.

Keterkaitan

Dalam bahasa alami, diberikan sekumpulan premis, kita ingin mengetahui kesimpulan apa yang dapat ditarik dari premis tersebut. Jika kita merepresentasikan premis-premis ini dengan satu rumus tunggal, maka rumus tersebut akan menjadi konjungsi dari premis-premis tersebut. Karena $\wedge$ bersifat komutatif dan asosiatif, konjungsi rumus dapat dengan mudah ditulis sebagai himpunan S rumus. Artinya, kita mengasumsikan bahwa $A_1 \wedge A_2 \wedge \dots \wedge A_n$ setara dengan $S = \{A_1, A_2, \dots, A_n\}$. Misalnya, $\{A, B\} \equiv A \wedge B$.

Untuk menangkap hubungan antara premis dan kesimpulan dalam logika, kita memiliki gagasan tentang implikasi.

Definisi 2.2.17 Diberikan dua rumus A dan B , kita katakan A mengimplikasikan B , atau B adalah konsekuensi logis dari A , yang dilambangkan dengan $A \models B$, jika $M(A) \subseteq M(B)$.

Dengan demikian, karena $M(T) = All_p \subseteq M(B)$ menyiratkan $M(B) = All_p$, $\models B$ dan $T \models B$ memiliki arti yang sama, yaitu, B valid.

Definisi di atas memungkinkan banyak rumus yang tidak relevan menjadi konsekuensi logis dari A , termasuk semua tautologi dan rumus yang secara logis setara. Terlepas dari hubungan yang tidak relevan antara A dan B ini, konsep implikasi sangat diperlukan dalam logika. Misalnya, aturan inferensi adalah pola yang mengambil rumus sebagai premis dan mengembalikan rumus sebagai kesimpulan. Untuk memeriksa kebenaran aturan inferensi, kita misalkan premis diwakili oleh rumus A dan kesimpulan diwakili oleh B , dan periksa apakah $A \models B$, yaitu, B merupakan konsekuensi logis dari A . Jika jumlah variabelnya kecil, kita dapat

menggunakan metode tabel kebenaran untuk memeriksa apakah $M(A) \subseteq M(B)$, yaitu, apakah setiap model A tetap menjadi model B .

Contoh 2.2.18 Versi proposisional dari aturan modus ponens mengatakan bahwa diberikan premis $p \rightarrow q$ dan p , maka kita menarik kesimpulan q . Misalkan A adalah $(p \rightarrow q) \wedge p$, maka $M(A) = \{pq\}$, dan $M(q) = \{pq, pq\}$. Karena $M(A) \subseteq M(q)$, maka q merupakan konsekuensi logis dari A , atau $A \models q$.

Proposisi 2.2.19 Relasi $\models$ bersifat transitif, yaitu, jika $A \models B$ dan $B \models C$, maka $A \models C$.

Pembuktiannya diserahkan sebagai latihan.

Definisi 2.2.20 Diberikan suatu rumus A , himpunan $\{B \mid A \models B\}$ disebut teori A dan dilambangkan dengan $T(A)$. Setiap anggota $T(A)$ disebut teorema A .

Dalam definisi di atas, A dapat berupa himpunan rumus untuk mewakili konjungsi anggotanya.

Proposisi 2.2.21 Tiga pernyataan berikut ini setara:

- (a) $A \models B$;
- (b) $M(A) \subseteq M(B)$; dan
- (c) $T(B) \subseteq T(A)$.

Bukti

(a) $\rightarrow$ (b): Berdasarkan definisi, $A \models B$ jika dan hanya jika $M(A) \subseteq M(B)$.

(b) $\rightarrow$ (c): Juga, menurut definisi, $B \models C$ untuk setiap $C \in T(B)$. Jika $A \models B$, maka $A \models C$ karena transitivitas $\models$, sehingga $C \in T(A)$.

(c) $\rightarrow$ (a): Jika $T(B) \subseteq T(A)$, karena $B \in T(B)$, maka $B \in T(A)$, sehingga $A \models B$.

Corollary 2.2.22 Untuk setiap rumus A , $T(T) \subseteq T(A) \subseteq T(\perp)$.

Bukti Ini berlaku karena $\perp \models A$ dan $A \models T$. Perhatikan bahwa $T(T)$ mengandung setiap tautologi dan $T(\perp)$ mengandung setiap rumus.

Akibat 2.2.23 Untuk setiap rumus A dan B , pernyataan berikut ini setara: (a) $A \equiv B$; (b) $M(A) = M(B)$; dan (c) $T(A) = T(B)$.

Bukti Karena $A \equiv B$ jika dan hanya jika $A \models B$ dan $B \models A$, maka korolari ini berlaku dengan menerapkan Proposisi 2.2.21 dua kali.

Secara intuitif, kita dapat menganggap formula A sebagai batasan untuk menandai beberapa interpretasi sebagai "contoh tandingan" dari validitasnya. Semakin kuat batasannya, semakin kecil himpunan modelnya. Batasan terkuat adalah $\perp$ (tidak ada interpretasi yang

tersisa) dan batasan terlemah adalah T (tidak ada batasan). Di sisi lain, $T(A)$ mengumpulkan semua batasan yang tidak menandai interpretasi apa pun dalam $M(A)$ sebagai "contoh tandingan."

Proposisi 2.2.24 Misalkan A dan B adalah himpunan formula. Jika $A \subseteq B \subseteq T(A)$, maka $M(A) = M(B)$.

Bukti Misalkan $C = B - A$. Karena $B \subseteq T(A)$, maka $C \subseteq T(A)$, atau $A \models C$, yang berarti $M(A) \subseteq M(C)$. Dengan demikian, $M(B) = M(A \cup C) = M(A \wedge C) = M(A) \cap M(C) = M(A)$, karena $M(A) \subseteq M(C)$.

Akibat 2.2.25 Jika $C \in T(A)$, maka $T(A) = T(A \wedge C)$.

Bukti Misalkan $B = A \cup \{C\}$, maka $A \subseteq B \subseteq T(A)$, sehingga $M(A) = M(B) = M(A \wedge C)$. Oleh karena itu, $T(A) = T(A \wedge C)$ berdasarkan Akibat 2.2.23.

Akibat 2.2.26 Jika $\perp \in T(A)$, maka A tidak dapat dipenuhi.

Bukti Misalkan $B = A \cup \{\perp\}$, maka $A \subseteq B \subseteq T(A)$, sehingga $M(A) = M(B) = M(A \wedge \perp) = M(\perp) = \emptyset$.

Salah satu tugas pembuktian teorema adalah menunjukkan bahwa, diberikan himpunan aksioma A , B adalah teorema dari A , yaitu, $B \in T(A)$. Jika kita menggunakan operator $\rightarrow$, maka $A \rightarrow B$ valid jika dan hanya jika B merupakan konsekuensi logis dari A .

Proposisi 2.2.27 Misalkan A dan B adalah rumus, $A \models B$ jika dan hanya jika $\models A \rightarrow B$.

Bukti $A \models B$ berarti, untuk interpretasi apa pun σ , jika $\sigma(A) = 1$, maka $\sigma(B)$ harus 1; jika $\sigma(A) = 0$, kita tidak peduli dengan nilai $\sigma(B)$. Kedua kasus tersebut dapat digabungkan sebagai $\sigma(A) \rightarrow \sigma(B) = 1$, atau $\sigma(A \rightarrow B) = 1$, yang berarti $A \rightarrow B$ valid, karena σ adalah interpretasi apa pun.

Proposisi di atas menunjukkan hubungan antara $\models$ (notasi semantik) dan $\rightarrow$ (simbol sintaksis). Penutupan relasi implikasi di bawah substitusi menggeneralisasi fakta bahwa dari $p \wedge q \models p$, semua implikasi dalam bentuk $A \wedge B \models A$ muncul dari substitusi A untuk p dan B untuk q , seperti halnya penutupan relasi ekuivalensi di bawah substitusi.

Pembuktian Teorema dan Masalah SAT

Konsep validitas, ketidakpuasan, atau implikasi menimbulkan masalah komputasi yang sulit. Ketiga masalah ini memiliki tingkat kesulitan yang sama dan dapat diwakili oleh pembuktian teorema, yang meminta untuk memeriksa apakah suatu rumus $A \in T(T(A) \text{ valid})$, atau $\perp \in T(A)$ (A tidak dapat dipenuhi), atau A berada di $T(B)$ (B menyiratkan A). Pembuktian

teorema dengan tabel kebenaran akan cepat kehabisan tenaga: Suatu rumus dengan n variabel memiliki tabel kebenaran dengan 2^n interpretasi, sehingga upaya yang dibutuhkan meningkat secara eksponensial dengan jumlah variabel. Untuk rumus dengan hanya 30 variabel, ada lebih dari satu miliar interpretasi yang harus diperiksa!

Masalah umum untuk memutuskan apakah suatu rumus A dapat dipenuhi disebut masalah SAT. Salah satu pendekatan untuk menyelesaikan SAT adalah dengan membuat tabel kebenaran dan memeriksa setiap baris untuk menemukan model. Seperti halnya pengujian validitas, pendekatan ini dengan cepat menjadi lambat untuk rumus dengan banyak variabel.

Kabar baiknya adalah bahwa SAT termasuk dalam kelas masalah NP , yang solusinya dapat diperiksa dalam waktu polinomial. Misalnya, diberikan interpretasi σ , kita dapat memeriksa dalam waktu linear (dalam hal ukuran A) apakah σ merupakan model dari A , sehingga menunjukkan bahwa A dapat dipenuhi. Ini dapat dilakukan menggunakan $eval(A, \sigma)$. Jika suatu masalah dapat diselesaikan dalam waktu polinomial (terhadap ukuran input), maka masalah tersebut termasuk dalam kelas P . SAT tidak diketahui berada di P , dan pembuktian teorema tidak diketahui berada di NP .

Kabar buruknya adalah SAT diketahui sebagai masalah tersulit di NP . Artinya, jika ada algoritma waktu polinomial untuk menyelesaikan SAT, maka setiap masalah di NP dapat diselesaikan dalam waktu polinomial. Masalah-masalah ini mencakup banyak masalah penting lainnya yang melibatkan penjadwalan, perutean, alokasi sumber daya, dan verifikasi sirkuit di berbagai disiplin ilmu termasuk pemrograman, aljabar, keuangan, dan teori politik. Ini adalah hasil terkenal dari Cook dan Levin yang membuktikan bahwa SAT adalah masalah NP -lengkap pertama. Apakah ini berarti kita tidak dapat menemukan algoritma yang membutuhkan waktu $O(n^{100})$, di mana n adalah ukuran A , untuk memutuskan apakah A dapat dipenuhi? Tidak ada yang tahu. $P = NP$? Ini adalah masalah terbuka paling terkenal dalam ilmu komputer. Ini juga salah satu dari tujuh masalah milenium: Clay Institute akan memberi Anda Rp. 1.000.000 jika Anda memecahkan masalah $P = NP$. Jika $P = NP$, maka penyelesaian SAT akan berada di P ; pembuktian teorema (dalam logika proposisional) juga akan berada di P . Jika $P \neq NP$, maka pembuktian teorema tidak akan berada di NP : Ini termasuk dalam komplemen masalah NP -lengkap. Banyak hal bergantung pada jawabannya, dan konsensus umum adalah $P \neq NP$. Dalam 20 tahun terakhir, telah ada kemajuan yang menarik dalam penyelesaian SAT, yang merupakan alat perangkat lunak untuk kepuasan proposisional, untuk aplikasi praktis seperti verifikasi sirkuit digital. Alat perangkat lunak ini menemukan penugasan yang memuaskan dengan efisiensi yang luar biasa bahkan untuk rumus dengan ribuan variabel. Pada dua bab berikutnya, kita akan mempelajari metode pembuktian teorema dan pemecah SAT.

2.3 BENTUK NORMAL

Yang dimaksud dengan “bentuk normal” adalah setiap rumus dapat diubah menjadi rumus yang setara dengan batasan sintaksis tertentu pada penggunaan dan posisi operator Boolean. Jika semua rumus yang setara memiliki bentuk normal yang unik, bentuk normal tersebut juga disebut “bentuk kanonik.”

Karena penggunaan tabel kebenaran untuk membuktikan relasi kesetaraan mahal jika jumlah variabelnya besar, pendekatan alternatifnya adalah menggunakan aljabar untuk membuktikan kesetaraan. Banyak operator berbeda dapat muncul dalam rumus proposisional, jadi langkah pertama yang berguna adalah menyingkirkan semua kecuali tiga: $\wedge$, $\vee$, dan $\neg$. Ini mudah karena setiap operator setara dengan rumus sederhana yang hanya menggunakan ketiga operator ini. Misalnya, $A \rightarrow B$ setara dengan $\neg A \vee B$.

Dengan menggunakan relasi kesetaraan yang dibahas di bagian sebelumnya, setiap rumus proposisional dapat dibuktikan setara dengan bentuk kanonik. Apa hubungannya dengan kesetaraan? Itu mudah: Untuk membuktikan bahwa dua rumus setara, ubahlah keduanya ke bentuk kanonik pada himpunan variabel yang muncul dalam rumus apa pun. Sekarang, jika dua rumus setara dengan bentuk kanonik yang sama, maka kedua rumus tersebut pasti setara. Sebaliknya, jika dua rumus setara, maka keduanya akan memiliki bentuk kanonik yang sama. Kita juga dapat menggunakan bentuk kanonik untuk menunjukkan apakah suatu rumus valid atau tidak dapat dipenuhi: Rumus tersebut valid jika bentuk kanoniknya adalah T ; rumus tersebut tidak dapat dipenuhi jika bentuk kanoniknya adalah $\perp$.

Pada bagian ini, kita akan menyajikan empat bentuk normal:

- Bentuk normal negasi (NNF)
- Bentuk normal konjungtif (CNF)
- Bentuk normal disjungtif (DNF)
- Bentuk normal ITE (jika-maka-lain) (INF)

Dari keempat bentuk normal tersebut, terdapat bentuk kanonik untuk tiga bentuk terakhir. Namun, bentuk kanonik yang diturunkan dari DNF atau CNF pada dasarnya adalah salinan tabel kebenaran, sehingga tidak cukup efektif untuk menunjukkan relasi kesetaraan. Hanya yang terakhir, INF, yang mungkin memiliki ukuran kompak sehingga kesetaraan dua rumus dapat dengan mudah ditetapkan dalam bentuk kanonik ini.

Kita akan mendefinisikan bentuk normal dengan memberikan serangkaian aturan transformasi, yang merupakan serangkaian pasangan rumus, satu adalah sisi kiri dan yang lainnya adalah sisi kanan dari aturan tersebut. Yang terpenting, sisi kiri setiap aturan secara logis setara dengan sisi kanannya. Setiap aturan menangani rumus yang tidak dalam bentuk normal dan penerapan aturan pada rumus tersebut akan menghasilkan rumus baru yang lebih dekat ke bentuk normal. Untuk mengubah rumus apa pun menjadi rumus dalam bentuk normal, kita memilih sebuah aturan, mencocokkan sisi kiri aturan dengan subrumus dan mengganti subrumus dengan sisi kanan aturan.

Prosedur umum untuk mendapatkan bentuk normal adalah sebagai berikut:

1. Jika rumus sudah dalam bentuk normal, berhenti dengan sukses.
2. Jika tidak, maka rumus tersebut mengandung subformula yang melanggar kriteria bentuk normal.
3. Pilih subformula tersebut dan temukan sisi kiri dari aturan yang sesuai.
4. Ganti subformula dengan sisi kanan aturan, kembali ke langkah 1 dan lanjutkan.

Bentuk Normal Negasi (NNF)

Definisi 2.3.1 Suatu rumus proposisional A adalah bentuk normal negasi (NNF) jika argumen dari simbol negasi $\neg$ adalah variabel proposisional.

Formula A dikatakan sebagai literal jika A adalah variabel (literal positif) atau A adalah $\neg p$ (literal negatif), di mana p adalah variabel. Dalam NNF, $\neg$ hanya muncul dalam literal negatif (sebagai subformula dari A); jika kita mengganti literal negatif dengan variabel, maka tidak akan ada $\neg$ dalam formula yang dihasilkan.

Proposisi 2.3.2 Setiap formula proposisional dapat ditransformasikan menjadi NNF yang setara.

Buktinya dilakukan dengan algoritma berikut, yang hanya menggunakan relasi kesetaraan dalam transformasi.

Aturan Kesetaraan untuk Memperoleh NNF

Algoritma 2.3.3 Algoritma untuk mengubah formula menjadi NNF menggunakan tiga kelompok aturan berikut:

1. Gunakan kesetaraan berikut untuk menghilangkan $\rightarrow$, $\oplus$, dan $\leftrightarrow$ dari formula.

$$A \rightarrow B \equiv \neg A \vee B$$

$$A \oplus B \equiv (A \vee B) \wedge (\neg A \vee \neg B)$$

$$A \leftrightarrow B \equiv (A \vee \neg B) \wedge (\neg A \vee B)$$

2. Selain $\neg T \equiv \perp$, $\neg \perp \equiv T$, dan $\neg \neg A \equiv A$, gunakan hukum de Morgan untuk menurunkan $\neg$ ke variabel:

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

3. (Opsional) Gunakan $A \vee \perp \equiv A$, $A \vee T \equiv T$, $A \wedge \perp \equiv \perp$, dan $A \wedge T \equiv A$, untuk menghilangkan T dan $\perp$ tambahan dari rumus.

Misalnya, T dan $\neg p \vee q$ adalah NNF. $\neg(\neg p \vee q)$ bukan NNF; kita dapat mengubahnya menjadi $p \wedge \neg q$, yang merupakan NNF. Jika rumus masukan hanya berisi $\neg$, $\vee$, $\wedge$, dan $\rightarrow$, hanya dibutuhkan waktu linier untuk mendapatkan NNF dengan bekerja secara rekursif dari atas ke bawah. Jika $\leftrightarrow$ atau $\oplus$ ada, waktu komputasi dan ukuran rumus keluaran mungkin eksponensial dalam hal ukuran aslinya, karena argumen $\leftrightarrow$ dan $\oplus$ diduplikasi selama transformasi.

Definisi 2.3.4 Diberikan NNF A , jika kita mengganti setiap kemunculan $\wedge$ dengan $\vee$, $\vee$ dengan $\wedge$, literal positif dengan negasinya, dan literal negatif $\neg p$ dengan

pasangannya p , kita memperoleh rumus B , yang disebut dual dari A , yang dilambangkan dengan $B = \text{dual}(A)$.

Misalnya, jika A adalah $p \wedge (q \vee (\neg p \wedge \neg r))$, maka $\text{dual}(A) = \neg p \vee (\neg q \wedge (p \vee r))$. Perhatikan bahwa baik A maupun $\text{dual}(A)$ adalah NNF.

Proposisi 2.3.5 Jika A berada dalam NNF, maka $\text{dual}(A)$ juga berada dalam NNF dan $\text{dual}(A) \equiv \neg A$.

Proposisi ini dapat dengan mudah dibuktikan dengan hukum de Morgan.

Bentuk Normal Konjungtif (CNF)

Definisi 2.3.6 Dalam logika proposisional, sebuah klausa adalah $\perp$ (klausa kosong), literal (klausa unit), atau disjungsi literal.

Klausa juga disebut maxterm dalam komunitas desain sirkuit.

Definisi 2.3.7 Sebuah formula adalah bentuk normal konjungtif (CNF) jika merupakan konjungsi klausa.

CNF juga disebut ekspresi hasil kali jumlah (POS). Ide dari bentuk normal konjungtif adalah bahwa setiap jenis penghubung muncul pada ketinggian yang berbeda dalam formula. Simbol negasi, $\neg$, berada di posisi terendah, hanya dengan variabel sebagai argumennya. Simbol disjungsi, $\vee$, berada di tengah, hanya dengan literal sebagai argumennya. Simbol konjungsi, $\wedge$, berada di atas, mengambil disjungsi atau literal sebagai argumennya. Beberapa operator mungkin hilang dalam CNF. Misalnya, $p \wedge q$ adalah CNF, yang berisi dua klausa unit: p dan q .

Aturan Kesetaraan untuk Memperoleh CNF

- Tahap 1: Aturan kesetaraan untuk memperoleh NNF.
- Tahap 2: $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$, di mana argumen $\wedge$ dan $\vee$ dapat dipertukarkan.

Proposisi 2.3.8 Setiap rumus proposisional dapat diubah menjadi CNF yang setara.

Untuk menunjukkan bahwa proposisi tersebut benar, pertama-tama kita mengubah rumus tersebut menjadi NNF dan kemudian menerapkan hukum distribusi untuk memindahkan $\vee$ di bawah $\wedge$. Proses ini akan berakhir karena kita hanya memiliki sejumlah $\vee$ dan $\wedge$ yang terbatas.

Contoh 2.3.9 Untuk mengubah $p_1 \leftrightarrow (p_2 \leftrightarrow p_3)$ menjadi CNF, pertama-tama kita menggunakan $A \leftrightarrow B \equiv (\neg A \vee B) \wedge (A \vee \neg B)$ untuk menghilangkan $\leftrightarrow$ kedua:

$$p_1 \leftrightarrow ((\neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_3))$$

lalu hapus $\leftrightarrow$ pertama:

$$\left(\neg p_1 \vee ((\neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_3)) \right) \wedge \left(p_1 \vee \neg((\neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_3)) \right)$$

NNF-nya adalah

$$\left(\neg p_1 \vee ((\neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_3)) \right) \wedge \left(p_1 \vee ((p_2 \wedge \neg p_3) \vee (\neg p_2 \wedge p_3)) \right)$$

Dengan demikian, kita memperoleh klausa-klausa non-tautologi berikut:

$$\neg p_1 \vee \neg p_2 \vee p_3, \quad \neg p_1 \vee p_2 \vee \neg p_3, \quad p_1 \vee p_2 \vee p_3, \quad p_1 \vee \neg p_2 \vee \neg p_3$$

Karena $\vee$ bersifat komutatif dan asosiatif dan $X \vee X \equiv X$, sebuah klausa dapat direpresentasikan oleh sekumpulan literal. Dalam hal ini, kita menggunakan $|$ untuk $\vee$ dan $\bar{p}$ untuk $\neg p$ dalam sebuah klausa.

Sebagai contoh, klausa-klausa di atas akan ditampilkan sebagai

$$(\bar{p}_1 | \bar{p}_2 | p_3), \quad (\bar{p}_1 | p_2 | \bar{p}_3), \quad (p_1 | p_2 | p_3), \quad (p_1 | \bar{p}_2 | \bar{p}_3)$$

Definisi 2.3.10 Jika suatu klausa mencakup setiap variabel tepat sekali, klausa tersebut lengkap. Suatu CNF disebut CNF lengkap jika setiap klausa dari CNF tersebut lengkap.

Jika suatu klausa C tidak memiliki variabel r , kita dapat mengganti C dengan dua klausa: $\{(C | r), (C | \bar{r})\}$. Dengan mengulangi proses ini untuk setiap variabel yang hilang, kita akan memperoleh CNF lengkap yang setara. Mengubah sekumpulan klausa menjadi CNF lengkap yang setara adalah sah karena untuk setiap rumus A dan B , hubungan berikut ini berlaku:

$$A \equiv (A \vee B) \wedge (A \vee \neg B)$$

yang dapat dibuktikan dengan tabel kebenaran.

Aturan Kesetaraan untuk Memperoleh CNF Lengkap

- Tahap 1: Aturan kesetaraan untuk NNF.
- Tahap 2: $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$, di mana argumen $\wedge$ dan $\vee$ dapat dipertukarkan.
- Tahap 3: $A \equiv (A \vee p) \wedge (A \vee \neg p)$, jika A adalah klausa dan p tidak muncul dalam A .

Proposisi 2.3.11 Setiap rumus proposisional dapat diubah menjadi CNF lengkap yang setara dan unik (hingga komutativitas dan asosiatifitas $\wedge$ dan $\vee$). Artinya, CNF lengkap adalah bentuk kanonik.

Diberikan CNF A , setiap klausa dapat dianggap sebagai batasan pada interpretasi mana yang tidak dapat menjadi model dari A . Misalnya, jika A berisi klausa unit $(\neg p)$, maka semua interpretasi di mana $p \mapsto 1$ (setengah dari semua interpretasi) tidak dapat menjadi model dari A . Jika A berisi klausa biner $(p \mid q)$, maka semua interpretasi di mana $p \mapsto 0, q \mapsto 0$ (ada seperempat dari interpretasi tersebut) tidak dapat menjadi model dari A . Jika A memiliki klausa C yang berisi setiap variabel tepat sekali, maka klausa ini hanya menghilangkan satu interpretasi, yaitu $\neg C$, sebagai model dari A . Jika A memiliki m model atas VP, yaitu $|M(A)| = m$, maka CNF lengkap dari A berisi $2^{|VP|} - m$ klausa, karena setiap klausa menghilangkan tepat satu interpretasi sebagai model dari A . Itulah tepatnya ide untuk pembuktian proposisi di atas.

Konsep CNF lengkap berguna untuk pembuktian teoretis. Sayangnya, ukuran CNF lengkap bersifat eksponensial dalam hal jumlah variabel, terlalu besar untuk aplikasi praktis apa pun. CNF berguna untuk menentukan banyak masalah dalam praktik, karena masalah praktis sering ditentukan oleh serangkaian kendala. Setiap kendala dapat ditentukan oleh serangkaian klausa, dan konjungsi dari semua klausa dari setiap kendala memberi kita spesifikasi lengkap dari masalah tersebut.

Bentuk Normal Disjungtif (DNF)

Jika kita menganggap $\wedge$ sebagai perkalian dan $\vee$ sebagai penjumlahan, maka CNF adalah "hasil perkalian dari penjumlahan." Di sisi lain, "jumlah dari hasil perkalian" adalah bentuk normal disjungtif (DNF). Rumus A dikatakan sebagai hasil perkalian jika A adalah T, literal, atau konjungsi literal. Hasil perkalian juga disebut minterm atau istilah hasil perkalian dalam komunitas desain sirkuit.

Seperti klausa, kita menghilangkan duplikat dalam hasil perkalian dan merepresentasikannya dengan serangkaian literal.

Definisi 2.3.12 Suatu rumus A adalah bentuk normal disjungtif (DNF) jika A merupakan disjungsi dari produk.

DNF juga disebut ekspresi jumlah produk (SOP). Suatu rumus seperti $p \vee \neg q$ atau $p \vee \neg q$ dapat berupa CNF dan DNF.

Aturan Kesetaraan untuk Memperoleh DNF

- Tahap 1: Aturan kesetaraan untuk NNF.
- Tahap 2: $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$, di mana argumen dari $\wedge$ dan $\vee$ dapat ditukar.

Proposisi 2.3.13 Setiap rumus proposisional dapat diubah menjadi DNF yang setara.

Untuk menunjukkan bahwa proposisi tersebut benar, pertama-tama kita mengubah rumus menjadi NNF pada tahap pertama dan kemudian menerapkan hukum distribusi pada tahap kedua untuk memindahkan $\wedge$ di bawah $\vee$.

Setelah rumus A diubah menjadi DNF, sangat mudah untuk memeriksa apakah A dapat dipenuhi: Jika DNF berisi produk di mana setiap variabel muncul paling banyak sekali, maka A dapat dipenuhi, karena kita dapat dengan mudah menetapkan 1 ke setiap literal dari produk

ini dan produk tersebut menjadi benar. Karena merupakan masalah yang sulit untuk memutuskan apakah suatu rumus dapat dipenuhi, maka memperoleh DNF juga sulit, karena ukurannya yang besar.

Seperti CNF, DNF bukanlah bentuk kanonik. Misalnya, kesetaraan berikut ini benar:

$$p \wedge \neg q \vee \neg p \wedge \neg r \vee q \wedge r \equiv \neg p \wedge q \vee p \wedge r \vee \neg q \wedge \neg r$$

Namun, kita tidak dapat mengubah kedua sisi menjadi rumus yang sama.

Definisi 2.3.14 Suatu produk dikatakan lengkap jika mengandung setiap variabel tepat sekali. Suatu DNF dikatakan lengkap jika setiap produk dalam DNF tersebut lengkap.

Suatu produk lengkap dalam DNF A setara dengan suatu model dari A . Misalnya, jika $V_p = \{p, q, r\}$, maka produk $p \wedge q \wedge \neg r$ menentukan model $\{p \rightarrow 1, q \rightarrow 1, r \rightarrow 0\}$ atau $pq\bar{r}$. Jelas, suatu DNF lengkap A memiliki m produk lengkap jika dan hanya jika A memiliki m model. Seperti CNF lengkap, setiap rumus proposisional setara dengan DNF lengkap yang unik, yaitu, DNF lengkap adalah bentuk kanonik.

Aturan Kesetaraan untuk Memperoleh DNF Lengkap

- Tahap 1: Aturan kesetaraan untuk NNF.
- Tahap 2: $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$, di mana argumen dari $\wedge$ dan $\vee$ dapat dipertukarkan.
- Tahap 3: $A \equiv (A \wedge p) \vee (A \wedge \neg p)$, jika A adalah produk dan p tidak muncul di A .

Proposisi 2.3.15 Setiap rumus proposisional dapat diubah menjadi DNF penuh yang setara dan unik (hingga komutativitas dan asosiatifitas dari $\wedge$ dan $\vee$).

DNF Penuh dan CNF Penuh dari Tabel Kebenaran

Tabel kebenaran sering digunakan untuk mendefinisikan fungsi Boolean. Dalam tabel kebenaran, nilai kebenaran fungsi diberikan untuk setiap interpretasi fungsi. Setiap interpretasi sesuai dengan produk penuh dan fungsi dapat didefinisikan sebagai jumlah (disjungsi) dari semua produk di mana fungsi tersebut benar dalam interpretasi yang sesuai.

Contoh 2.3.16 Misalkan kita ingin mendefinisikan fungsi Boolean $f: B^3 \rightarrow B$, di mana $B = \{0, 1\}$ dan nilai kebenaran $f(a, b, c)$ diberikan dalam tabel kebenaran berikut. Mengikuti konvensi EDA (otomasi desain elektronik), kita akan menggunakan $+$ untuk $\vee$, $\cdot$ untuk $\wedge$, dan $\bar{A}$ untuk $\neg A$.

a	b	c	f		Produk	Klausula
0	0	0	0		$m_0 = \bar{a}\bar{b}\bar{c}$	$m_0 = a + b + c$
0	0	1	1		$m_1 = \bar{a}\bar{b}c$	$m_1 = a + b + \bar{c}$

0	1	0	0	$m_2 = \bar{a}b\bar{c}$	$m_2 = a + \bar{b} + c$
0	1	1	1	$m_3 = \bar{a}bc$	$m_3 = a + \bar{b} + \bar{c}$
1	0	0	0	$m_4 = a\bar{b}\bar{c}$	$m_4 = \bar{a} + b + c$
1	0	1	1	$m_5 = a\bar{b}c$	$m_5 = \bar{a} + b + \bar{c}$
1	1	0	0	$m_6 = ab\bar{c}$	$m_6 = \bar{a} + \bar{b} + c$
1	1	1	0	$m_7 = abc$	$m_7 = \bar{a} + \bar{b} + \bar{c}$

Untuk setiap interpretasi σ dari (a, b, c) , kita mendefinisikan produk m_i , di mana i adalah nilai desimal dari σ , ketika σ dipandang sebagai bilangan biner. Untuk setiap m_i , kita juga mendefinisikan klausa M_i dan hubungan antara m_i dan M_i adalah $\neg m_i \equiv M_i$. Untuk fungsi f yang didefinisikan dalam tabel kebenaran, $f = m_1 + m_3 + m_5 = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}c$.

Dalam contoh di atas, untuk setiap model f , kita membuat produk (istilah produk) dalam DNF lengkapnya: Jika variabel bernilai benar dalam model, literal positif dari variabel ini ada dalam produk; jika variabel bernilai salah, literal negatif dari variabel tersebut ada dalam produk. f kemudian didefinisikan oleh DNF lengkap yang terdiri dari produk-produk ini. Dengan kata lain, setiap fungsi Boolean dapat didefinisikan oleh DNF. Kita juga dapat mendefinisikan fungsi yang sama dengan CNF lengkap. Ini dapat diproses sebagai berikut: Pertama, kita mendefinisikan negasi dari f dengan DNF lengkap dari tabel kebenaran:

$$\bar{f} = m_0 + m_2 + m_4 + m_6 + m_7$$

Dengan menerapkan negasi pada kedua sisi persamaan di atas, kita peroleh

$$f = M_0 \cdot M_2 \cdot M_4 \cdot M_6 \cdot M_7$$

di mana $\cdot$ melambangkan $\wedge$ dan $\bar{m}_i \equiv M_i$. Dengan kata lain, f dapat didefinisikan oleh CNF lengkap. Perhatikan bahwa $M_0 \cdot M_2 \cdot M_4 \cdot M_6 \cdot M_7$ adalah dual dari $m_0 + m_2 + m_4 + m_6 + m_7$. Secara umum, dual dari DNF adalah CNF dan dual dari CNF adalah DNF.

Setelah kita mengetahui prinsipnya, kita dapat membangun CNF lengkap secara langsung dari tabel kebenaran suatu rumus. Artinya, untuk setiap interpretasi non-model dalam tabel kebenaran, kita membuat satu klausa lengkap: Jika suatu variabel bernilai benar dalam interpretasi tersebut, literal negatif dari variabel ini ada dalam klausa; jika suatu variabel bernilai salah, literal positif dari variabel tersebut ada dalam klausa.

Contoh 2.3.17 Tabel kebenaran $(p \vee q) \rightarrow \neg r$ memiliki delapan baris dan tiga interpretasi (p, q, r) bukan model: $(1, 1, 1)$, $(1, 0, 1)$, dan $(0, 1, 1)$. Dari ketiga interpretasi ini, kita memperoleh tiga klausa lengkap: $(\neg p \vee \neg q \vee \neg r) \wedge (\neg p \vee q \vee \neg r) \wedge (p \vee \neg q \vee \neg r)$, atau $\{(\bar{p} \mid \bar{q} \mid \bar{r}), (\bar{p} \mid q \mid \bar{r}), (p \mid \bar{q} \mid \bar{r})\}$.

Diagram Keputusan Biner (BDD)

Misalkan $ite : \{1, 0\}^3 \rightarrow \{1, 0\}$ adalah operator “if-then-else” sedemikian sehingga, untuk setiap formula A dan B , $ite(1, A, B) \equiv A$ dan $ite(0, A, B) \equiv B$. Faktanya, ite , 1, dan 0 dapat digunakan untuk mewakili operator logika apa pun. Misalnya, $\neg y \equiv ite(y, 0, 1)$. Tabel 2.3 menunjukkan bahwa setiap operator logika biner dapat diwakili oleh ite . Mengikuti konvensi pada BDD, kita akan menggunakan 1 untuk T dan 0 untuk $\perp$ dalam setiap formula proposisional.

Tabel 2.3 Operator logika biner oleh ite . Kolom pertama menunjukkan keluaran fungsi pada pasangan input $(x, y) = (0, 0), (0, 1), (1, 0)$, dan $(1, 1)$. Kolom kedua menunjukkan bentuk konvensionalnya dan kolom terakhir memberikan ekuivalen dari INF. Pada kolom 3, Y mewakili $ite(y, 1, 0)$ dan $\neg Y$ mewakili $ite(y, 0, 1)$.

Output	Rumus	Rumus ite
0000	0	0
0001	$x \wedge y$	$ite(x, Y, 0)$
0010	$x \wedge \neg y (x > y)$	$ite(x, \neg Y, 0)$
0011	x	$ite(x, 1, 0)$
0100	$\neg x \wedge y (x < y)$	$ite(x, 0, Y)$
0101	y	$ite(y, 1, 0)$
0110	$x \oplus y$	$ite(x, \neg Y, Y)$
0111	$x \vee y$	$ite(x, 1, Y)$
1000	$x \downarrow y$	$ite(x, 0, \neg Y)$
1001	$x \leftrightarrow y$	$ite(x, Y, \neg Y)$
1010	$\neg y$	$ite(y, 0, 1)$
1011	$x \vee \neg y (x \geq y)$	$ite(x, 1, \neg Y)$
1100	$\neg x$	$ite(x, 0, 1)$
1101	$\neg x \vee y (x \leq y)$	$ite(x, Y, 1)$
1110	$x \uparrow y$	$ite(x, \neg Y, 1)$
1111	1	1

Definisi 2.3.18 Suatu rumus dikatakan sebagai *bentuk normal ITE* (INF) jika rumus tersebut adalah 0, 1, atau berbentuk $ite(p, A, B)$ di mana p adalah variabel proposisional, dan A dan B adalah INF.

Sebuah INF menggunakan paling banyak tiga operator: 0 (untuk $\perp$), 1 (untuk T), dan ite . Perhatikan bahwa 1 dan 0 adalah INF, tetapi p bukan; $ite(p, 1, 0)$ adalah INF.

Proposisi 2.3.19 Setiap rumus proposisional dapat diubah menjadi INF yang setara.

Misalnya, misalkan A adalah $\neg p \vee q \wedge \neg r$; INF yang setara adalah $ite(p, ite(q, ite(r, 0, 1), 0), 1)$. Untuk membuktikan proposisi di atas, kami menyediakan algoritma di bawah ini untuk melakukan transformasi.

Algoritma 2.3.20 Algoritma untuk mengubah rumus menjadi INF adalah prosedur rekursif *convertINF*:

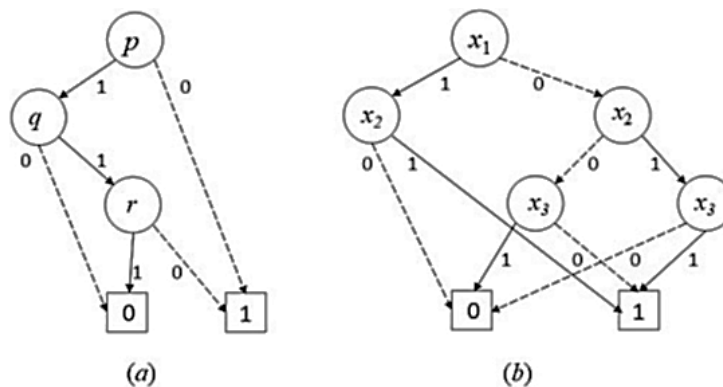
```

proc convertINF( $A$ )
  if  $A = 1$  return 1;
  if  $A = 0$  return 0;
  else if  $A$  contains  $p \in V_p$ 
    return  $ite(p, convertINF(A[p \leftarrow 1]), convertINF(A[p \leftarrow 0]))$ ;
  else return simplify( $A$ ).

```

Perhatikan bahwa $A[p \leftarrow 1]$ (atau $A[p \leftarrow 0]$) mewakili rumus yang dihasilkan dari penggantian setiap kemunculan p dengan 1 (atau 0) dalam A . Subrutin *simplify*(A) akan mengembalikan nilai kebenaran dari rumus A yang tidak memiliki variabel proposisional. Rumus yang dikembalikan oleh *convertINF* hanya berisi *ite*, 0, dan 1 sebagai operator logika. Semua variabel proposisional muncul sebagai argumen pertama dari *ite*.

Contoh 2.3.21 Misalkan A adalah $\neg p \vee q \wedge \neg r$, maka $A[p \leftarrow 1] = \neg 1 \vee q \wedge \neg r \equiv q \wedge \neg r$; $A[p \leftarrow 0] = \neg 0 \vee q \wedge \neg r \equiv 1$. Jadi $convertINF(A) = ite(p, convertINF(q \wedge \neg r), 1)$. $convertINF(q \wedge \neg r) = ite(q, convertINF(\neg r), 0)$, dan $convertINF(\neg r) = ite(r, 0, 1)$. Menggabungkan semuanya, kita memiliki $convertINF(A) = ite(p, ite(q, ite(r, 0, 1), 0), 1)$.



Gambar 2.2 (a) BDD dari $ite(p, ite(q, ite(r, 0, 1), 0), 1)$ dari contoh terakhir; (b) BDD dari INF $ite(x_1, ite(x_2, 1, 0), ite(x_2, ite(x_3, 1, 0), ite(x_3, 0, 1)))$ yang diturunkan dari $\overline{x_1 x_2 x_3} \vee x_1 x_2 \vee x_2 x_3$

Diagram keputusan biner (BDD) adalah grafik asiklik terarah (DAG) G , yang merepresentasikan INF dengan memaksimalkan pembagian subformula umum. G memiliki

tepat dua simpul daun (tanpa tepi keluar) dengan label 1 dan 0, dan satu set simpul internal (dengan tepi keluar); setiap simpul internal merepresentasikan INF $ite(p, A, B)$, yaitu, simpul tersebut diberi label p dan memiliki dua tepi keluar ke simpul yang merepresentasikan A dan B . Gambar 2.2 memberikan dua contoh BDD.

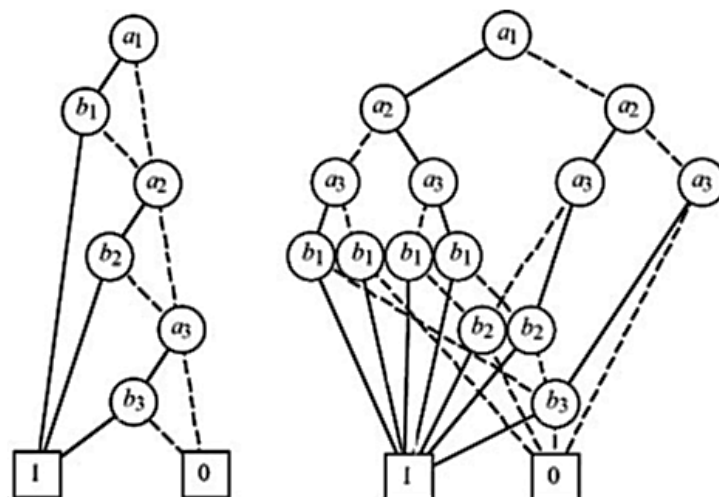
Diberikan urutan pada variabel proposisional, jika kita menggunakan urutan ini untuk memilih setiap variabel dalam prosedur $convertINF$, INF yang dihasilkan adalah BDD terurut (OBDD). Untuk contoh di atas, urutan $p > q > r$ digunakan. Jika kita menggunakan $r > q > p$, maka untuk $A = \neg p \vee q \wedge \neg r$, $convertINF(A) = ite(r, ite(p, 0, 1), ite(q, 1, ite(p, 1, 0)))$.

Pilihan orde memiliki dampak besar pada ukuran OBDD, karena satu orde memberikan ukuran linier (dalam hal jumlah variabel); orde lain dapat memberikan ukuran eksponensial. Contoh pada Gambar 2.3 menunjukkan dua OBDD untuk $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$ menggunakan dua orde. Jika kita menggeneralisasi contoh ini ke $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \dots \vee (a_n \wedge b_n)$, ukuran OBDD pertama akan menjadi $O(n)$, sedangkan OBDD kedua akan memiliki ukuran eksponensial.

Karena $ite(C, A, A) \equiv A$, kita dapat menggunakan kesetaraan ini berulang kali untuk menyederhanakan INF, hingga tidak ada lagi penyederhanaan yang dapat dilakukan. INF yang dihasilkan disebut tereduksi. BDD yang sesuai disebut tereduksi jika, selain itu, subformula identik dari sebuah INF diwakili oleh sebuah node unik.

Proposisi 2.3.22 Semua INF yang setara, menggunakan urutan yang sama pada variabel proposisional, memiliki OBDD tereduksi (ROBDD) yang unik.

Bukti Karena ROBDD adalah grafik yang mewakili INF A , kita dapat melakukan pembuktian induksi pada jumlah variabel dalam A . Jika A tidak mengandung variabel, maka 1 dan 0 diwakili oleh node unik. Misalkan $p_n > p_{n-1} > \dots > p_2 > p_1$ adalah urutan pada n variabel dalam A . Maka $A \equiv ite(p_n, B, C)$ untuk INF B, C , dan $B \not\equiv C$. Jika $n = 1$, maka $\{B, C\} = \{0, 1\}$, dan A direpresentasikan oleh sebuah node unik. Jika $n > 1$, karena B dan C adalah INF tanpa variabel p_n , berdasarkan hipotesis induksi, B dan C direpresentasikan oleh ROBDD unik, masing-masing. Dengan demikian, $A = ite(p_n, B, C)$ memiliki representasi ROBDD yang unik.



Gambar 2.3 BDD pertama menggunakan $a_1 > b_1 > a_2 > b_2 > a_3 > b_3$ dan BDD kedua menggunakan $a_1 > a_2 > a_3 > b_1 > b_2 > b_3$ untuk rumus yang sama $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$

Sangat mudah untuk menerapkan $ite(p, A, A) \equiv A$ sebagai aturan reduksi pada INF. Untuk mengimplementasikan pembagian subformula umum, kita dapat menggunakan tabel hash yang berisi triple (v, x, y) untuk setiap node $ite(v, x, y)$ dalam ROBDD, di mana v adalah variabel proposisional dan x dan $y, x \neq y$, adalah node dalam ROBDD. Tabel hash menyediakan dua operasi: *lookupHashTable* (v, x, y) memeriksa apakah (v, x, y) ada dalam tabel hash: Jika ya, node dikembalikan; jika tidak, *null* dikembalikan. Operasi lainnya adalah *saveCreateNode* (v, x, y) , yang membuat node baru untuk triple (v, x, y) dan menyimpan node tersebut dalam tabel hash. Kode semu algoritma untuk membuat ROBDD diberikan di bawah ini.

Algoritma 2.3.23 Algoritma ROBDD mengambil formula A dan mengembalikan node ROBDD untuk A . *vars* (A) mengembalikan himpunan variabel proposisional dari A ; *topVariable* (S) mengembalikan variabel maksimal dari S berdasarkan urutan variabel yang diberikan. *simplify* (A) mengembalikan nilai kebenaran dari formula tanpa variabel.

```

proc ROBDD( $A$ )
  if  $vars(A) = \emptyset$  return simplify $(A)$ 
   $v := topVariable(vars(A))$ 
   $x := ROBDD(A[v \leftarrow 1])$ 
   $y := ROBDD(A[v \leftarrow 0])$ 
  if  $(x = y)$  return  $x$  // reduksi
   $p := lookupHashTable(v, x, y)$ 
  if  $(p \neq null)$  return  $p$  // berbagi
  return saveCreateNode $(v, x, y)$  // node baru dibuat

```

Input A ke ROBDD dapat berisi node-node dari sebuah ROBDD, sehingga kita dapat melakukan berbagai operasi logika pada ROBDD.

Contoh 2.3.24 Misalkan $a > b > c$, dan misalkan A adalah $ite(F, G, H)$, di mana $F = ite(a, 1, B)$, $B = ite(b, 1, 0)$, $G = ite(a, C, 0)$, $C = ite(c, 1, 0)$, $H = ite(b, 1, D)$, dan $D = ite(d, 1, 0)$. *ROBDD* (A) akan memanggil *ROBDD* $(A[a \leftarrow 1])$ dan *ROBDD* $(A[a \leftarrow 0])$, yang masing-masing mengembalikan C dan $J = ite(b, 0, H)$. Akhirnya, *ROBDD* (A) mengembalikan $ite(a, C, J)$.

ROBDD telah digunakan untuk menyajikan fungsi Boolean, simulasi simbolik rangkaian kombinasi, pemeriksaan kesetaraan dan verifikasi fungsi Boolean, serta pencarian dan penghitungan model formula proposisional.

2.4 MASALAH OPTIMASI

Terdapat banyak masalah optimasi dalam logika proposisional. Misalnya, salah satunya adalah menemukan urutan variabel untuk suatu formula, sehingga ROBDD yang dihasilkan minimal. Pada bagian ini, kita akan memperkenalkan tiga masalah optimasi.

Himpunan Operator Minimum

Untuk merepresentasikan kalimat seperti A diberikan B , kita tidak memerlukan formula seperti $A \Leftarrow B$, karena kita dapat menggunakan $B \rightarrow A$ saja. Alasan ini dan alasan serupa menjelaskan mengapa hanya beberapa dari enam belas fungsi Boolean biner yang memerlukan notasi. Dalam CNF atau DNF, kita hanya membutuhkan tiga operator Boolean: $\neg$, $\vee$ dan $\wedge$; semua operator Boolean lainnya dapat didefinisikan berdasarkan operator-operator tersebut.

Definisi 2.4.1 Suatu himpunan S operator Boolean dikatakan cukup jika setiap operator Boolean lainnya dapat didefinisikan hanya menggunakan operator dalam S . S dikatakan cukup minimal jika S cukup dan tidak ada himpunan bagian yang tepat dari S yang merupakan himpunan operator yang cukup.

Jelas, $\{\neg, \vee, \wedge\}$ adalah himpunan operator yang cukup. Misalnya, $A \oplus B \equiv (A \vee \neg B) \wedge (\neg A \vee B)$ dan $ite(A, B, C) \equiv (A \wedge B) \vee (\neg A \wedge C)$. Apakah himpunan $\{\neg, \vee, \wedge\}$ cukup minimal? Jawabannya adalah tidak, karena $A \vee B \equiv \neg(\neg A \wedge \neg B)$. Jadi, $\{\neg, \wedge\}$ adalah himpunan operator yang cukup. $\{\neg, \vee\}$ adalah himpunan cukup lainnya, karena $A \wedge B \equiv \neg(\neg A \vee \neg B)$. Untuk menunjukkan bahwa $\{\neg, \wedge\}$ cukup minimal, kita perlu menunjukkan bahwa baik $\{\neg\}$ maupun $\{\wedge\}$ tidak cukup. Karena $\neg$ hanya mengambil satu argumen dan tidak dapat digunakan sendiri untuk mendefinisikan $\wedge$, $\{\neg\}$ tidak dapat cukup. Kita akan melihat nanti bahwa $\{\wedge\}$ tidak cukup.

Dari INF (bentuk normal ITE), kita mungkin berpikir $\{ite\}$ cukup minimal. Itu tidak benar, karena kita juga menggunakan 1 dan 0 (melambangkan T dan $\perp$) yang merupakan operator nullary. Untuk himpunan $\{\neg, \wedge\}$, $\perp \equiv p \wedge \neg p$ untuk variabel proposisional p apa pun.

Apakah kita memiliki himpunan operator Boolean yang cukup minimal yang mengandung satu operator tunggal? Jawabannya adalah ya, dan ada dua operator biner sebagai kandidatnya. Operator tersebut adalah $\uparrow$ (*nand*, negasi dari *and*) dan $\downarrow$ (*nor*, negasi dari *or*):

$$A \uparrow B \equiv \neg(A \wedge B) \quad A \downarrow B \equiv \neg(A \vee B)$$

Proposisi 2.4.2 Baik $\{\uparrow\}$ maupun $\{\downarrow\}$ adalah operator yang cukup minimal.

Bukti Untuk menunjukkan bahwa $\{\uparrow\}$ cukup, kita hanya perlu mendefinisikan $\neg$ dan $\wedge$ menggunakan $\uparrow$: $\neg(A) \equiv A \uparrow A$ dan $A \wedge B \equiv \neg(A \uparrow B)$. Bukti bahwa $\{\downarrow\}$ cukup diserahkan sebagai latihan.

Faktanya, di antara semua 16 operator biner, hanya $\uparrow$ dan $\downarrow$ yang memiliki sifat tersebut.

Teorema 2.4.3 Untuk setiap operator biner yang dapat digunakan sendiri untuk mendefinisikan baik $\neg$ maupun $\wedge$, operator tersebut harus berupa $\uparrow$ atau $\downarrow$.

Bukti Misalkan o adalah operator biner apa pun. Jika $\neg$ dapat didefinisikan oleh o , maka $\neg A \equiv t(o, A)$, di mana $t(o, A)$ adalah rumus yang diwakili oleh pohon biner t yang simpul daunnya diberi label A dan simpul internalnya diberi label o . Lebih lanjut, kita mengasumsikan bahwa t adalah pohon biner dengan jumlah simpul terkecil. Jika kita mengganti A dengan 1, maka $t(o, 1)$ seharusnya memiliki nilai kebenaran 0, karena $\neg 1 = t(o, 1) = 0$.

Jika ada simpul internal dari $t(o, 1)$ yang memiliki nilai kebenaran 1, kita dapat mengganti simpul tersebut dengan simpul daun 1; jika ada simpul internal selain akar t yang memiliki nilai kebenaran 0, kita dapat menggunakan subpohon pada simpul internal ini sebagai t . Dalam kedua kasus tersebut, ini merupakan pelanggaran karena kita mengasumsikan bahwa t adalah pohon dengan jumlah simpul terkecil. Akibatnya, t seharusnya hanya memiliki satu node internal, yaitu $1 o 1 = \neg 1 = 0$. Demikian pula, kita seharusnya memiliki $0 o 0 = \neg 0 = 1$.

Ada empat kasus mengenai nilai kebenaran dari $1 o 0$ dan $0 o 1$.

- Kasus 1: $1 o 0 = 0$ dan $0 o 1 = 1$. Maka $A o B = \neg A$ dan kita tidak dapat menggunakan o ini untuk mendefinisikan $A \wedge B$, karena output dari o pada A dan B akan menjadi salah satu dari $A, \neg A, B$, atau $\neg B$.
- Kasus 2: $1 o 0 = 1$ dan $0 o 1 = 0$. Maka $A o B = \neg B$, dan kita masih tidak dapat menggunakan o ini untuk mendefinisikan $A \wedge B$ karena alasan yang sama seperti pada kasus 1.
- Kasus 3: $1 o 0 = 1$ dan $0 o 1 = 1$. o adalah $\uparrow$.
- Kasus 4: $1 o 0 = 0$ dan $0 o 1 = 0$. o adalah $\downarrow$.

Dua kasus terakhir memberi kita definisi $\uparrow$ dan $\downarrow$, seperti yang ditunjukkan pada Tabel 2.4.

Tabel 2.4 $p o_1 q \equiv \neg p$ (negasi proyeksi pada argumen pertama); $p o_2 q \equiv \neg q$ (negasi proyeksi pada argumen kedua); $p o_3 q \equiv p \uparrow q$ (NAND); dan $p o_4 q \equiv p \downarrow q$ (NOR)

p	q	$p o_1 q$	$p o_2 q$	$p o_3 q$	$p o_4 q$
1	1	0	0	0	0
0	1	1	0	1	0
1	0	0	1	1	0
0	0	1	1	1	1

Akibat 2.4.4 Baik $\{\neg, \wedge\}$ dan $\{\neg, \vee\}$ adalah himpunan operator yang cukup minimal.

Dalam aljabar Boolean, sifat dari setiap himpunan operator yang cukup dapat ditentukan menggunakan persamaan dengan variabel dan operator. Untuk menentukan sifat $\uparrow$, kita dapat menggunakan tiga persamaan: $0 \uparrow Y = 1$, $X \uparrow 0 = 1$, dan $1 \uparrow 1 = 0$. Untuk $\uparrow$, berapa jumlah persamaan minimal yang dibutuhkan? Dan berapa jumlah simbol minimal dalam persamaan tersebut? Pertanyaan-pertanyaan ini telah dijawab dengan tegas: Hanya satu persamaan, tiga variabel (dengan total delapan kemunculan), dan enam salinan $\uparrow$ yang dibutuhkan. Kami nyatakan hasilnya di bawah ini tanpa bukti.

Proposisi 2.4.5 Aljabar Boolean dapat dihasilkan oleh persamaan berikut:

$$((x \uparrow y) \uparrow z) \uparrow (x \uparrow ((x \uparrow z) \uparrow x)) = z$$

Persamaan di atas mengandung 3 variabel dan 14 simbol (tidak termasuk tanda kurung). Untuk himpunan $\{\neg, \vee\}$, terdapat juga satu aksioma tunggal untuk menentukan aljabar Boolean:

Proposisi 2.4.6 Aljabar Boolean dapat dihasilkan oleh persamaan berikut:

$$\neg \left(\neg (\neg (x \vee y) \vee z) \vee \neg (x \vee \neg (\neg z \vee \neg (z \vee u))) \right) = z$$

Persamaan di atas berisi 4 variabel dan 21 simbol (tidak termasuk tanda kurung).

Minimisasi Logika

Minimisasi logika, juga disebut optimasi logika, adalah bagian dari proses desain sirkuit dan tujuannya adalah untuk mendapatkan sirkuit kombinasi terkecil yang diwakili oleh rumus Boolean. Minimisasi logika berupaya menemukan representasi ekuivalen dari sirkuit logika yang ditentukan di bawah satu atau lebih batasan yang ditentukan. Umumnya, sirkuit dibatasi pada area chip minimum yang memenuhi penundaan yang telah ditentukan sebelumnya. Mengurangi jumlah gerbang akan mengurangi konsumsi daya sirkuit. Memilih gerbang dengan lebih sedikit transistor akan mengurangi area sirkuit. Mengurangi jumlah level gerbang bersarang akan mengurangi penundaan sirkuit. Minimisasi logika akan secara substansial mengurangi biaya sirkuit dan meningkatkan kualitasnya.

Peta Karnaugh untuk Meminimalkan DNF dan CNF

Pada masa awal otomatisasi desain elektronik (EDA), fungsi Boolean sering didefinisikan oleh tabel kebenaran dan kita dapat menurunkan DNF (jumlah produk) atau CNF (produk jumlah) lengkap langsung dari tabel kebenaran, seperti yang ditunjukkan pada Contoh 2.3.16. DNF atau CNF yang diperoleh perlu diminimalkan, agar dapat menggunakan jumlah gerbang yang paling sedikit untuk mengimplementasikan fungsi ini.

Sesuai dengan konvensi EDA, kita akan menggunakan $+$ untuk $\vee$, $\cdot$ untuk $\wedge$, dan $\bar{A}$ untuk $\neg A$.

Contoh 2.4.7 Misalkan $f: \{0, 1\}^2 \rightarrow \{0, 1\}$ didefinisikan oleh $f(A, B) = \bar{A}\bar{B} + A\bar{B} + \bar{A}B$. Kita membutuhkan tiga gerbang AND dan satu gerbang OR dengan 3 input.

Namun, $f(A, B) \equiv A + B$. Dengan menggunakan $A + B$, kita hanya membutuhkan satu gerbang OR.

Relasi kesetaraan yang digunakan dalam proses penyederhanaan di atas adalah $AB + A\bar{B} \equiv A$ dan $A + \bar{A}B \equiv A + B$ (atau $\bar{A} + AB \equiv \bar{A} + B$). Untuk contoh di atas, $\bar{A}\bar{B} + A\bar{B} \equiv \bar{B}$ dan $\bar{B} + \bar{A}B \equiv \bar{B} + \bar{A}$.

Kita ingin menemukan rangkaian ekuivalen dengan ukuran sekecil mungkin, dan ini adalah masalah komputasi yang sulit. Ada banyak alat seperti peta Karnaugh yang tersedia untuk mencapai tujuan ini.

Peta Karnaugh (Kmap) adalah metode populer untuk menyederhanakan rumus DNF atau CNF, yang awalnya diusulkan oleh Maurice Karnaugh pada tahun 1953. Peta Karnaugh mengurangi kebutuhan akan perhitungan yang ekstensif dengan menampilkan keluaran fungsi secara grafis dan memanfaatkan kemampuan pengenalan pola manusia. Metode ini cocok untuk fungsi Boolean dengan dua hingga empat variabel. Jika jumlah variabel lebih dari 4, lebih baik menggunakan alat otomatis.

Peta Karnaugh menggunakan grid dua dimensi di mana setiap sel mewakili interpretasi (atau setara dengan, produk lengkap) dan sel tersebut berisi nilai kebenaran fungsi untuk interpretasi tersebut. Posisi setiap sel berisi semua informasi dari suatu interpretasi, dan sel-sel tersebut disusun sedemikian rupa sehingga sel-sel yang berdekatan berbeda tepat satu nilai kebenaran dalam interpretasi tersebut. Angka satu yang berdekatan dalam peta Karnaugh mewakili peluang untuk menyederhanakan rumus.

Produk untuk rumus akhir ditemukan dengan melingkari kelompok angka satu dalam peta. Grup produk harus berbentuk persegi panjang dan memiliki luas yang merupakan pangkat dua (yaitu, 1, 2, 4, 8, ...). Persegi panjang produk harus sebesar mungkin tanpa mengandung angka nol (dapat mengandung sel yang tidak penting). Grup dapat memiliki angka satu yang tumpang tindih. Jumlah grup terkecil yang mencakup semua angka satu akan mewakili produk dari DNF fungsi tersebut. Produk-produk ini dapat digunakan untuk menulis DNF minimal yang mewakili fungsi yang dibutuhkan dan dengan demikian diimplementasikan oleh jumlah gerbang AND terkecil yang masuk ke gerbang OR. Peta ini juga dapat digunakan untuk mendapatkan CNF minimal yang diimplementasikan oleh gerbang OR yang masuk ke gerbang AND.

Misalkan $f(x, y) = x\bar{y} + \bar{x}y + \bar{x}\bar{y}$. Ada empat produk x dan y , dan peta K memiliki empat sel, yaitu, empat produk yang sesuai dengan empat interpretasi pada x dan y . Sebagai output dari f , tiga sel memiliki nilai 1 dan satu sel (yaitu, xy) memiliki nilai 0. Awalnya, setiap sel termasuk dalam kelompok yang berbeda yang terdiri dari satu anggota (Gambar 2.4).

x	y	f
0	0	1
0	1	1
1	0	1
1	1	0

	y	0	1
x	0	1	1
	1	1	0

Gambar 2.4 Peta K untuk $f(x, y) = x\bar{y} + \bar{x}y + \bar{x}\bar{y}$

Operasi penggabungan pada peta K: Menggabungkan dua kelompok yang berdekatan dan terpisah dengan ukuran dan nilai kebenaran yang sama menjadi satu kelompok yang lebih besar.

Operasi ini menciptakan kelompok dengan ukuran 2, 4, 8, dst., dan sel diperbolehkan muncul di kelompok yang berbeda secara bersamaan. Misalnya, sel $x\bar{y}$ dan $\bar{x}\bar{y}$ pada Gambar 2.4 berdekatan; keduanya digabungkan menjadi satu kelompok $\{x\bar{y}, \bar{x}\bar{y}\}$, yang dapat direpresentasikan oleh $\bar{y}$ sebagai singkatan dari kelompok tersebut. Demikian pula, sel $x\bar{y}$ dan $\bar{x}y$ dapat digabungkan menjadi $\{x\bar{y}, \bar{x}y\}$, yang dapat direpresentasikan oleh $\bar{x}$. Perhatikan bahwa $\bar{x}\bar{y}$ digunakan dua kali dalam dua operasi penggabungan. Sekarang tidak ada lagi operasi penggabungan yang dapat dilakukan, dan hasil akhirnya adalah $f = \bar{x} + \bar{y}$ (f adalah fungsi NAND). Hasil ini dapat dibuktikan secara logis sebagai berikut:

$$\begin{aligned} f &= x\bar{y} + \bar{x}y + \bar{x}\bar{y} \\ &= (x\bar{y} + \bar{x}\bar{y}) + (\bar{x}y + \bar{x}\bar{y}) \\ &= (x + \bar{x})\bar{y} + \bar{x}(y + \bar{y}) \\ &= \bar{y} + \bar{x} \end{aligned}$$

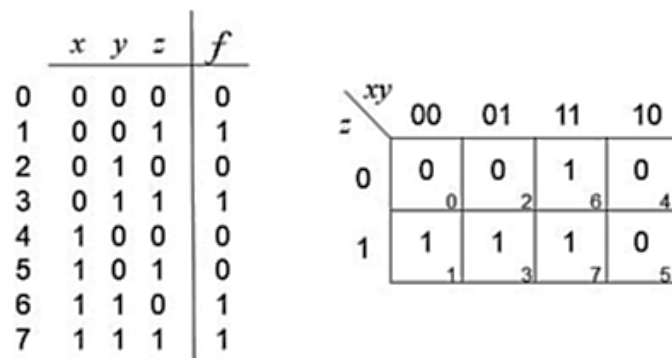
Contoh 2.4.8 Perhatikan fungsi f yang didefinisikan oleh tabel kebenaran pada Gambar 2.5. Peta K yang sesuai memiliki empat sel dengan nilai kebenaran 1 karena f memiliki empat model yang ditunjukkan dalam tabel kebenaran oleh baris bernomor 1, 3, 6, dan 7. Ada tiga kemungkinan operasi penggabungan: (i) $\bar{x}\bar{y}z$ dan $\bar{x}yz$ bergabung menjadi $\bar{x}z$; (ii) $xy\bar{z}$ dan xyz bergabung menjadi xy ; dan (iii) $\bar{x}yz$ dan xyz bergabung menjadi yz . Karena semua sel dengan nilai 1 berada dalam kelompok yang diwakili oleh $\bar{x}z$ dan xy , maka yz bersifat redundan. Jadi, fungsi yang disederhanakan adalah $f = \bar{x}y + xz$, bukan $f = xy + \bar{x}z + yz$. Di sisi lain, jika $g = xy + xz + yz$, maka tidak ada satu pun dari ketiga produk dalam g yang bersifat redundan, karena jika Anda menghapus satu, Anda akan kehilangan beberapa, meskipun produk lengkap xyz dicakup oleh ketiga produk tersebut.

Dari contoh-contoh di atas, jelas bahwa setelah setiap operasi penggabungan, dua produk digabungkan menjadi satu produk yang berisi satu variabel lebih sedikit. Artinya, semakin besar grup, semakin sedikit jumlah variabel dalam produk untuk mewakili grup tersebut. Hal ini dimungkinkan karena sel-sel yang berdekatan mewakili interpretasi yang berbeda dengan nilai kebenaran satu variabel.

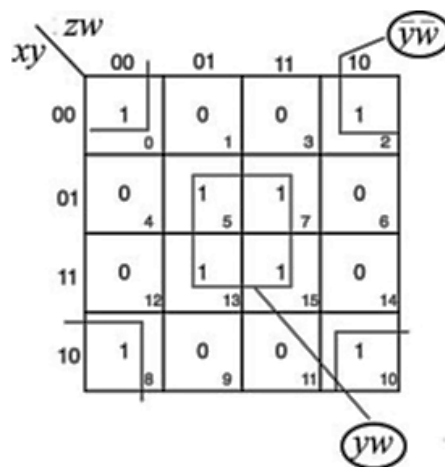
Karena K-map bersifat dua dimensi, beberapa hubungan berdekatan antar sel tidak ditampilkan pada K-map. Misalnya, pada Gambar 2.5, $m_0 = \bar{x}\bar{y}\bar{z}$ dan $m_4 = x\bar{y}\bar{z}$ berdekatan, dan $m_1 = \bar{x}\bar{y}z$ dan $m_5 = x\bar{y}z$ berdekatan. Secara tegas, K-map terhubung secara toroidal, yang berarti bahwa grup persegi panjang dapat membungkus tepi. Sel-sel di paling kanan sebenarnya "berdekatan" dengan sel-sel di paling kiri, dalam arti bahwa interpretasi yang sesuai hanya berbeda satu nilai kebenaran. Kita membutuhkan grafik tiga dimensi untuk

sepenuhnya menunjukkan hubungan berdekatan dari tiga variabel, grafik empat dimensi untuk sepenuhnya menunjukkan hubungan berdekatan dari empat variabel, dan seterusnya.

Berdasarkan ide K-map, alat otomatis akan membuat grafik dengan 2^n node untuk n variabel, setiap node mewakili satu interpretasi, dan dua node memiliki sisi jika interpretasi mereka berbeda satu nilai kebenaran. Operasi penggabungan kemudian diimplementasikan pada grafik ini untuk menghasilkan produk untuk DNF ukuran terkecil. Grafik yang sama juga digunakan untuk menghasilkan kode Gray (dengan menemukan jalur Hamiltonian dalam grafik).



Gambar 2.5 K-map untuk mendapatkan $f(x, y, z) = xy + \bar{x}z$



Gambar 2.6 Peta K untuk $f(x, y, z, w) = m_0 + m_2 + m_5 + m_7 + m_8 + m_{10} + m_{13} + m_{15}$.
Rumus yang disederhanakan adalah $f = \bar{y}w + yw$

Contoh 2.4.9 Peta K untuk $f(x, y, z, w) = m_0 + m_2 + m_5 + m_7 + m_8 + m_{10} + m_{13} + m_{15}$, diberikan pada Gambar 2.6, di mana $m_0 = \bar{x}\bar{y}\bar{z}\bar{w}$, $m_2 = \bar{x}\bar{y}z\bar{w}$, $m_5 = \bar{x}y\bar{z}\bar{w}$, $m_7 = \bar{x}yz\bar{w}$, $m_8 = x\bar{y}\bar{z}\bar{w}$, $m_{10} = x\bar{y}z\bar{w}$, $m_{13} = xy\bar{z}\bar{w}$, dan $m_{15} = xyz\bar{w}$ (i dalam m_i adalah nilai desimal dari $xyzw$ sebagai bilangan biner). Pertama, m_0 dan m_2 bersebelahan dan dapat digabungkan menjadi $\bar{x}\bar{y}\bar{w}$; m_8 dan m_{10} juga bersebelahan dan dapat digabungkan menjadi $x\bar{y}\bar{w}$. Kemudian $\bar{x}\bar{y}\bar{w}$ dan $x\bar{y}\bar{w}$ dapat digabungkan lebih lanjut menjadi $\bar{y}\bar{w}$. Demikian pula, m_5 dan m_7 bergabung menjadi $\bar{x}y\bar{w}$; m_{13} dan m_{15}

bergabung menjadi xyw . Kemudian $\bar{x}yw$ dan xyw bergabung menjadi yw . Hasil akhirnya adalah $f = \bar{y}\bar{w} + yw$. Penyederhanaan ini juga dapat diperoleh dengan menerapkan relasi ekuivalensi secara hati-hati, tetapi waktu yang dibutuhkan untuk melakukannya meningkat secara eksponensial dengan jumlah produk.

Perhatikan bahwa rumus yang diperoleh dari peta K pada umumnya tidak unik. Misalnya, $g(x, y, z, w) = x\bar{y} + z\bar{w} + \bar{x}y\bar{w}$ atau $g(x, y, z, w) = x\bar{y} + z\bar{w} + \bar{y}z\bar{w}$ adalah kemungkinan keluaran untuk g dari peta K. Ini berarti keluaran peta K tidak kanonik.

Peta K dapat digunakan untuk menghasilkan CNF yang disederhanakan untuk fungsi Boolean. Dalam Contoh 2.3.16, kita telah menunjukkan cara memperoleh CNF lengkap dari DNF lengkap. Dengan menggunakan ide yang sama, jika kita menyederhanakan DNF lengkap terlebih dahulu, CNF yang diperoleh dengan meniadakan DNF yang disederhanakan akan menjadi CNF yang disederhanakan.

$x \backslash yz$	00	01	11	10
0	0 ₀	1 ₁	0 ₃	1 ₂
1	1 ₄	1 ₅	0 ₇	0 ₆

Gambar 2.7 Peta K untuk $f(x, y, z) = \bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}\bar{z} + x\bar{y}z$. Bentuk CNF yang disederhanakan adalah $f = (x + y + z)(\bar{x} + \bar{y})(\bar{y} + \bar{z})$

Contoh 2.4.10 Misalkan $f(x, y, z) = \bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}\bar{z} + x\bar{y}z$. dan peta K-nya diberikan pada Gambar 2.7. Negasi dari f adalah

$$\bar{f} = \bar{x}\bar{y}\bar{z} + \bar{x}yz + x\bar{y}z + xyz.$$

Karena $\bar{x}yz$ dan xyz dapat bergabung menjadi yz (seperti yang ditunjukkan pada gambar), $\bar{x}\bar{y}\bar{z}$ dan $x\bar{y}\bar{z}$ dapat bergabung menjadi $\bar{x}\bar{y}$, maka $\bar{f}$ yang disederhanakan adalah $\bar{f} = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y} + yz$. CNF yang disederhanakan untuk f adalah $f = (x + y + z)(\bar{x} + \bar{y})(\bar{y} + \bar{z})$.

Peta Karnaugh juga dapat digunakan untuk menangani nilai "tidak peduli" dan dapat menyederhanakan ekspresi logika dalam desain perangkat lunak. Kondisi Boolean, seperti yang digunakan, misalnya, dalam pernyataan kondisional, dapat menjadi sangat rumit, yang membuat kode sulit dibaca dan dipelihara. Setelah diminimalkan, ekspresi jumlah-hasil-perkalian dan hasil-hasil-perkalian dapat diimplementasikan langsung menggunakan operator logika AND dan OR.

Diskusi di atas mengilustrasikan ide dasar optimasi sirkuit. Presentasi metode optimasi praktis yang digunakan untuk chip VLSI (integrasi skala sangat besar) menggunakan teknologi CMOS (complementary metal–oxide–semiconductor) berada di luar cakupan buku ini.

Kepuasan Maksimum

Ketika suatu CNF tidak dapat dipenuhi, kita masih tertarik untuk menemukan interpretasi yang membuat sebagian besar klausa bernilai benar. Ini disebut masalah kepuasan maksimum (Max-SAT) di mana kita menentukan jumlah maksimum klausa yang bernilai benar dari CNF yang diberikan di bawah interpretasi apa pun. Ini adalah generalisasi dari masalah kepuasan Boolean, yang menanyakan apakah ada penugasan nilai benar yang membuat semua klausa bernilai benar.

Contoh 2.4.11 Misalkan A adalah himpunan klausa yang tidak dapat dipenuhi $\{(\bar{p}), (p \vee \bar{q}), (p \vee q), (p \vee \bar{r}), (p \vee r)\}$. Jika kita menetapkan 0 untuk p , dua klausa akan bernilai salah terlepas dari nilai apa pun yang diberikan kepada q dan r . Di sisi lain, jika p diberi nilai 1, hanya klausa $(\bar{p})$ yang akan bernilai salah. Oleh karena itu, jika A diberikan sebagai contoh masalah MAX-SAT, solusi untuk masalah tersebut adalah 4 (empat klausa bernilai benar dalam satu interpretasi).

Secara lebih umum, versi berbobot dari MAX-SAT dapat didefinisikan sebagai berikut: diberikan rumus bentuk normal konjungtif dengan bobot non-negatif yang diberikan kepada setiap klausa, temukan nilai kebenaran untuk variabelnya yang memaksimalkan bobot gabungan dari klausa yang terpenuhi. Masalah MAX-SAT adalah contoh dari MAX-SAT berbobot di mana semua bobot adalah 1.

Menemukan jadwal ujian akhir untuk universitas besar adalah pekerjaan yang menantang. Ada kendala keras seperti tidak ada mahasiswa yang dapat mengikuti dua ujian pada waktu yang bersamaan, dan tidak ada ruang kelas yang dapat menyelenggarakan dua ujian pada waktu yang bersamaan. Ada juga kendala lunak seperti satu mahasiswa lebih disukai untuk mengikuti paling banyak dua ujian dalam 1 hari, atau dua ujian dipisahkan setidaknya 2 jam. Untuk menentukan masalah seperti itu sebagai contoh dari MAX-SAT berbobot itu mudah: Kita hanya memberikan bobot yang sangat besar pada kendala keras dan bobot kecil pada kendala lunak.

Berbagai metode telah dikembangkan untuk menyelesaikan masalah MAX-SAT atau MAX-SAT berbobot. Metode-metode ini terkait dengan penyelesaian masalah SAT dan akan kami sajikan nanti dalam buku ini.

2.5 MENGGUNAKAN LOGIKA PROPOSISIONAL

Proposisi dan operator logika muncul sepanjang waktu dalam program komputer. Semua bahasa pemrograman seperti C, C++, dan Java menggunakan simbol seperti “&&”, “||”, dan “!” sebagai pengganti $\wedge$, $\vee$, dan $\neg$. Perhatikan ekspresi Boolean yang muncul dalam pernyataan *if*:

$$\text{if}(x > 0 \parallel (x \leq 0 \ \&\& \ y > 100)) \dots$$

Jika kita misalkan p menyatakan $x > 0$ dan q menyatakan $y > 100$, maka ekspresi Boolean tersebut diabstraksikan sebagai $p \vee \neg p \wedge q$. Karena $(p \vee \neg p \wedge q) \equiv (p \vee q)$, kode asli dapat ditulis ulang sebagai berikut:

$$\text{if } (x > 0 \parallel y > 100) \dots$$

Dengan kata lain, pengetahuan tentang logika dapat membantu kita menulis kode yang lebih rapi. Perhatikan potongan kode lain:

$$\text{if } (x \geq 0 \&\& A[x] == 0) \dots$$

Misalkan p adalah $x \geq 0$ dan q adalah $A[x] == 0$, maka ekspresi Boolean adalah $p \wedge q$. Karena $p \wedge q \equiv q \wedge p$, wajar untuk berpikir bahwa $(x \geq 0 \&\& A[x] == 0)$ dapat digantikan oleh $(A[x] == 0 \&\& x \geq 0)$ dan programnya akan sama. Sayangnya, ini tidak demikian: Eksekusi $(A[x] == 0 \&\& x \geq 0)$ akan dihentikan ketika $x < 0$. Contoh ini menunjukkan bahwa "&&" dalam bahasa pemrograman pada umumnya tidak komutatif. Begitu pula dengan "||" karena $(x < 0 \parallel A[x] == 0)$ berbeda dengan $(A[x] == 0 \parallel x < 0)$. Kejanggalaan logika ini tidak hanya muncul pada indeks array tetapi juga ketika penugasan diizinkan dalam ekspresi Boolean. Misalnya, $x > 0 \parallel (x = c) > 10$ berbeda dengan $(x = c) > 10 \parallel x > 0$ dalam C, di mana $x = c$ adalah perintah penugasan. Sebagai seorang programmer, kita perlu berhati-hati terhadap kejanggalaan ini.

Operator Bitwise

Dalam bahasa pemrograman seperti C, C++, dan Java, bilangan bulat direpresentasikan oleh daftar 16, 32, atau 64 bit (untuk C, char dianggap sebagai bilangan bulat 8-bit atau byte). Operasi bitwise beroperasi pada satu atau dua bilangan bulat pada tingkat bit individualnya dan beberapa di antaranya adalah operasi logika pada bit. Ini adalah tindakan yang cepat dan sederhana, didukung langsung oleh prosesor, dan harus digunakan kapan pun memungkinkan.

Untuk tujuan ilustrasi, kita akan menggunakan bilangan bulat 8-bit sebagai contoh. Secara umum, untuk setiap bilangan bulat 8-bit x , representasi biner dari x adalah vektor 8-bit $(x)^2 = (b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$, di mana $b_i \in \{0, 1\}$ dan $x = \sum_{i=0}^7 b_i * 2^i$. Misalnya, untuk $a = 67$, vektor bitnya adalah $(a)^2 = (0, 1, 0, 0, 0, 0, 1, 1)$, karena $67 = 64 + 2 + 1 = 2^6 + 2^1 + 2^0$, dan untuk $b = 18$, $(b)^2 = (0, 0, 0, 1, 0, 0, 1, 0)$, karena $18 = 16 + 2 = 2^4 + 2^1$. Sebenarnya, a dan b disimpan sebagai $(a)_2$ dan $(b)_2$ di komputer.

Tabel 2.5 menyediakan beberapa operator bitwise populer dengan contoh, memperlakukan setiap bit sebagai nilai Boolean dan melakukan operasi pada bit tersebut secara bersamaan. Kita akan menggunakan vektor bit $a = 67$ dan $b = 18$ sebagai contoh. Jika perlu, operasi bitwise lainnya dapat diimplementasikan menggunakan empat operasi dalam Tabel 2.5. Misalnya, implikasi bitwise pada a dan b dapat diimplementasikan sebagai $\sim a \mid b$.

Tabel 2.5 Contoh operator bitwise

$(a)_2$	=	(0, 1, 0, 0, 0, 0, 1, 1)
$(b)_2$	=	(0, 0, 0, 1, 0, 0, 1, 0)
<i>Nama</i>	<i>Simbol</i>	<i>Contoh</i>
<i>NOT</i>	$\sim$	$(\sim a)_2 = (1, 0, 1, 1, 1, 1, 0, 0)$
<i>OR</i>	$ $	$(a b)_2 = (0, 1, 0, 1, 0, 0, 1, 1)$
<i>AND</i>	$\&$	$(a \& b)_2 = (0, 0, 0, 0, 0, 0, 1, 0)$
<i>XOR</i>	$\wedge$	$(a \wedge b)_2 = (0, 1, 0, 1, 0, 0, 0, 1)$
<i>Geser Kanan</i>	$>>$	$(a >> 2)_2 = (0, 0, 0, 1, 0, 0, 0, 0)$
<i>Geser kiri</i>	$<<$	$(b << 2)_2 = (0, 1, 0, 0, 1, 0, 0, 0)$

Pergeseran bit juga dianggap sebagai operasi bitwise, karena operasi ini memperlakukan bilangan bulat sebagai vektor bit dan bukan sebagai nilai numerik. Dalam operasi ini, bit dipindahkan, atau digeser, ke kiri atau ke kanan. Register dalam prosesor komputer memiliki lebar tetap, sehingga beberapa bit akan "digeser keluar" dari register di satu ujung, sementara jumlah bit yang sama "digeser masuk" dari ujung lainnya. Biasanya, bit yang "digeser keluar" dibuang dan bit yang "digeser masuk" semuanya adalah 0. Ketika bit terdepan adalah tanda bilangan bulat, operasi pergeseran kiri menjadi lebih rumit, dan pembahasannya berada di luar cakupan buku ini. Dalam tabel, kita melihat bahwa operator pergeseran mengambil dua bilangan bulat: $x \ll y$ akan menggeser $(x)_2$ ke kiri sebanyak y bit; $x \gg y$ akan menggeser $(x)_2$ ke kanan sebanyak y bit.

Nilai vektor bit $x \ll y$ sama dengan $x * 2^y$, asalkan perkalian ini tidak menyebabkan overflow. Misalnya, untuk $b = 18$, $b \ll 2 = 18 * 4 = 72$. Demikian pula, nilai $x \gg y$ sama dengan pembagian bilangan bulat $x/2^y$. Misalnya, untuk $a = 67$, $a \gg 2 = \lfloor 67/4 \rfloor = 16$. Pergeseran memberikan cara yang efisien untuk perkalian dan pembagian ketika operan kedua adalah pangkat dua. Biasanya, operasi bitwise jauh lebih cepat daripada pembagian, beberapa kali lebih cepat daripada perkalian.

Misalkan Anda adalah manajer sebuah klub sosial dengan 64 anggota. Untuk setiap pertemuan, Anda perlu mencatat kehadiran semua anggota. Jika setiap anggota memiliki ID anggota unik $\{0, 1, \dots, 63\}$, Anda dapat menggunakan himpunan ID anggota untuk mencatat kehadiran. Alih-alih daftar bilangan bulat, Anda dapat menggunakan satu bilangan bulat 64-bit, misalnya s , untuk menyimpan himpunan tersebut. Artinya, jika seorang anggota hadir dan ID-nya adalah i , maka bit ke- i dari s adalah 1; jika tidak, bit ke- i adalah 0 untuk menunjukkan ketidakhadirannya. Untuk tujuan ini, kita membutuhkan cara untuk mengatur bit dalam s menjadi 1. Misalkan $s = 0$ pada awalnya; kita dapat menggunakan perintah berikut dalam Java atau C:

$$s = s | (1 \ll i)$$

Untuk mengatur bit ke- i dari s menjadi 1. Untuk mengatur nilai bit ke- i dari s menjadi 0, gunakan

$$s = s \& \sim(1 \ll i)$$

Untuk mendapatkan nilai bit ke- i dari s , gunakan $s \& (1 \ll i)$. Untuk pertemuan lain, kita menggunakan bilangan bulat lain t . Sekarang mudah untuk menghitung gabungan atau irisan dari himpunan-himpunan ini: $s | t$ adalah gabungan dan $s \& t$ adalah irisan. Misalnya, jika Anda ingin memeriksa bahwa tidak ada yang menghadiri kedua pertemuan tersebut, Anda dapat memeriksa apakah $s \& t = 0$, yang lebih efisien daripada menggunakan daftar untuk himpunan.

Masalah Keputusan dalam Logika Proposisional

Logika proposisional adalah bahasa yang sangat sederhana tetapi cukup kaya untuk menentukan sebagian besar masalah keputusan dalam ilmu komputer, karena masalah-masalah ini termasuk dalam kelas masalah NP dan SAT adalah NP-lengkap. Bahkan, logika proposisional adalah bahasa formal yang jauh lebih ketat daripada bahasa alami. Karena bahasa alami bersifat ambigu, mengekspresikan kalimat dalam logika membantu kita untuk memahami arti sebenarnya dari kalimat tersebut. Misalnya, dua kalimat berikut tampaknya memiliki arti yang sama.

- Mahasiswa dan lansia membayar setengah harga.
- Siswa atau lansia membayar setengah harga.

Satu kalimat menggunakan "dan" dan kalimat lainnya menggunakan "atau," seolah-olah "dan" dan "atau" adalah sinonim. Bagaimana menjelaskannya? Misalkan "menjadi siswa" dan "menjadi lansia" disingkat dengan p dan q , dan "membayar setengah harga" dengan r . Misalkan A adalah $(p \rightarrow r) \wedge (q \rightarrow r)$ dan B adalah $(p \vee q) \rightarrow r$, maka A dan B mengungkapkan isi faktual dari kedua kalimat di atas dengan lebih tepat. Mudah untuk menunjukkan bahwa rumus A dan B secara logis setara. Pernyataan bahasa sehari-hari dari A dan B mengaburkan perbedaan struktural A dan B melalui penggunaan kata "dan" dan "atau" yang tampaknya sinonim.

Kita akan menunjukkan bagaimana cara menentukan berbagai masalah dalam logika proposisional melalui contoh.

Contoh 2.5.1 Masalah yang disebut n -ratu dinyatakan sebagai berikut: Diberikan papan catur berukuran $n \times n$, di mana $n > 0$, bagaimana menempatkan n ratu di papan sedemikian rupa sehingga tidak ada dua ratu yang saling menyerang? Untuk $n = 2, 3$, tidak ada solusi; untuk n lainnya, ada solusi. Untuk menentukan masalah ini dalam logika, kita membutuhkan n^2 variabel proposisional. Misalkan variabelnya adalah $p_{i,j}$, di mana $1 \leq i, j \leq n$, dan arti dari $p_{i,j}$ adalah bahwa $p_{i,j}$ benar jika dan hanya jika ada ratu di baris i dan kolom j . Sifat-sifat berikut memastikan bahwa ada n ratu di papan sedemikian rupa sehingga tidak ada dua ratu yang saling menyerang:

1. Setidaknya ada satu ratu di setiap baris.

$$\bigwedge_{1 \leq i \leq n} (p_{i,1} \vee p_{i,2} \vee \cdots \vee p_{i,n})$$

2. Tidak lebih dari satu ratu di setiap baris.

$$\bigwedge_{1 \leq i \leq n} (\bigwedge_{1 \leq j < k \leq n} (\neg p_{i,j} \vee \neg p_{i,k}))$$

3. Tidak lebih dari satu ratu di setiap kolom.

$$\bigwedge_{1 \leq i \leq n} (\bigwedge_{1 \leq j < k \leq n} (\neg p_{j,i} \vee \neg p_{k,i}))$$

4. Tidak lebih dari satu ratu di setiap diagonal.

$$\bigwedge_{1 \leq i < k \leq n} (\bigwedge_{1 \leq j, l \leq n, k-i=|l-j|} (\neg p_{i,j} \vee \neg p_{k,l}))$$

Rumus yang mengikuti setiap properti menyatakan properti tersebut secara formal dan dapat dengan mudah diubah menjadi klausa. Ketika $n = 2, 3$, rumusnya tidak dapat dipenuhi. Untuk $n \geq 4$, rumusnya dapat dipenuhi, dan setiap model memberi kita solusi. Misalnya, ketika $n = 4$, kita memiliki dua solusi: $\{p_{1,2}, p_{2,4}, p_{3,1}, p_{4,3}\}$ dan $\{p_{1,3}, p_{2,1}, p_{3,4}, p_{4,2}\}$. Jika kita menetapkan empat variabel dalam satu solusi sebagai benar dan variabel lainnya sebagai salah, kita memperoleh sebuah model. Perhatikan bahwa kita mengabaikan properti bahwa “ada setidaknya satu ratu di setiap kolom,” karena itu berlebihan.

Contoh selanjutnya adalah teka-teki logika.

Contoh 2.5.2 Tiga orang pergi ke pantai, masing-masing menggunakan moda transportasi yang berbeda (mobil, sepeda motor, dan perahu) dengan warna yang berbeda (biru, oranye, hijau). Siapa yang menggunakan apa? Berikut petunjuk yang diberikan:

1. Abel menyukai jeruk, tetapi ia membenci perjalanan di atas air.
2. Bob tidak menggunakan kendaraan hijau.
3. Carol mengendarai mobil.

Langkah pertama untuk memecahkan teka-teki menggunakan logika proposisional adalah memastikan properti apa yang diasumsikan dan kondisi apa yang diberikan oleh petunjuk. Untuk teka-teki ini, properti pertama yang diasumsikan adalah bahwa “setiap orang menggunakan alat transportasi yang unik.”

Untuk menentukan properti ini dalam logika, ada beberapa cara untuk melakukannya. Salah satu caranya adalah dengan menggunakan serangkaian variabel proposisional untuk menentukan hubungan “siapa menggunakan transportasi apa”: Misalkan $p_{x,y}$ menunjukkan serangkaian sembilan variabel, di mana $x \in \{a, b, c\}$ (a untuk Abel, b untuk Bob, dan c untuk Carol), dan $y \in \{c, m, b\}$ (c untuk mobil, m untuk sepeda motor, dan b untuk perahu). Sekarang properti bahwa “setiap orang menggunakan alat transportasi yang unik” dapat ditentukan secara formal sebagai klausa berikut:

- (1) $(p_{a,c} \mid p_{a,m} \mid p_{a,b})$ Abel menggunakan c, m , atau b .
- (2) $(p_{b,c} \mid p_{b,m} \mid p_{b,b})$ Bob menggunakan c, m , atau b .
- (3) $(p_{c,c} \mid p_{c,m} \mid p_{c,b})$ Carol menggunakan c, m , atau b .

(4)	$(\overline{p_{a,c}} \mid \overline{p_{b,c}})$	Abel dan Bob tidak mungkin sama-sama menggunakan c .
(5)	$(\overline{p_{a,m}} \mid \overline{p_{b,m}})$	Abel dan Bob tidak mungkin sama-sama menggunakan m .
(6)	$(\overline{p_{a,b}} \mid \overline{p_{b,b}})$	Abel dan Bob tidak mungkin sama-sama menggunakan b .
(7)	$(\overline{p_{a,c}} \mid \overline{p_{c,c}})$	Abel dan Carol tidak mungkin sama-sama menggunakan c .
(8)	$(\overline{p_{a,m}} \mid \overline{p_{c,m}})$	Abel dan Carol tidak mungkin sama-sama menggunakan m .
(9)	$(\overline{p_{a,b}} \mid \overline{p_{c,b}})$	Abel dan Carol tidak mungkin sama-sama menggunakan b .
(10)	$(\overline{p_{b,c}} \mid \overline{p_{c,c}})$	Bob dan Carol tidak mungkin keduanya menggunakan c .
(11)	$(\overline{p_{b,m}} \mid \overline{p_{c,m}})$	Bob dan Carol tidak mungkin keduanya menggunakan m .
(12)	$(\overline{p_{b,b}} \mid \overline{p_{c,b}})$	Bob dan Carol tidak mungkin keduanya menggunakan b .

Tampaknya cukup rumit bahwa 12 klausa diperlukan untuk menentukan satu properti. Perhatikan bahwa tujuan penggunaan logika adalah untuk menemukan solusi secara otomatis. Bagi komputer, 12 juta klausa bukanlah jumlah yang terlalu banyak. Dari petunjuk teka-teki, kita memiliki dua klausa unit berikut:

(13)	$(p_{c,c})$	Carol yang mengemudikan mobil.
(14)	$(\overline{p_{a,b}})$	Abel membenci perjalanan di atas air.

Kedua klausa unit ini memaksa $p_{c,c}$ untuk diberi nilai 1 dan $p_{a,a}$ menjadi 0. Kemudian kita dapat menentukan nilai kebenaran dari tujuh variabel lainnya:

$p_{a,c} \rightarrow 0$	dari	$p_{c,c} \rightarrow 1, (7).$
$p_{b,c} \rightarrow 0$	dari	$p_{c,c} \rightarrow 1, (11).$
$p_{a,m} \rightarrow 1$	dari	$p_{a,c} \rightarrow 0, p_{a,b} \rightarrow 0, (1).$
$p_{b,m} \rightarrow 0$	dari	$p_{a,m} \rightarrow 1, (5).$
$p_{c,m} \rightarrow 0$	dari	$p_{a,m} \rightarrow 1, (8).$
$p_{b,b} \rightarrow 1$	dari	$p_{b,c} \rightarrow 0, p_{b,m} \rightarrow 0, (2).$
$p_{c,b} \rightarrow 0$	dari	$p_{b,b} \rightarrow 1, (12).$

Sifat kedua yang diasumsikan adalah bahwa “setiap alat transportasi memiliki warna yang unik.” Jika kita membaca teka-teki dengan saksama, petunjuknya melibatkan hubungan bahwa “siapa yang akan menggunakan warna transportasi apa.” Jadi, lebih baik menggunakan $q_{x,z}$ untuk mewakili hubungan ini bahwa orang x menggunakan warna z , di mana $x \in \{a, b, c\}$ dan $z \in \{b, o, g\}$ (b untuk biru, o untuk oranye, dan g untuk hijau).

Kita dapat menentukan sifat bahwa “setiap orang menggunakan warna yang unik” dengan cara yang sama seperti kita menentukan “setiap orang menggunakan alat transportasi yang unik.” Petunjuk memberi kita dua klausa unit: $(q_{a,o})$ dan $(\overline{q_{b,g}})$. Mulai dari dua klausa unit ini, kita dapat menurunkan nilai kebenaran dari semua sembilan variabel $q_{x,z}$. Teka-teki berikutnya terlihat sangat berbeda dari yang sebelumnya, tetapi spesifikasinya serupa.

Contoh 2.5.3 Empat orang duduk di bangku dengan empat kursi untuk foto grup: dua orang Amerika (A), seorang warga Inggris (B), dan seorang warga Kanada (C). Mereka bertanya (i) dua orang Amerika tidak ingin duduk bersebelahan; (ii) orang Inggris suka duduk bersebelahan dengan orang Kanada. Misalkan kita menggunakan variabel proposisional X_y , di mana $X \in \{A, B, C\}$ dan $1 \leq y \leq 4$, dengan arti bahwa " X_y benar jika dan hanya jika orang-orang berkebangsaan X duduk di kursi y di bangku." Bagaimana Anda menentukan masalah ini dalam bentuk CNF (Constant Shape Form) atas X_y sedemikian rupa sehingga model CNF tersebut sesuai persis dengan semua solusi tempat duduk keempat pria tersebut?

CNF akan menentukan kondisi berikut:

1. Setiap kursi ditempati setidaknya oleh satu orang: $(A_1 \mid B_1 \mid C_1), (A_2 \mid B_2 \mid C_2), (A_3 \mid B_3 \mid C_3), (A_4 \mid B_4 \mid C_4)$. Kebenaran klausa-klausa ini menyiratkan bahwa setidaknya ada empat orang yang menggunakan kursi-kursi ini.
 2. Setiap kursi dapat menampung paling banyak satu orang: $(\neg A_1 \mid \neg B_1), (\neg A_1 \mid \neg C_1), (\neg B_1 \mid \neg C_1), (\neg A_2 \mid \neg B_2), (\neg A_2 \mid \neg C_2), (\neg B_2 \mid \neg C_2), (\neg A_3 \mid \neg B_3), (\neg A_3 \mid \neg C_3), (\neg B_3 \mid \neg C_3), (\neg A_4 \mid \neg B_4), (\neg A_4 \mid \neg C_4), (\neg B_4 \mid \neg C_4)$. Jika digabungkan dengan kondisi pertama, kondisi ini menyiratkan bahwa ada tepat empat orang yang duduk di bangku tersebut.
 3. Paling banyak dua orang Amerika di bangku hakim: $(\neg A_1 \mid \neg A_2 \mid \neg A_3), (\neg A_1 \mid \neg A_2 \mid \neg A_4), (\neg A_1 \mid \neg A_3 \mid \neg A_4), (\neg A_2 \mid \neg A_3 \mid \neg A_4)$. Klausul-klausul ini menyatakan bahwa untuk setiap tiga kursi, salah satunya tidak boleh ditempati oleh orang Amerika.
 4. Paling banyak satu orang Inggris di bangku hakim: $(\neg B_1 \mid \neg B_2), (\neg B_1 \mid \neg B_3), (\neg B_1 \mid \neg B_4), (\neg B_2 \mid \neg B_3), (\neg B_2 \mid \neg B_4), (\neg B_3 \mid \neg B_4)$. Klausul-klausul ini menyatakan bahwa untuk setiap dua kursi, salah satunya tidak boleh ditempati oleh orang Inggris.
 5. Paling banyak satu orang Kanada di bangku cadangan: $(\neg C_1 \mid \neg C_2), (\neg C_1 \mid \neg C_3), (\neg C_1 \mid \neg C_4), (\neg C_2 \mid \neg C_3), (\neg C_2 \mid \neg C_4), (\neg C_3 \mid \neg C_4)$.
 6. Orang Amerika tidak ingin duduk bersebelahan: $(\neg A_1 \mid \neg A_2), (\neg A_2 \mid \neg A_3), (\neg A_3 \mid \neg A_4)$.
 7. Orang Inggris suka duduk di sebelah orang Kanada: $(\neg C_1 \mid B_2), (\neg C_2 \mid B_1 \mid B_3), (\neg C_3 \mid B_2 \mid B_4), (\neg C_4 \mid B_3), (\neg B_1 \mid C_2), (\neg B_2 \mid C_1 \mid C_3), (\neg B_3 \mid C_2 \mid C_4), (\neg B_4 \mid C_3)$.
- Jika kita membaca $(\neg C_2 \mid B_1 \mid B_3)$ sebagai $C_2 \rightarrow B_1 \vee B_3$, itu berarti jika orang Kanada mengambil kursi kedua, maka orang Inggris harus duduk di kursi 1 atau kursi 3.

Beberapa kondisi seperti setidaknya ada dua orang Amerika, dan setidaknya satu orang Inggris dan satu orang Kanada, juga dapat dengan mudah ditentukan oleh klausa. Namun, klausa-klausa ini merupakan konsekuensi logis dari klausa-klausa lainnya, karena kondisi 1 dan 2 memastikan bahwa ada empat orang yang duduk di bangku. Jika paling banyak hanya ada satu orang Inggris dan satu orang Kanada, maka kondisi 2 juga berlebihan, dan kita dapat menghapusnya dengan aman.

Klausul masukan suatu masalah berfungsi sebagai himpunan aksioma. Himpunan ini minimum jika tidak ada klausa yang merupakan konsekuensi logis dari klausa lain. Tentu saja, sangat mahal untuk memastikan bahwa suatu himpunan klausa adalah minimum. Artinya, tanpa pencarian menyeluruh, kita tidak dapat mengatakan bahwa setiap klausa independen, yaitu, bukan konsekuensi logis dari klausa lain. Ketika menentukan suatu masalah dalam logika proposisional, kita sering mengabaikan sifat minimum dari klausa masukan.

Masalah selanjutnya disebut *Ksatria dan Penipu*, yang dicetuskan oleh Raymond Smullyan dalam karyanya tahun 1978, *Apa Nama Buku Ini?* Masalah ini sebenarnya adalah kumpulan teka-teki logika serupa di mana beberapa karakter hanya dapat menjawab pertanyaan dengan jujur, dan yang lain hanya dengan salah. Teka-teki tersebut berlatar di sebuah pulau fiktif di mana semua penduduknya adalah ksatria, yang selalu mengatakan yang sebenarnya, atau penipu, yang selalu berbohong. Teka-teki ini melibatkan seorang pengunjung pulau yang bertemu dengan kelompok kecil penduduk. Biasanya, tujuannya adalah agar pengunjung menyimpulkan tipe penduduk dari pernyataan mereka, tetapi beberapa teka-teki jenis ini meminta fakta lain untuk disimpulkan. Teka-teki tersebut juga dapat berupa menentukan pertanyaan ya-tidak yang dapat diajukan pengunjung untuk menemukan informasi tertentu.

Contoh 2.5.4 Anda bertemu dua penduduk: Zoey dan Mel. Zoey memberi tahu Anda bahwa “Mel adalah seorang penipu.” Mel berkata, “Baik Zoey maupun saya bukanlah seorang penipu.” Dapatkah Anda menentukan siapa yang merupakan seorang ksatria dan siapa yang merupakan seorang penipu?

Misalkan p mewakili “Zoey adalah seorang ksatria,” dengan pemahaman bahwa jika p benar, maka Zoey adalah seorang ksatria; jika p salah, maka Zoey adalah seorang penipu. Demikian pula, definisikan q sebagai “Mel adalah seorang ksatria.” Dengan menggunakan fungsi, apa yang mereka katakan dapat diungkapkan sebagai

$$(1) \text{ says}(\text{Zoey}, \neg q); \quad (2) \text{ says}(\text{Mel}, p \wedge q),$$

bersama dengan dua aksioma:

$$\text{says}(\text{knight}, X) \equiv X; \text{ says}(\text{knave}, X) \equiv \neg X$$

Sekarang, dengan menggunakan tabel kebenaran untuk memeriksa semua nilai kebenaran p dan q , hanya ketika $p \rightarrow 1$ dan $q \rightarrow 0$, maka (1) dan (2) keduanya benar. Artinya, kita mengganti Zoey dengan ksatria dan Mel dengan penipu dalam (1) dan (2), untuk mendapatkan $\neg q$ dan $\neg(p \wedge q)$, masing-masing. Maka $\{p \rightarrow 1, q \rightarrow 0\}$ adalah model dari $\neg q \wedge \neg(p \wedge q)$. Jadi, solusinya adalah “Zoey adalah seorang ksatria dan Mel adalah seorang penipu.”

Contoh 2.5.5 Anda bertemu dua penduduk: Peggy dan Zippy. Peggy memberi tahu Anda bahwa “dari Zippy dan saya, tepat satu yang merupakan seorang ksatria.” Zippy memberi tahu Anda bahwa hanya seorang penipu yang akan mengatakan bahwa Peggy adalah seorang penipu. Dapatkah Anda menentukan siapa yang merupakan seorang ksatria dan siapa yang merupakan seorang penipu?

Misalkan p adalah “Peggy adalah seorang ksatria” dan q adalah “Zippy adalah seorang ksatria.” Lalu apa yang mereka katakan dapat diungkapkan sebagai

$$(1) \text{ says}(Peggy, p \oplus q); (2) \text{ says}(Zippy, \text{says}(knave, \neg p)).$$

Satu-satunya solusi adalah $p \rightarrow 0$ dan $q \rightarrow 0$, yaitu, baik Peggy maupun Zippy adalah penipu.

2.6 LATIHAN SOAL

1. Berapa banyak operator logika berbeda yang memiliki lima argumen? Artinya, berapa banyak fungsi Boolean tipe $f : B^5 \rightarrow B$, di mana $B = \{0, 1\}$?
2. Misalkan $V_P = \{p, q, r\}$. Maka $M((p \rightarrow q) \wedge r) = ?$ dan $M((p \rightarrow q) \wedge \neg q) = ?$
3. (Teorema 2.2.11) Buktikan bahwa, diberikan dua rumus proposisional A dan B , $M(A \wedge B) = M(A) \cap M(B)$ dan $M(\neg A) = All_p - M(A)$.
4. Jawablah pertanyaan-pertanyaan berikut dengan penjelasan:
 - (a) Misalkan Anda telah menunjukkan bahwa setiap kali X benar, maka Y benar, dan setiap kali X salah, maka Y salah. Apakah Anda sekarang telah menunjukkan bahwa X dan Y secara logis setara?
 - (b) Misalkan Anda telah menunjukkan bahwa setiap kali X benar, maka Y benar, dan setiap kali Y salah, maka X salah. Apakah Anda sekarang telah menunjukkan bahwa X dan Y secara logis setara?
 - (c) Misalkan Anda tahu bahwa X benar jika dan hanya jika Y benar, dan Anda tahu bahwa Y benar jika dan hanya jika Z benar. Apakah ini cukup untuk menunjukkan bahwa X, Y , dan Z semuanya secara logis setara?
 - (d) Misalkan Anda tahu bahwa setiap kali X benar, maka Y benar; bahwa setiap kali Y benar, maka Z benar; dan setiap kali Z benar, maka X benar. Apakah ini cukup untuk menunjukkan bahwa X, Y , dan Z semuanya secara logis setara?
5. Berikan tabel kebenaran untuk mendefinisikan $ite: B^3 \rightarrow B$, operator jika-maka-lain.
6. Berikan tata bahasa BNF yang mendefinisikan rumus-rumus yang terbentuk dengan baik tanpa tanda kurung yang tidak perlu, dengan asumsi relasi prioritas dari yang tertinggi ke yang terendah: $\neg, \wedge, \vee, \rightarrow, \{\oplus, \leftrightarrow\}$.
7. Buktikan dengan metode tabel kebenaran bahwa rumus-rumus berikut ini valid.
 - (a) $(A \wedge (A \rightarrow B)) \rightarrow B$
 - (b) $(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$
 - (c) $(A \oplus B) \leftrightarrow ((A \vee B) \wedge (\neg A \vee \neg B))$

- (d) $(A \leftrightarrow B) \leftrightarrow ((A \vee \neg B) \wedge (\neg A \vee B))$
8. Buktikan dengan metode tabel kebenaran bahwa implikasi berikut berlaku:
- (a) $(A \wedge (A \rightarrow B)) \models B$
- (b) $(A \rightarrow (B \rightarrow C)) \models (A \rightarrow B) \rightarrow (A \rightarrow C)$
- (c) $(\neg B \rightarrow \neg A) \models (A \rightarrow B)$
- (d) $((A \vee B) \wedge (\neg A \vee C)) \models (B \vee C)$
9. Buktikan dengan metode tabel kebenaran bahwa kesetaraan logis berikut berlaku:
- (a) $\neg(A \vee B) \equiv \neg A \wedge \neg B$
- (b) $(A \vee B) \wedge (A \vee \neg B) \equiv A$
- (c) $(A \leftrightarrow B) \equiv ((A \wedge B) \vee (\neg A \wedge \neg B))$
- (d) $(A \oplus B) \equiv ((A \wedge \neg B) \vee (\neg A \wedge B))$
- (e) $(A \downarrow B) \uparrow C \equiv A \uparrow (B \downarrow C)$
- (f) $(A \uparrow B) \downarrow C \equiv A \downarrow (B \uparrow C)$
10. Buktikan kesetaraan rumus-rumus berikut dengan menggunakan relasi kesetaraan lainnya:
- (a) $A \oplus B \equiv \neg(A \leftrightarrow B) \equiv \neg A \leftrightarrow B \equiv A \leftrightarrow \neg B$
- (b) $(A \rightarrow B) \wedge (A \rightarrow C) \equiv \neg A \vee (A \wedge B \wedge C)$
- (c) $A \rightarrow (B \rightarrow (C \vee D)) \equiv (A \rightarrow C) \vee (B \rightarrow D).$
11. Buktikan Teorema 2.2.4. (Petunjuk: Induksi pada struktur A .)
12. Buktikan bahwa untuk setiap rumus A, B , dan C , jika $A \models B$, maka $A \wedge C \models B \wedge C$.
13. Definisikan semua 16 fungsi Boolean biner dengan rumus yang hanya menggunakan $\neg$, $\downarrow$, dan $\wedge$.
14. Buktikan bahwa operator nor $\downarrow$, di mana $x \downarrow y = \neg(x \vee y)$, dapat berfungsi sebagai himpunan operator Boolean yang minimal cukup.
15. Ubah rumus-rumus berikut menjadi CNF yang setara:
- (a) $p \rightarrow (q \wedge r).$
- (b) $(p \rightarrow q) \rightarrow r.$
- (c) $\neg(\neg p \vee q) \vee (r \rightarrow \neg s).$
- (d) $p \vee (\neg q \wedge (r \rightarrow \neg p)).$
- (e) $\neg(((p \rightarrow q) \rightarrow p) \rightarrow q).$
- (f) $(p \rightarrow q) \leftrightarrow (p \rightarrow r)$
16. Ubah rumus-rumus berikut menjadi DNF yang setara:
- (a) $p \rightarrow (q \wedge r).$
- (b) $(p \rightarrow q) \rightarrow r.$
- (c) $\neg(\neg p \vee q) \vee (r \rightarrow \neg s).$
- (d) $p \vee (\neg q \wedge (r \rightarrow \neg p)).$
- (e) $\neg(((p \rightarrow q) \rightarrow p) \rightarrow q).$
- (f) $(p \rightarrow q) \leftrightarrow (p \rightarrow r).$
17. Susunlah bentuk CNF lengkap untuk rumus-rumus berikut, di mana $\bar{a}$ adalah $\neg a$, $+$ adalah $\vee$ dan $\wedge$ dihilangkan:
- (a) $F = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + a\bar{b}c.$
- (b) $G = abc + \bar{a}bc + a\bar{b}c + ab\bar{c}.$

- (c) $H = a \oplus b \oplus c$.
 (d) $I = (a \wedge b) \rightarrow (b \rightarrow c)$.
18. Susun DNF lengkap untuk rumus-rumus berikut, di mana $\bar{a}$ adalah $\neg a$, $+$ adalah $\vee$ dan $\wedge$ dihilangkan:
- (a) $F = (\bar{a} + \bar{b} + c)(\bar{a} + b + \bar{c})(a + \bar{b} + \bar{c})(\bar{a} + \bar{b} + \bar{c})$.
 (b) $G = (a + b + c)(\bar{a} + b + c)(a + \bar{b} + c)(a + b + \bar{c})$.
 (c) $H = a \oplus b \oplus c$.
 (d) $I = (a \wedge b) \rightarrow (b \rightarrow c)$.
19. Asumsikan $a > b > c$; buat ROBDD untuk rumus-rumus berikut, di mana $\bar{a}$ adalah $\neg a$, $+$ adalah $\vee$ dan $\wedge$ dihilangkan sebagai produksi:
- (a) $F = abc + \bar{a}\bar{b}c + a\bar{b}\bar{c}$.
 (b) $G = \bar{a}bc + abc + a\bar{b}c$.
 (c) $H = a \oplus b \oplus c$.
 (d) $I = (a \wedge b) \rightarrow (b \rightarrow c)$
20. Misalkan F adalah himpunan semua rumus yang dibangun di atas himpunan n variabel proposisional. Buktikan bahwa terdapat tepat 2^{2^n} kelas rumus ekuivalen di antara F .
21. (a) Buktikan bahwa $\{\rightarrow, \neg\}$ adalah himpunan operator Boolean yang minimal cukup.
 (b) Buktikan bahwa $\{\rightarrow, \perp\}$ adalah himpunan operator Boolean yang minimal cukup.
22. Buktikan bahwa $\{\wedge, \vee, \rightarrow, \top\}$ bukan himpunan operator Boolean yang cukup.
23. Buktikan bahwa:
- (a) $\{\wedge, \vee, \top, \perp\}$ bukan himpunan operator Boolean yang cukup dan
 (b) jika kita menambahkan fungsi o apa pun yang tidak dapat dinyatakan dalam bentuk $\wedge, \vee, \top$, dan $\perp$, maka $\{\wedge, \vee, \top, \perp, o\}$ adalah himpunan yang cukup.
24. Identifikasi pasangan rumus ekuivalen dari kandidat berikut:
- (a) $p \uparrow (q \uparrow r)$
 (b) $(p \uparrow q) \uparrow r$
 (c) $\uparrow(p, q, r)$
 (d) $p \downarrow (q \downarrow r)$
 (e) $(p \downarrow q) \downarrow r$
 (f) $\downarrow(p, q, r)$
 (g) $p \uparrow (q \downarrow r)$
 (h) $(p \uparrow q) \downarrow r$
 (i) $p \downarrow (q \uparrow r)$
 (j) $(p \downarrow q) \uparrow r$
25. Cobalah untuk menemukan DNF yang setara dengan ukuran minimal untuk rumus DNF berikut menggunakan peta K:
- (a) $f(x, y, z) = \bar{x}\bar{y}\bar{z} + \bar{x}y + xy\bar{z} + xz$.
 (b) $f(x, y, z) = \bar{x}y + y\bar{z} + yz + x\bar{y}\bar{z}$.
 (c) $f(x, y, z, w) = \bar{x}\bar{y}zw + xyz + y\bar{z}\bar{w} + y\bar{z}w$.
 (d) $f(x, y, z, w) = m_0 + m_1 + m_5 + m_7 + m_8 + m_{10} + m_{14} + m_{15}$,
 di mana m_i hasil perkalian penuh dari x, y, z dan w , and i adalah nilai desimal dari string biner $xyzw$.

26. Cobalah untuk menemukan CNF yang setara dengan ukuran minimal untuk fungsi Boolean berikut menggunakan peta Karnaugh (K-map):
- (a) $f(x, y, z) = \bar{x}\bar{y} + \bar{x}y + xy\bar{z} + xz$.
 - (b) $f(x, y, z) = \bar{x}yz + xy\bar{z} + x\bar{y}z + \bar{x}\bar{y}\bar{z}$.
 - (c) $f(x, y, z, w) = m_0 + m_1 + m_3 + m_4 + m_5 + m_7 + m_{12} + m_{13} + m_{15}$.
 - (d) $f(x, y, z, w) = m_0 + m_1 + m_2 + m_4 + m_5 + m_6 + m_8 + m_9 + m_{12} + m_{13} + m_{14}$.
27. Suatu fungsi $f: B^n \rightarrow B$, di mana $B = \{0, 1\}$, disebut linear jika $f(x_1, x_2, \dots, x_n) = a_0 + a_1x_1 + \dots + a_nx_n$ untuk koefisien yang sesuai $a_0, \dots, a_n \in B$. Di sini $+$ menunjukkan penjumlahan modulo 2, dan perkalian standar (tidak ditampilkan) adalah perkalian atas bilangan bulat.
- (a) Tunjukkan bahwa representasi fungsi linear f di atas adalah unik.
 - (b) Tentukan jumlah fungsi linear n -ary.
28. Misalkan $a = 31$ dan $b = 19$. Berikan vektor 8-bit $(a)_2$ dan $(b)_2$ untuk a dan b dan tunjukkan hasilnya (baik dalam vektor bit maupun nilai desimal) dari $(\neg a)_2$, $(\neg b)_2$, $(a \mid b)$, $(a \& b)$, $(a \wedge b)_2$, $a \gg 2$, $b \gg 2$, $a \ll 2$, dan $b \ll 2$.
29. Misalkan x adalah bilangan bulat positif 32-bit. Berikan program Java untuk setiap tugas pemrograman berikut, menggunakan sebanyak mungkin operasi bitwise:
- (a) Periksa apakah x genap atau ganjil.
 - (b) Dapatkan posisi bit tertinggi 1 dalam x .
 - (c) Dapatkan posisi bit terendah 1 dalam x .
 - (d) Atur bit ke- i dari x menjadi 1, di mana i adalah bilangan bulat.
 - (e) Atur bit ke- i dari x menjadi 0, di mana i adalah bilangan bulat.
 - (f) Hitung angka nol di belakang x sebagai bilangan biner.
 - (g) Geser x ke kanan sebanyak 3 bit dan letakkan 3 bit terakhir dari x di awal hasil pergeseran. Seluruh proses ini disebut "rotasi x ke kanan" sebanyak 3 bit.
 - (h) Konversikan serangkaian angka desimal menjadi bilangan biner tanpa menggunakan perkalian dengan 10.
30. Misalkan sebuah rumus A berisi 15 variabel, setiap variabel diwakili oleh bilangan bulat dari 0 hingga 14. Kita menggunakan 15 bit terakhir dari variabel bilangan bulat x untuk mewakili interpretasi A , sedemikian sehingga bit ke- i adalah 1 jika dan hanya jika kita menetapkan 1 ke variabel i . Modifikasi algoritma $eval(A, \sigma)$, di mana σ diganti dengan x menggunakan operasi bitwise. Rancang algoritma yang mencantumkan semua interpretasi A dan terapkan setiap interpretasi ke A menggunakan $eval(A, x)$. Kode semu algoritma tersebut harus seperti program Java.
31. Arti dari variabel proposisional berikut diberikan di bawah ini:
- t : "pajak dinaikkan"
 - e : "pengeluaran meningkat"
 - d : "batas utang dinaikkan"
 - c : "biaya pengumpulan pajak meningkat"
 - g : "pemerintah meminjam lebih banyak uang"
 - i : "suku bunga meningkat"

- (a) Nyatakan pernyataan-pernyataan berikut sebagai rumus proposisional;
- (b) ubah rumus-rumus tersebut menjadi CNF.

Pajak dinaikkan atau jika pengeluaran meningkat maka batas utang dinaikkan. Jika pajak dinaikkan, maka biaya pengumpulan pajak meningkat. Jika peningkatan pengeluaran berarti pemerintah meminjam lebih banyak uang, maka jika batas utang dinaikkan, maka suku bunga meningkat. Jika pajak tidak dinaikkan dan biaya pengumpulan pajak tidak meningkat, maka jika batas utang dinaikkan, maka pemerintah meminjam lebih banyak uang. Biaya pengumpulan pajak tidak meningkat. Suku bunga tidak meningkat, atau pemerintah tidak meminjam lebih banyak uang.

32. Tentukan teka-teki berikut dalam logika proposisional dan ubah spesifikasi tersebut menjadi CNF. Temukan model CNF Anda dan gunakan model ini untuk membangun solusi teka-teki tersebut.

Empat anak, Abel, Bob, Carol, dan David, sedang makan siang di hari musim panas yang terik. Masing-masing memiliki segelas besar air, sandwich, dan jenis buah yang berbeda (apel, pisang, jeruk, dan anggur).

Buah apa yang dimiliki setiap anak? Petunjuk yang diberikan adalah:

- (a) Abel dan Bob harus mengupas buah mereka sebelum memakannya.
 - (b) Carol tidak suka anggur.
 - (c) Abel memiliki serbet untuk menyeka jus dari jarinya.
33. Tentukan teka-teki berikut dalam logika proposisional dan ubah spesifikasi tersebut menjadi CNF. Temukan model CNF Anda dan gunakan model ini untuk membangun solusi teka-teki tersebut.

Empat mobil listrik diparkir di stasiun pengisian daya dan jarak tempuhnya masing-masing adalah 10.000, 20.000, 30.000, dan 40.000 mil. Biaya pengisian dayanya masing-masing adalah Rp. 150.000, Rp. 300.000, Rp. 450.000, dan Rp. 600.000.

Cari tahu berapa biaya pengisian daya untuk setiap mobil dari petunjuk berikut:

- (a) Mobil Jerman memiliki jarak tempuh 30.000 mil.
- (b) Mobil Jepang memiliki jarak tempuh 20.000 mil dan biaya pengisiannya Rp. 150.000.
- (c) Mobil Prancis memiliki jarak tempuh lebih sedikit daripada mobil Italia.
- (d) Kendaraan yang dikenakan biaya Rp. 300.000 memiliki jarak tempuh 10.000 mil lebih banyak daripada mobil Italia.
- (e) Mobil yang dikenakan biaya Rp. 450.000 memiliki jarak tempuh 20.000 mil lebih sedikit daripada mobil Jerman.

34. Tentukan teka-teki berikut dalam logika proposisional dan ubah spesifikasi tersebut menjadi CNF. Temukan model CNF Anda dan gunakan model ini untuk membangun solusi teka-teki tersebut.

Empat anak laki-laki sedang menunggu makan malam Thanksgiving dan masing-masing memiliki makanan favorit yang unik. Nama mereka adalah Larry, Nick, Philip, dan Tom. Usia mereka semua berbeda dan termasuk dalam himpunan {8, 9, 10, 11}.

Cari tahu makanan apa yang mereka harapkan untuk dimakan dan berapa usia mereka.

- (a) Larry menantikan untuk makan kalkun.
 - (b) Anak laki-laki yang menyukai pai labu berusia 1 tahun lebih muda dari Philip.
 - (c) Tom lebih muda dari anak laki-laki yang menyukai kentang tumbuk.
 - (d) Anak laki-laki yang menyukai ham berusia 2 tahun lebih tua dari Philip.
35. Enam orang duduk di bangku panjang untuk foto grup. Dua di antaranya orang Amerika (A), dua orang Kanada (C), satu orang Italia (I), dan satu orang Spanyol (S). Misalkan kita menggunakan variabel proposisional X_y , di mana $x \in \{A, C, I, S\}$ dan $1 \leq y \leq 6$, dan X_y benar jika dan hanya jika orang-orang berkebangsaan X duduk di kursi y di bangku tersebut. Harap tentukan masalah ini secara formal dalam rumus proposisional menggunakan X_y , dan ubah rumus tersebut menjadi CNF sedemikian rupa sehingga setiap model CNF memberi kita satu solusi agar keenam orang tersebut duduk di bangku, dengan menerapkan aturan bahwa:
- (a) orang-orang dari negara yang sama tidak akan duduk bersebelahan
 - (b) orang Italia harus dikelilingi oleh orang Amerika, dan
 - (c) setiap orang Amerika duduk di sebelah orang Kanada. Harap sebutkan semua model CNF Anda.
36. Sebuah pulau yang sangat istimewa hanya dihuni oleh ksatria dan penipu. Ksatria selalu mengatakan yang sebenarnya, dan penipu selalu berbohong. Anda bertemu dua penduduk: Sally dan Zippy. Sally menyatakan, "Saya dan Zippy tidak sama." Zippy berkata, "Dari saya dan Sally, tepat satu adalah seorang ksatria." Dapatkah Anda menentukan siapa yang merupakan ksatria dan siapa yang merupakan penipu?
37. Sebuah pulau yang sangat istimewa hanya dihuni oleh ksatria dan penipu. Ksatria selalu mengatakan yang sebenarnya, dan penipu selalu berbohong. Anda bertemu dua penduduk: Homer dan Bozo. Homer memberi tahu Anda, "Setidaknya salah satu dari berikut ini benar: bahwa saya adalah seorang ksatria atau bahwa Bozo adalah seorang ksatria." Bozo menyatakan, "Homer bisa mengatakan bahwa saya adalah seorang penipu." Dapatkah Anda menentukan siapa yang merupakan ksatria dan siapa yang merupakan penipu?
38. Sebuah pulau yang sangat istimewa hanya dihuni oleh ksatria dan penipu. Ksatria selalu mengatakan yang sebenarnya, dan penipu selalu berbohong. Anda bertemu dua penduduk: Bart dan Ted. Bart menyatakan, "Saya dan Ted sama-sama ksatria atau sama-sama penipu." Ted berkata kepadamu, "Bart akan memberitahumu bahwa aku

adalah seorang penipu.” Dapatkah kamu menentukan siapa yang ksatria dan siapa yang penipu?

BAB 3

PENALARAN DALAM LOGIKA PROPOSISIONAL

Pada Bab 1, kita telah memperkenalkan konsep “prosedur pembuktian.” Bab ini akan menyajikan banyak prosedur pembuktian untuk logika proposisional. Tujuan dari presentasi ini bukanlah untuk memperkenalkan prosedur pembuktian yang efisien, tetapi untuk menyajikan formalisme sehingga logika lain dapat menggunakannya.

3.1 PROSEDUR PEMBUKTIAN

Diberikan himpunan aksioma A (yaitu, rumus-rumus yang dianggap benar) dan sebuah rumus B , sebuah prosedur pembuktian $P(A, B)$ akan menjawab pertanyaan apakah B merupakan teorema dari A . Jika $P(A, B)$ mengembalikan nilai benar, kita katakan B terbukti; jika $P(A, B)$ mengembalikan nilai salah, kita katakan B tidak terbukti. Secara umum, himpunan aksioma A dianggap setara dengan konjungsi rumus-rumus dalam A . Untuk logika proposisional, sebuah teorema adalah setiap rumus B sedemikian sehingga $A \models B$, yaitu, $B \in T(A)$. Artinya, $P(A, B)$ juga disebut pembukti teorema. Prosedur P yang sama dapat digunakan untuk menunjukkan bahwa B valid (yaitu, B adalah tautologi) dengan memanggil $P(T, B)$, atau bahwa C tidak dapat dipenuhi dengan memanggil $P(C, \perp)$.

Artinya, membuktikan relasi implikasi tidak lebih sulit daripada membuktikan validitas atau membuktikan ketidakpuasan. Dalam logika proposisional, ketiga masalah tersebut setara dan termasuk dalam kelas masalah co-NP-complete, yaitu komplemen dari kelas masalah NP-complete. Masalah co-NP-complete dianggap "lebih sulit" daripada masalah NP-complete, karena yang terakhir memiliki algoritma waktu polinomial untuk memverifikasi solusinya, tetapi yang pertama tidak diketahui memiliki algoritma seperti itu.

Jenis dan Gaya Prosedur Pembuktian

Dalam praktiknya, setiap prosedur pembuktian dirancang untuk menyelesaikan salah satu dari tiga masalah dan termasuk dalam salah satu dari tiga jenis prosedur pembuktian.

- Pembuktian teorema
 $P(X, Y)$ mengembalikan nilai benar jika dan hanya jika rumus Y adalah teorema (suatu implikasi) dari X (dengan asumsi himpunan rumus setara dengan konjungsi dari semua anggotanya).
- Pembuktian tautologi
 $T(A)$ mengembalikan nilai benar jika dan hanya jika rumus A valid (suatu tautologi).
- Pembuktian sanggahan
 $R(B)$ mengembalikan nilai benar jika dan hanya jika rumus B tidak dapat dipenuhi.

Jika kita memiliki salah satu dari tiga prosedur di atas, yaitu $P(X, Y)$, $T(A)$, atau $R(B)$, kita dapat mengimplementasikan dua prosedur lainnya sebagai berikut:

- Jika $P(X, Y)$ tersedia, implementasikan $T(A)$ sebagai $P(T, A)$ dan $R(B)$ sebagai $P(B, \perp)$.
- Jika $T(A)$ tersedia, implementasikan $P(X, Y)$ sebagai $T(X \rightarrow Y)$ dan $R(B)$ sebagai $T(\neg B)$.

- Jika $R(B)$ tersedia, implementasikan $P(X, Y)$ sebagai $R(X \wedge \neg Y)$ dan $T(A)$ sebagai $R(\neg A)$.

Kebenaran implementasi di atas didasarkan pada teorema berikut, yang pembuktiannya diserahkan sebagai latihan.

Teorema 3.1.1 Diberikan rumus A dan B , tiga pernyataan berikut ini setara:

1. $A \models B$.
2. $A \rightarrow B$ valid.
3. $A \wedge \neg B$ tidak dapat dipenuhi.

Dengan demikian, satu pembukti dapat melakukan ketiganya, baik secara langsung maupun tidak langsung. Pada Bab 2, kita telah membahas beberapa konsep dan beberapa di antaranya dapat digunakan untuk membangun prosedur pembuktian seperti yang diilustrasikan di bawah ini.

- **Tabel kebenaran:** Dapat digunakan untuk membangun pembuktian teorema, pembuktian tautologi, dan pembuktian sanggahan.
- **CNF:** Dapat digunakan untuk membangun pembuktian tautologi, karena CNF valid jika dan hanya jika setiap klausa valid, dan sangat mudah untuk memeriksa apakah suatu klausa valid.
- **DNF:** Dapat digunakan untuk membangun pembuktian sanggahan, karena DNF tidak dapat dipenuhi jika dan hanya jika setiap produk tidak dapat dipenuhi, dan sangat mudah untuk memeriksa apakah suatu produk tidak dapat dipenuhi.
- **ROBDD atau INF:** Dapat digunakan untuk membangun pembuktian tautologi atau pembuktian sanggahan, karena ROBDD adalah bentuk kanonik. Rumus A valid (tidak dapat dipenuhi) jika dan hanya jika bentuk kanonik A adalah $T (\perp)$.
- **Substitusi aljabar:** Ini dapat digunakan untuk membangun pembukti tautologi atau pembukti penyangkalan, karena substitusi "sama dengan sama" mempertahankan relasi ekuivalensi. Jika kita sampai pada T atau $\perp$, kita dapat menyatakan bahwa rumus masukan valid atau tidak dapat dipenuhi.

Dalam hal prinsip kerjanya, prosedur pembuktian dapat memiliki salah satu dari tiga gaya: enumerasi, reduksi, dan deduksi.

- **Gaya enumerasi:** mengenumerasi semua interpretasi yang terlibat.
- **Gaya reduksi:** menggunakan relasi ekuivalensi untuk mengubah sekumpulan rumus menjadi sekumpulan rumus yang lebih sederhana.
- **Gaya deduksi:** menggunakan aturan inferensi untuk menyimpulkan rumus baru.

Berdasarkan definisi, $A \models B$ jika dan hanya jika $M(A) \subseteq M(B)$, yaitu, setiap model A adalah model B . Kita dapat menggunakan tabel kebenaran untuk menghitung semua interpretasi yang melibatkan A dan B , untuk melihat apakah setiap model A adalah model B . Prosedur pembuktian berdasarkan tabel kebenaran adalah prosedur pembuktian gaya enumerasi yang menggunakan pencarian menyeluruh. Gaya prosedur pembuktian ini mudah dijelaskan tetapi

sangat tidak efisien. Misalnya, jika A berisi 30 variabel, kita perlu membuat tabel kebenaran dengan 230 baris.

Prosedur pembuktian gaya reduksi menggunakan relasi kesetaraan untuk mengubah sekumpulan rumus menjadi rumus sederhana yang diinginkan, seperti diagram keputusan biner terurut tereduksi (ROBDD). Karena ROBDD bersifat kanonik, kita memutuskan bahwa suatu rumus valid jika ROBDD-nya adalah T (atau 1), atau tidak dapat dipenuhi jika ROBDD-nya adalah $\perp$ (atau 0). Kita juga dapat menggunakan bentuk normal disjungtif (DNF) untuk menunjukkan bahwa suatu formula tidak dapat dipenuhi: DNF yang disederhanakan adalah $\perp$. Selanjutnya, kita akan memperkenalkan metode tableau semantik, yang merupakan prosedur pembuktian gaya reduksi dan bekerja dengan cara yang sama seperti mendapatkan DNF dari formula masukan.

Prosedur pembuktian gaya deduksi menggunakan aturan inferensi untuk menghasilkan formula baru, hingga formula yang diinginkan dihasilkan. Aturan inferensi biasanya mengambil satu atau dua formula sebagai masukan dan menghasilkan satu atau dua formula baru sebagai keluaran. Beberapa aturan ini berasal dari relasi kesetaraan antara formula masukan dan keluaran. Secara umum, kita mensyaratkan bahwa keluaran dari aturan inferensi harus diimplikasikan oleh masukan.

Banyak prosedur pembuktian tidak diklasifikasikan secara jelas sebagai salah satu gaya enumerasi, reduksi, atau deduksi. Namun, biasanya prinsip kerjanya dapat dipahami sebagai melakukan enumerasi, atau reduksi, atau deduksi, atau hibrida dari semuanya secara implisit. Misalnya, mengubah formula menjadi DNF dapat berupa reduksi; Namun, mengubah formula menjadi DNF lengkap dapat berupa enumerasi, karena setiap istilah dalam DNF lengkap sesuai dengan satu model formula. Dalam praktiknya, reduksi sering digunakan dalam sebagian besar prosedur pembuktian bergaya deduksi untuk efisiensi.

Dalam Bab 1, kita mengatakan bahwa suatu prosedur adalah prosedur keputusan jika prosedur tersebut berakhir untuk setiap input dan menjawab pertanyaan dengan benar. Untungnya bagi logika proposisional, setiap prosedur pembuktian yang diperkenalkan dalam bab ini adalah prosedur keputusan. Logika lain mungkin tidak memiliki prosedur keputusan untuk pertanyaan tertentu.

Tabel Semantik

Metode tabel semantik, juga disebut metode "pohon kebenaran", adalah prosedur pembuktian gaya reduksi bukan hanya untuk logika proposisional, tetapi juga untuk logika lain, termasuk logika orde pertama, temporal, dan modal. Tabel untuk suatu formula adalah struktur pohon; setiap simpul pohon dikaitkan dengan himpunan formula yang diturunkan dari formula masukan. Metode tabel membangun pohon ini dan menggunakannya untuk membuktikan/menyangkal kepuasan formula masukan. Metode tabel juga dapat menentukan kepuasan himpunan formula terbatas dari berbagai logika. Ini adalah prosedur pembuktian yang paling populer untuk logika modal.

Untuk logika proposisional, tabel semantik memeriksa apakah suatu formula dapat dipenuhi atau tidak, dengan "memecah" formula kompleks menjadi formula yang lebih kecil seperti yang kita lakukan untuk mengubah formula menjadi DNF (bentuk normal disjungtif).

Tabel yang digunakan oleh metode ini adalah struktur pohon yang mencatat proses transformasi.

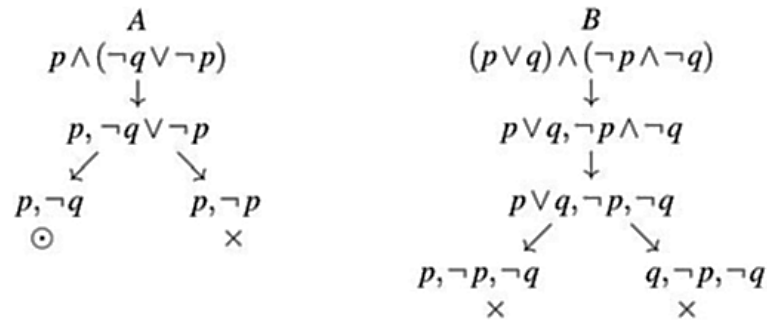
Metode ini dimulai dengan simpul akar yang berisi rumus masukan dan mengembangkan pohon dengan memperluas simpul daun dan menetapkan serangkaian rumus ke simpul baru. Dalam setiap perluasan, satu atau dua penerus ditambahkan ke simpul daun yang dipilih.

- Setiap simpul pohon diberi serangkaian rumus dan koma “,” dalam himpunan tersebut merupakan sinonim untuk $\wedge$.
- Suatu simpul disebut tertutup jika berisi pasangan literal (atau rumus) yang saling melengkapi, atau $\perp$, atau $\neg T$.
- Suatu simpul disebut terbuka jika hanya berisi serangkaian literal yang konsisten, yang setara dengan produk dalam DNF.
- Jika suatu simpul tidak tertutup maupun terbuka dan tidak memiliki anak (daun dalam pohon saat ini), maka simpul tersebut dapat diperluas. Kita dapat menerapkan aturan transformasi untuk DNF pada salah satu rumus dalam simpul yang dapat diperluas untuk membuat anak. Perhatikan bahwa aturan tersebut hanya berlaku untuk operator logika teratas dari suatu rumus. Perluasan berhenti ketika pohon tidak memiliki simpul yang dapat diperluas.
- Tautan pohon menunjukkan relasi kesetaraan di antara himpunan formula. Jika sebuah simpul x memiliki dua penerus, misalnya y dan z , himpunan formula x setara dengan disjungsi dari dua himpunan formula y dan z . Aturan transformasi yang digunakan untuk mendapatkan dua penerus disebut aturan β . Jika x hanya memiliki satu penerus, aturan yang digunakan disebut aturan α , dan himpunan formula x setara dengan himpunan formula simpul penerus.

Berikut ini, kita akan menggunakan pohon dan tabel sebagai sinonim. Untuk membuktikan ketidakpuasan suatu formula, metode ini dimulai dengan menghasilkan pohon seperti yang dijelaskan di atas dan memeriksa bahwa setiap simpul daun tertutup, yaitu, tidak ada simpul terbuka dan tidak ada simpul yang dapat diperluas sebagai daun. Dalam hal ini, kita mengatakan pohon tersebut tertutup.

Contoh 3.1.2 Untuk formula A menjadi $p \wedge (\neg q \vee \neg p)$ dan B menjadi $(p \vee q) \wedge (\neg p \wedge \neg q)$, tabelnya ditunjukkan pada Gambar 3.1. Untuk A , tabelnya memiliki satu simpul terbuka (ditandai dengan $\odot$) dan satu simpul tertutup (ditandai dengan $\times$). Untuk B , hanya memiliki dua simpul tertutup. Dengan demikian, A dapat dipenuhi karena memiliki simpul terbuka; dan B tidak dapat dipenuhi karena tabelnya tertutup.

Tabel dapat ditampilkan secara linier, menggunakan string di atas $\{1, 2\}$ untuk memberi label hubungan induk-anak: Simpul akar memiliki label kosong. Jika sebuah simpul memiliki label x , anak pertamanya memiliki label $x1$ dan anak keduanya memiliki label $x2$, dan seterusnya. Misalnya, kita dapat menampilkan tabel A dan B pada Gambar 3.1 sebagai berikut:



Gambar 3.1 Tabel untuk $A = p \wedge (\neg q \vee \neg p)$ dan $B = (p \vee q) \wedge (\neg p \wedge \neg q)$

	$p \wedge (\neg q \vee \neg p)$
1 :	$p, (\neg q \vee \neg p)$
11 :	$p, \neg q \odot$
12 :	$p, \neg p \times$

(a) Tabel A

	$(p \vee q) \wedge (\neg p \wedge \neg q)$
1 :	$(p \vee q), (\neg p \wedge \neg q)$
11 :	$(p \vee q), \neg p, \neg q$
111 :	$p, \neg p, \neg q \times$
112 :	$q, \neg p, \neg q \times$

(a) Tabel B

Label pada node dapat secara unik mengidentifikasi posisi (atau alamat) node tersebut dalam sebuah tabel.

Aturan α dan Aturan β

Berikut adalah daftar aturan α dan aturan β , yang berlaku untuk operator logika teratas dalam sebuah rumus.

α	α_1, α_2
$A \wedge B$	A, B
$\neg(A \vee B)$	$\neg A, \neg B$
$\neg(A \rightarrow B)$	$A, \neg B$
$\neg(A \oplus B)$	$(A \vee \neg B), (\neg A \vee B)$
$A \leftrightarrow B$	$(A \vee \neg B), (\neg A \vee B)$
$\neg(A \uparrow B)$	A, B
$A \downarrow B$	$\neg A, \neg B$
$\neg\neg A$	A

β	β_1	β_2
$\neg(A \wedge B)$	$\neg A$	$\neg B$
$A \vee B$	A	B
$A \rightarrow B$	$\neg A$	B
$A \oplus B$	$A, \neg B$	$\neg A, B$
$\neg(A \leftrightarrow B)$	$A, \neg B$	$\neg A, B$
$A \uparrow B$	$\neg A$	$\neg B$
$\neg(A \downarrow B)$	A	B

Aturan-aturan tersebutlah yang kita gunakan untuk mengubah suatu rumus menjadi DNF (Negation Normal Form). Jika kita memulai dengan rumus dalam NNF (Negation Normal Form), kita hanya membutuhkan satu aturan α untuk $A \wedge B$ dan satu aturan β untuk $A \vee B$. Semua aturan α dan β lainnya tidak diperlukan untuk NNF, karena dari NNF ke DNF, hanya hukum distributif yang dibutuhkan.

Untuk setiap aturan α , kita mengubah rumus α menjadi α_1, α_2 ; untuk setiap aturan β , kita mengubah rumus β menjadi β_1 dan β_2 . Mudah untuk memeriksa bahwa $\alpha \equiv \alpha_1 \wedge \alpha_2$ dan $\beta \equiv \beta_1 \vee \beta_2$. Karena tableau mempertahankan sifat bahwa himpunan formula dari node saat ini setara dengan disjungsi dari himpunan formula anak-anaknya, tableau adalah representasi

seperti pohon dari sebuah formula yang setara dengan disjungsi dari semua himpunan formula node daunnya, yaitu, DNF, dengan asumsi himpunan formula setara dengan konjungsi dari anggotanya.

Contoh 3.1.3 Untuk menunjukkan bahwa formula Frege, $(p \rightarrow (q \rightarrow r)) \rightarrow (p \rightarrow q) \rightarrow (p \rightarrow r)$, valid, kita menunjukkan bahwa tableau semantik dari negasinya tertutup.

	$\neg((p \rightarrow (q \rightarrow r)) \rightarrow (p \rightarrow q) \rightarrow (p \rightarrow r))$	$\alpha \neg \rightarrow$
1:	$(p \rightarrow (q \rightarrow r)), \neg(p \rightarrow q) \rightarrow (p \rightarrow r)$	$\alpha \neg \rightarrow$
11:	$(p \rightarrow (q \rightarrow r)), (p \rightarrow q), \neg(p \rightarrow r)$	$\alpha \neg \rightarrow$
111:	$(p \rightarrow (q \rightarrow r)), (p \rightarrow q), p, \neg r$	$\beta \rightarrow$
1111:	$\neg p, (p \rightarrow q), p, \neg r$	$\times$
1112:	$(q \rightarrow r), (p \rightarrow q), p, \neg r$	$\beta \rightarrow$
11121:	$(q \rightarrow r), \neg p, p, \neg r$	$\times$
11122:	$(q \rightarrow r), q, p, \neg r$	$\beta \rightarrow$
111221:	$\neg q, q, p, \neg r$	$\times$
111222:	$r, q, p, \neg r$	$\times$

Pada Bab 1, kita telah menunjukkan bahwa ada dua sifat penting yang berkaitan dengan prosedur pembuktian: kebenaran dan kelengkapan. Jika tabel semantik digunakan sebagai pembuktian sanggahan, kebenaran berarti bahwa jika prosedur tersebut mengatakan bahwa rumus tersebut tidak dapat dipenuhi, maka rumus tersebut harus tidak dapat dipenuhi. Kelengkapan berarti bahwa jika rumus tersebut tidak dapat dipenuhi, maka prosedur tersebut harus dapat memberikan hasil ini kepada kita.

Teorema 3.1.4 Metode tabel semantik adalah prosedur pengambilan keputusan untuk logika proposisional.

Bukti Misalkan tanpa mengurangi keumuman, himpunan rumus pada satu simpul adalah $\{X, Y, Z\}$, di mana X mewakili rumus α , Y rumus β , dan Z rumus lainnya, dan himpunan ini setara dengan $A = X \wedge Y \wedge Z$. Jika X ditransformasikan oleh aturan α menjadi X_1 dan X_2 , maka dari $X \equiv X_1 \wedge X_2, A \equiv X_1 \wedge X_2 \wedge Y \wedge Z$. Jika Y ditransformasikan oleh aturan β menjadi Y_1 dan Y_2 , maka dari $Y \equiv Y_1 \vee Y_2, A \equiv (X \wedge Y_1 \wedge Z) \vee (X \wedge Y_2 \wedge Z)$. Dengan kata lain, kesetaraan logis dipertahankan untuk setiap tautan induk-anak dalam pohon.

Dengan induksi sederhana pada struktur pohon, rumus di akar pohon setara dengan disjungsi himpunan rumus dari semua simpul daun. Artinya, prosedur tersebut memodifikasi pohon sedemikian rupa sehingga disjungsi himpunan formula dari semua simpul daun pada tabel yang dihasilkan setara dengan formula masukan. Salah satu himpunan formula ini dapat berisi sepasang literal komplementer, yaitu, simpul yang bersesuaian tertutup, dalam hal ini himpunan tersebut setara dengan false. Jika himpunan semua simpul daun setara dengan

false, yaitu, setiap simpul utama tertutup, formula masukan tidak dapat dipenuhi. Ini menunjukkan kebenaran metode tabel semantik.

Jika suatu formula tidak dapat dipenuhi, maka setiap istilah dalam DNF-nya harus setara dengan false dan simpul yang bersesuaian dalam tabel tertutup. Dengan demikian, kita akan memiliki tabel tertutup. Ini menunjukkan kelengkapan metode tabel semantik.

Karena baik aturan α maupun aturan β tidak dapat diterapkan selamanya (setiap aturan mengurangi satu kejadian operator biner atau dua negasi), metode tabel semantik harus berakhir.

Metode tabel semantik dapat digunakan untuk menunjukkan kepuasan suatu formula. Jika terdapat simpul terbuka dalam tabel, kita dapat menetapkan nilai 1 untuk setiap literal dalam simpul ini untuk membuat interpretasi, dan himpunan rumus dari simpul ini bernilai benar di bawah interpretasi ini. Rumus masukan juga akan bernilai benar di bawah interpretasi ini karena rumus masukan setara dengan disjungsi dari himpunan rumus semua simpul daun. Dengan demikian, interpretasi ini merupakan model dari rumus masukan. Ini menunjukkan bahwa prosedur ini dapat digunakan untuk menemukan model ketika rumus tersebut dapat dipenuhi.

Tabel semantik menentukan aturan apa yang dapat digunakan tetapi tidak menentukan aturan mana yang harus digunakan terlebih dahulu atau simpul daun mana yang harus diperluas terlebih dahulu. Secara umum, aturan α lebih baik diterapkan sebelum aturan β , sehingga jumlah simpul dalam pohon tumbuh perlahan. Pengguna memiliki kebebasan untuk menerapkan aturan β pada rumus apa pun dalam himpunan, menghasilkan pohon dengan bentuk yang berbeda. Prosedur dapat berakhir setelah simpul terbuka ditemukan, jika tujuannya adalah untuk menunjukkan bahwa rumus masukan dapat dipenuhi.

Tabel semantik jauh lebih ekspresif dan lebih mudah digunakan daripada tabel kebenaran, meskipun bukan itu alasan diperkenalkannya tabel semantik. Terkadang, tabel semantik menggunakan lebih banyak ruang daripada tabel kebenaran. Misalnya, CNF penuh yang tidak dapat dipenuhi dengan n variabel akan menghasilkan tabel semantik dengan lebih dari $n!$ node, yang lebih besar daripada tabel kebenaran dengan 2^n baris. Keindahan tabel semantik terletak pada kesederhanaan penyajian prosedur pembuktian menggunakan seperangkat aturan. Kita akan melihat di bab-bab selanjutnya bagaimana aturan baru ditambahkan ke dalam prosedur ini sehingga prosedur pembuktian untuk logika lain dapat diperoleh.

3.2 SISTEM DEDUKTIF

Dalam logika, sistem deduktif S terdiri dari seperangkat aturan inferensi, di mana setiap aturan mengambil premis sebagai masukan dan mengembalikan kesimpulan (atau kesimpulan) sebagai keluaran. Aturan inferensi populer dalam logika proposisional meliputi modus ponens (MP), modus tollens (MT), dan kontraposisi (CP), yang masing-masing dapat ditampilkan sebagai berikut:

$$\frac{A \rightarrow B \quad A}{B} \text{ (MP)}$$

$$\frac{A \rightarrow B \quad \neg B}{\neg A} \text{ (MT)}$$

$$\frac{A \rightarrow B}{\neg B \rightarrow \neg A} \text{ (CP)}$$

Aturan Inferensi dan Bukti

Secara umum, aturan inferensi dapat ditentukan sebagai

$$\frac{P_1 \quad P_2 \quad \dots \quad P_k}{C}$$

di mana $P_1, P_2, \dots, P_k$ adalah premis dan C adalah kesimpulan. Kita juga dapat menuliskannya sebagai

$$P_1, P_2, \dots, P_k \vdash C$$

Jika premisnya kosong ($k = 0$), kita katakan aturan inferensi ini adalah aturan aksioma.

Suatu aturan inferensi dikatakan sah jika premis-premisnya menyiratkan kesimpulan, yaitu,

$$\{P_1, P_2, \dots, P_k\} \models C$$

untuk setiap aturan inferensi " $P_1, P_2, \dots, P_k \vdash C$ " (atau secara ekuivalen, $(P_1 \wedge P_2 \wedge \dots \wedge P_k) \rightarrow C$ adalah tautologi). Misalnya, MP (modus ponens) sah karena $\{A \rightarrow B, A\} \models B$; MT (modus tollens) sah karena $\{A \rightarrow B, \neg B\} \models \neg A$.

Diberikan himpunan A dari aksioma, yang merupakan rumus-rumus yang diasumsikan benar, bukti rumus B dalam S adalah urutan rumus $F_1, F_2, \dots, F_n$, sedemikian sehingga $F_n = B$ dan setiap F_i adalah rumus dalam A atau dapat dihasilkan oleh aturan inferensi S , menggunakan rumus-rumus sebelum F_i dalam urutan tersebut sebagai premis.

Jika bukti seperti itu ada, kita menunjuknya dengan $A \vdash_S B$; indeks S dapat dihilangkan jika S dipahami dari konteksnya. Prosedur pembuktian $P(A, B)$ berdasarkan S hanyalah menunjukkan $A \vdash_S B$.

Jelas, ini adalah prosedur pembuktian bergaya deduksi.

Contoh 3.2.1 Misalkan A adalah himpunan aksioma $\{p \rightarrow (q \rightarrow r), p, \neg r\}$, dan $S = \{MP, CT\}$. Bukti $A \vdash_S \neg q$ diberikan di bawah ini:

1. $p \rightarrow (q \rightarrow r)$ axiom
2. p axiom
3. q MP, 1, 2
4. $\neg r$ axiom
5. $\neg q$ MT, 3, 4

di mana " $MP, 1, 2$ " berarti rumus 3 diperoleh dengan aturan MP dengan rumus 1 dan 2 sebagai premis.

Rumus-rumus dalam sebuah pembuktian dapat disusun ulang sehingga semua aksioma muncul di awal urutan. Misalnya, pembuktian di atas dapat ditulis ulang sebagai berikut.

1.	$p \rightarrow (q \rightarrow r)$	axiom
2.	p	axiom
3.	$\neg r$	axiom
4.	$q \rightarrow r$	MP, 1, 2
5.	$\neg q$	MT, 3, 4

Setiap bukti dapat direpresentasikan oleh graf berarah di mana setiap simpul adalah rumus dalam bukti, dan jika rumus B diturunkan dari $A_1, A_2, \dots, A_k$ oleh aturan inferensi, maka terdapat sisi (A_i, B) , $1 \leq i \leq k$, dalam graf tersebut. Graf bukti ini harus asiklik karena kita mengasumsikan premis harus ada dalam urutan sebelum rumus baru diturunkan oleh aturan inferensi.

Setiap pengurutan topologi simpul dalam graf ini seharusnya memberi kita bukti linier. Karena setiap graf berarah asiklik (DAG) dapat diubah menjadi pohon (dengan menduplikasi beberapa simpul) dan tetap mempertahankan hubungan induk-anak, kita juga dapat menyajikan bukti dengan pohon. Pohon bukti untuk contoh bukti di atas diberikan pada Gambar 3.2. Perhatikan bahwa semua aksioma tidak memiliki sisi masuk. Rumus akhir dalam bukti tidak memiliki sisi keluar. Semua rumus perantara memiliki sisi masuk dan keluar.

Keabsahan suatu sistem inferensi berasal dari keabsahan semua aturan inferensi. Secara umum, kita mensyaratkan semua aturan inferensi harus valid untuk mempertahankan kebenaran semua rumus turunan. Untuk sistem inferensi valid S , setiap rumus turunan B adalah teorema dari A , yaitu, Jika A adalah himpunan tautologi, maka B juga merupakan tautologi.

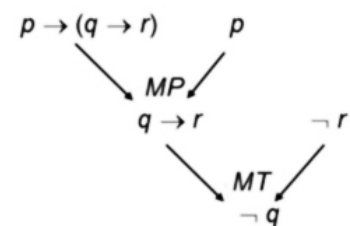
Teorema 3.2.2 Jika suatu sistem inferensi S valid dan $A \vdash_S B$, maka $A \models B$.

Bukti $A \vdash_S B$ berarti ada bukti B dalam S . Misalkan buktinya adalah $F_1, F_2, \dots, F_n$ dan $F_n = B$. Dengan induksi pada n , kita menunjukkan bahwa $A \models F_i$ untuk semua $i = 1, \dots, n$. Jika $F_i \in A$, maka $A \models F_i$. Jika F_i diturunkan dari aturan inferensi " $P_1, P_2, \dots, P_m \vdash C$ ", menggunakan $F_{j_1}, F_{j_2}, \dots, F_{j_m}$ sebagai premis, dengan induksi hipotesis, untuk $1 \leq k \leq m$, $A \models F_{j_k}$, karena $j_k < i$. Artinya, $A \models \bigwedge_{k=1}^m F_{j_k}$.

Karena aturan inferensi tersebut sah, maka $\bigwedge_{k=1}^m P_k \models C$. Dengan menerapkan teorema substitusi, $\bigwedge_{k=1}^m F_{j_k} \models F_i$. Karena $\models$ bersifat transitif, maka $A \models F_i$.

Sistem inferensi yang berbeda diperoleh dengan mengubah aksioma atau aturan inferensi. Dalam logika proposisional, semua sistem ini setara dalam arti bahwa semuanya sah dan lengkap.

Sistem Hilbert



Gambar 3.2 Pohon pembuktian dari $\neg q$

Sistem Hilbert, kadang-kadang disebut kalkulus Hilbert, atau sistem inferensi gaya Hilbert, adalah jenis sistem deduksi yang dikaitkan dengan Gottlob Frege dan David Hilbert. Varian sistem Hilbert untuk logika proposisional ada dan sistem deduktif ini diperluas ke logika orde pertama dan logika lainnya.

Definisi 3.2.3 Sistem Hilbert H berisi tiga aturan aksioma ditambah MP (modus ponens):

$$\begin{aligned} H_1 &: \vdash (A \rightarrow (B \rightarrow A)) \\ H_2 &: \vdash (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)), \\ H_3 &: \vdash (\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B), \\ MP &: A, A \rightarrow B \vdash B \end{aligned}$$

Aturan aksioma H_3 tidak diperlukan jika $\rightarrow$ adalah satu-satunya operator dalam semua rumus. Operator logika lainnya dapat didefinisikan menggunakan $\rightarrow$ dan $\neg$. Misalnya, $A \wedge B \equiv \neg(A \rightarrow \neg B)$ dan $A \vee B \equiv (\neg A \rightarrow B)$. $\neg A$ dapat digantikan oleh $A \rightarrow \perp$ dalam H_3 .

Contoh 3.2.4 Untuk menunjukkan bahwa $\vdash_H (p \rightarrow p)$, kita memiliki bukti berikut dalam H :

$$\begin{array}{ll} 1. & p \rightarrow ((p \rightarrow p) \rightarrow p) & H_1, A = p, B = (p \rightarrow p) \\ 2. & (p \rightarrow ((p \rightarrow p) \rightarrow p)) & H_2, A = C = p, B = p \rightarrow p \\ & ((p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p)) \\ 3. & (p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p) & MP, 1, 2 \\ 4. & (p \rightarrow (p \rightarrow p)) & H_1, A = p, B = p \\ 5. & p \rightarrow p & MP, 3, 4 \end{array}$$

Semua tautologi dapat diturunkan dalam sistem Hilbert.

Proposisi 3.2.5 Sistem Hilbert H adalah sah: Jika $A \vdash_H B$, maka $A \models B$.

Bukti Mudah untuk memeriksa bahwa setiap aturan inferensi H adalah sah. Artinya, setiap aturan aksioma $H_i, i = 1, 2, 3$, adalah tautologi, sehingga setiap contoh aturan tersebut adalah tautologi. Untuk $MP, (p \wedge (p \rightarrow q)) \rightarrow q$ adalah tautologi. Berdasarkan Teorema 3.2.2, proposisi tersebut berlaku.

Sistem Hilbert awalnya mengasumsikan bahwa $A = \emptyset$ (atau $A \equiv T$); dengan demikian, setiap rumus turunan adalah tautologi dalam sebuah bukti. Pembatasan ini dilonggarkan saat ini sehingga kita dapat mengambil sekumpulan rumus sebagai aksioma. Kebenaran rumus menjadi bersyarat pada kebenaran aksioma.

Contoh 3.2.6 Misalkan $X = \{p \rightarrow q, q \rightarrow r\}$; untuk menunjukkan bahwa $X \vdash_H (p \rightarrow r)$, kita memiliki bukti berikut di H :

$$\begin{array}{ll} 1. & p \rightarrow q & \text{axiom} \\ 2. & q \rightarrow r & \text{axiom} \end{array}$$

3. $((q \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r)))$	H_1
4. $p \rightarrow (q \rightarrow r)$	$MP, 2, 3$
5. $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$	H_2
6. $(p \rightarrow q) \rightarrow (p \rightarrow r)$	$MP, 4, 5$
7. $p \rightarrow r$	$MP, 1, 6$

Karena H adalah sistem yang valid, dari $A \vdash_H (p \rightarrow r)$, kita memiliki $(p \rightarrow q) \wedge (q \rightarrow r) \models_H (p \rightarrow r)$, atau $((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$ adalah valid.

Sistem Hilbert adalah salah satu sistem inferensi paling awal dan memiliki dampak besar pada penciptaan banyak sistem inferensi. Namun, pembuktian teorema dalam sistem ini selalu dianggap sebagai tantangan karena kompleksitasnya yang tinggi. Karena sistem Hilbert hampir tidak dapat menjadi alat praktis untuk pemecahan masalah, kami menyatakan tanpa bukti hasil kelengkapan sistem Hilbert.

Teorema 3.2.7 Sistem Hilbert valid dan lengkap, yaitu, $A \models B$ jika dan hanya jika $A \vdash_H B$.

Deduksi Alami

Baik tabel semantik maupun sistem Hilbert untuk membangun bukti memiliki kekurangan masing-masing. Sistem Hilbert sulit untuk membangun bukti dan penggunaan utamanya bersifat metalogis. Jumlah aturan yang sedikit membuatnya lebih mudah untuk membuktikan kebenaran logika. Metode tabel di sisi lain mudah digunakan secara mekanis, tetapi, karena bentuk aturan dan fakta bahwa tabel dimulai dari negasi rumus yang akan dibuktikan, bukti yang dihasilkan bukanlah urutan langkah-langkah yang mudah dibenarkan secara alami. Demikian pula, sangat sedikit bukti dalam matematika yang berasal langsung dari aksioma. Para matematikawan dalam praktiknya biasanya bernalar dengan cara yang lebih fleksibel.

Deduksi alami adalah sistem deduktif di mana penalaran logis diungkapkan oleh aturan inferensi yang terkait erat dengan cara penalaran "alami". Ini berbeda dengan sistem Hilbert, yang justru menggunakan aturan aksioma sebanyak mungkin untuk mengungkapkan hukum logika penalaran deduktif. Deduksi alamiah dalam bentuk modernnya secara independen diusulkan oleh matematikawan Jerman Gerhard Gentzen pada tahun 1934.

Dalam sistem deduksi alamiah Gentzen, sebuah rumus diwakili oleh himpunan rumus $A, A = \{A_1, A_2, \dots, A_n\}$, di mana koma dalam A dipahami sebagai $\vee$. Untuk menghindari kebingungan, kita akan menulis A sebagai

$$(A_1 | A_2 | \dots | A_n)$$

A disebut sebagai sekuen oleh Gentzen. Jika $n = 0$, maka $A = ()$, yang setara dengan $\perp$. Karena A adalah himpunan, rumus yang berulang dihilangkan dan urutan rumus dalam A tidak relevan.

Definisi 3.2.8 Suatu sistem deduktif alami G terdiri dari tiga aturan inferensi berikut, diikuti oleh aturan logika yang memandu pengenalan dan penghapusan operator logika dalam sekuen.

Nama	aturan inferensi
<i>axiom</i>	$\vdash (A \mid \neg A \mid \alpha)$
<i>cut</i>	$(A \mid \alpha), (\neg A \mid \beta) \vdash (\alpha \mid \beta)$
<i>thinning</i>	$(\alpha) \vdash (A \mid \alpha)$

Operator	Pengenalan	Penghapusan
$\neg$	$(A \mid \alpha) \vdash (\neg \neg A \mid \alpha)$	$(\neg \neg A \mid \alpha) \vdash (A \mid \alpha)$
$\vee$	$(A \mid B \mid \alpha) \vdash (A \vee B \mid \alpha)$	$(A \vee B \mid \alpha) \vdash (A \mid B \mid \alpha)$
$\wedge$	$(A \mid \alpha), (B \mid \alpha) \vdash (A \wedge B \mid \alpha)$	(a) $(A \wedge B \mid \alpha) \vdash (A \mid \alpha)$ (b) $(A \wedge B \mid \alpha) \vdash (B \mid \alpha)$
$\rightarrow$	$(\neg A \mid B \mid \alpha) \vdash (A \rightarrow B \mid \alpha)$	$(A \rightarrow B \mid \alpha) \vdash (\neg A \mid B \mid \alpha)$
$\neg \vee$	$(\neg A \mid \alpha), (\neg B \mid \alpha) \vdash (\neg(A \vee B) \mid \alpha)$	(a) $(\neg(A \vee B) \mid \alpha) \vdash (\neg A \mid \alpha)$ (b) $(\neg(A \vee B) \mid \alpha) \vdash (\neg B \mid \alpha)$
$\neg \wedge$	$(\neg A \mid \neg B \mid \alpha) \vdash (\neg(A \wedge B) \mid \alpha)$	$(\neg(A \wedge B) \mid \alpha) \vdash (\neg A \mid \neg B \mid \alpha)$
$\neg \rightarrow$	$(A \mid \alpha), (\neg B \mid \alpha) \vdash (\neg(A \rightarrow B) \mid \alpha)$	(a) $(\neg(A \rightarrow B) \mid \alpha) \vdash (A \mid \alpha)$ (b) $(\neg(A \rightarrow B) \mid \alpha) \vdash (\neg B \mid \alpha)$

di mana α dan β , mungkin kosong, menunjukkan sisa rumus dalam suatu sekuen.

Dalam deduksi alamiah, suatu sekuen disimpulkan dari sekumpulan aksioma dengan menerapkan aturan inferensi berulang kali. Setiap sekuen disimpulkan dari sekuen lain pada baris sebelumnya dalam suatu bukti sesuai dengan aturan inferensi, memberikan perkiraan yang lebih baik terhadap gaya pembuktian yang digunakan oleh para matematikawan daripada sistem Hilbert.

Contoh 3.2.9 Berikut ini adalah bukti dari $\neg(p \wedge q) \rightarrow (\neg p \vee \neg q)$ dalam G :

1. $(p \mid \neg p \mid \neg q)$	<i>axiom</i> , $A = p, \alpha = \neg q$
2. $(q \mid \neg q \mid \neg p)$	<i>axiom</i> , $A = q, \alpha = \neg p$
3. $(p \wedge q \mid \neg p \mid \neg q)$	$\wedge_I, 1, 2, A = p, B = q, \alpha = \neg p \mid \neg q$
4. $(\neg \neg(p \wedge q) \mid \neg p \mid \neg q)$	$\neg_I, 4, A = (p \wedge q), \alpha = (\neg p \vee \neg q)$
5. $(\neg \neg(p \wedge q) \mid (\neg p \vee \neg q))$	$\vee_I, 3, A = \neg p, B = \neg q, \alpha = \neg \neg(p \wedge q)$
6. $(\neg(p \wedge q) \rightarrow (\neg p \vee \neg q))$	$\rightarrow_I, 5, A = \neg(p \wedge q), B = (\neg p \vee \neg q), \alpha = \perp$

Perhatikan bahwa I dalam $\wedge_I$ berarti aturan pengenalan untuk $\wedge$ digunakan. Untuk pemahaman yang lebih baik, kami menambahkan substitusi variabel di setiap langkah.

Proposisi 3.2.10 Sistem deduktif G adalah sah, yaitu, jika $A \vdash_G B$, maka $A \models B$.

Bukti Mudah untuk memeriksa bahwa setiap aturan inferensi G adalah sah. Secara khusus, aturan pengenalan operator mempertahankan kesetaraan logis. Dalam aturan eliminasi operator, beberapa sekuen menghasilkan dua hasil. Misalnya, $(A \wedge B \mid \alpha)$ menyimpulkan (a) $(A \mid \alpha)$ dan (b) $(B \mid \alpha)$. Jika kita mengikat kedua hasil tersebut bersama-sama dengan $\wedge$, aturan-aturan ini juga mempertahankan kesetaraan logis. Di sisi lain, aturan pemotongan dan aturan penipisan tidak mempertahankan kesetaraan logis; hanya relasi implikasi yang berlaku. Artinya, untuk aturan pemotongan, $(A \mid \alpha), (\neg A \mid \beta) \vdash (\alpha \mid \beta)$, kita memiliki

$$(A \vee \alpha) \wedge (\neg A \vee \beta) \models \alpha \vee \beta$$

dan untuk aturan penipisan, $(\alpha) \vdash (A \mid \alpha)$, kita memiliki $\alpha \models A \vee \alpha$.

Perhatikan bahwa bukti dalam contoh di atas hanya menggunakan aturan aksioma (dua kali) dan aturan pengenalan untuk operator logika. Ini adalah kasus ketika kita mencari bukti tautologi. Ingat bagaimana kita akan membuktikan bahwa $\neg(p \wedge q) \rightarrow (\neg p \vee \neg q)$ adalah tautologi dengan tabel semantik. Karena tabel semantik adalah pembukti penyangkalan, kita perlu menunjukkan bahwa negasi dari tautologi memiliki tabel tertutup.

id	label	Node Rumus	Aturan
a.	:	$\neg(\neg(p \wedge q) \rightarrow (\neg p \vee \neg q))$	$\alpha \neg \rightarrow$
b.	1:	$\neg(p \wedge q), \neg(\neg p \vee \neg q)$	$\alpha \neg \vee$
c.	11:	$\neg(p \wedge q), \neg\neg p, \neg\neg q$	$\alpha \neg$
d.	111:	$\neg(p \wedge q), p, \neg\neg q$	$\alpha \neg$
e.	1111:	$\neg(p \wedge q), p, q$	$\beta \neg \wedge$
f.	11111:	$\neg p, p, q$	$\times$
g.	11112:	$\neg q, p, q$	$\times$

Dengan membandingkan bukti G dan tabel di atas, kita dapat melihat korespondensi yang jelas: Node tertutup (f) dan (g) berkorespondensi dengan (1) dan (2), yang merupakan aksioma: (e) dengan (3), (c) dan (d) dengan (4), (b) dengan (5), dan akhirnya (a) dengan (6), masing-masing. Artinya, rumus yang diwakili oleh setiap sekuen dalam bukti G selalu dapat menemukan negasinya di node yang sesuai dari tabel semantik.

Kami akan menunjukkan di bawah ini bahwa G dapat digunakan sebagai pembukti sanggahan dengan contoh yang sama.

Contoh 3.2.11 Misalkan $A = \{\neg(\neg(p \wedge q) \rightarrow (\neg p \vee \neg q))\}$. Bukti $A \models \perp$ dalam G diberikan di bawah ini.

1. $(\neg(\neg(p \wedge q) \rightarrow (\neg p \vee \neg q)))$	<i>assumed</i>
2. $(\neg(p \wedge q))$	$\neg \rightarrow_E (a), 1, A = \neg(p \wedge q)$
3. $(\neg(\neg p \vee \neg q))$	$\neg \rightarrow_E (b), 1, B = (\neg p \vee \neg q)$
4. $(\neg p \mid \neg q)$	$\neg \wedge_E, 2, A = p, B = q$
5. $(\neg\neg p)$	$\neg \vee_E (a), 3, A = \neg p$
6. $(\neg\neg q)$	$\neg \vee_E (b), 3, B = \neg q$

7. (p)	$\neg_E, 5, A = p$
8. (q)	$\neg_E, 6, A = q$
9. $(\neg q)$	$cut, 4, 7, A = p, \alpha = \perp, \beta = \neg q$
10. $\perp$	$cut, 8, 9, A = q, \alpha = \perp, \beta = \perp$

Perhatikan bahwa $\perp$ pada langkah terakhir melambangkan sekuen kosong.

Dalam bukti ini, aturan inferensi yang digunakan adalah aturan pemotongan dan aturan untuk menghilangkan operator logika. Jika kita menerapkan aturan penghapusan operator berulang kali pada suatu formula, kita akan memperoleh daftar sekuen, yang masing-masing mewakili sebuah klausa. Dengan kata lain, aturan-aturan ini mengubah formula menjadi CNF, kemudian aturan pemotongan bekerja padanya. Aturan pemotongan untuk klausa memiliki nama lain, yaitu resolusi, yang merupakan topik bagian selanjutnya.

Memang, G mengandung lebih banyak aturan daripada yang diperlukan: Hanya dengan menggunakan aturan aksioma dan aturan untuk pengenalan operator, ia dapat meniru dual dari tableau semantik secara terbalik. Hanya dengan menggunakan aturan pemotongan dan aturan untuk penghapusan operator, ia dapat melakukan apa yang dapat dilakukan oleh pembukti resolusi. Baik tableau semantik maupun pembukti resolusi adalah prosedur keputusan untuk logika proposisional. Implikasinya adalah bahwa deduksi alami G juga merupakan prosedur keputusan. Kami memberikan hasil utama tanpa bukti.

Teorema 3.2.12 Deduksi alami G adalah sah dan lengkap, yaitu, $A \models B$ jika dan hanya jika $A \vdash_G B$.

Graf Inferensi

Kita telah melihat dua contoh sistem deduktif: sistem Hilbert H dan deduksi alami Gentzen G . Untuk menunjukkan kelengkapan sistem deduktif ini, kita membutuhkan strategi tentang bagaimana menggunakan aturan inferensi dalam sistem ini dan kelengkapannya bergantung pada strategi ini. Untuk mempermudah pembahasan, mari kita perkenalkan beberapa konsep.

Definisi 3.2.13 Diberikan sistem inferensi S , graf inferensi S di atas himpunan aksioma A didefinisikan sebagai graf berarah $G = (V, E)$, di mana setiap simpul V adalah himpunan x rumus, di mana $A \in V, x \subseteq T(A)$, dan $(x, y) \in E$ jika dan hanya jika $y = x \cup \{C\}$, di mana $C \notin x$ adalah hasil dari aturan inferensi $r \in S$ menggunakan beberapa rumus dalam x sebagai premis r .

Graf $G = (V, E)$ mendefinisikan ruang pencarian untuk $A \vdash_S B$, yang menjadi masalah pencarian: Menemukan bukti $A \vdash_S B$ sama dengan menemukan jalur dari A ke simpul V yang mengandung B . Artinya, jika semua aksioma A dalam suatu bukti muncul di awal urutan bukti, maka bukti ini menghadirkan jalur dalam graf inferensi. Misalkan bukti tersebut adalah urutan rumus $F_1, F_2, \dots, F_n$ sedemikian sehingga k rumus pertama berasal dari A , yaitu, $F_i \in A$ untuk

$1 \leq i \leq k$ dan $F_i \notin A$ untuk $i > k$. Maka bukti ini mewakili jalur terarah di $x_0, x_1, \dots, x_{n-k}$ dalam graf inferensi sedemikian sehingga $x_0 = A, F_i \in A$ untuk $1 \leq i \leq k$, dan $F_j \in x_{j-k}$ untuk $k < j \leq n$. Dengan kata lain, bukti adalah cara ringkas untuk menyajikan jalur terarah dalam grafik inferensi G .

Contoh 3.2.14 Dalam Contoh 3.2.1, bukti kedua disalin di sini:

1. Aksioma	$p \rightarrow (q \rightarrow r)$
2. Aksioma	p
3. Aksioma	$\neg r$
4. MP, 1, 2	$q \rightarrow r$
5. MT, 3, 4	$\neg q$

Bukti ini sesuai dengan jalur (x_0, x_1, x_2) dalam grafik inferensi, di mana $x_0 = \{1., 2., 3.\}$, $x_1 = x_0 \cup \{4.\}$, dan $x_2 = x_1 \cup \{5.\}$.

Diberikan seperangkat aturan inferensi, prosedur pembuktian dapat dengan mudah dibangun jika kita menggunakan strategi yang adil untuk mencari grafik inferensi. Misalnya, dimulai dari A dalam grafik inferensi, pencarian breadth-first adalah strategi yang adil, tetapi pencarian depth-first tidak adil jika grafiknya tak terbatas. Strategi yang adil mensyaratkan bahwa jika suatu aturan inferensi dapat diterapkan pada suatu titik, penerapan ini akan terjadi pada akhirnya. Strategi pencarian yang adil apa pun, termasuk pencarian heuristik, pencarian terbaik pertama, A*, dll., dapat digunakan untuk prosedur pembuktian.

Jika setiap aturan inferensi valid, prosedur pembuktian yang sesuai akan valid. Persyaratan tambahan seperti strategi yang adil diperlukan secara umum untuk menunjukkan bahwa prosedur pembuktian lengkap atau berhenti. Jika grafik inferensi terbatas, penghentian prosedur pembuktian dijamin, tetapi ini tidak menyiratkan kelengkapan prosedur karena kurangnya aturan inferensi tertentu.

3.3 RESOLUSI

Pada bagian sebelumnya, kita telah memperkenalkan dua sistem deduktif, sistem Hilbert dan deduksi alamiah. Kedua sistem ini merupakan contoh untuk pengembangan banyak sistem deduktif tetapi tidak praktis untuk pembuktian teorema dalam logika proposisional. Salah satu alasannya adalah penggunaan aturan aksioma yang memberikan kemungkinan tak terbatas. Tidak seperti aturan reduksi, aturan inferensi dapat diterapkan tanpa penghentian. Untuk pembuktian teorema otomatis, kita membutuhkan sistem deduktif yang berisi sangat sedikit aturan karena komputer dapat melakukan hal-hal sederhana dengan cepat. Resolusi R adalah sistem seperti itu, yang berisi satu aturan inferensi tunggal, yang disebut resolusi. Perhatikan bahwa resolusi adalah kasus khusus dari aturan pemotongan ketika semua sekuen adalah klausa.

Aturan Resolusi

Untuk menggunakan aturan resolusi, rumus A pertama-tama diubah menjadi CNF, yang merupakan konjungsi klausa (disjungsi literal). Kita akan menggunakan $A = \{C_1, C_2, \dots, C_m\}$ untuk menunjukkan konjungsi klausa; setiap klausa C_i dilambangkan dengan

$$(l_1 | l_2 | \dots | l_k),$$

di mana setiap l_j adalah literal (variabel proposisional atau negasinya). Jika $k = 0$, klausa C_i kosong, dilambangkan dengan $\perp$; jika $k = 1$, C_i disebut klausa unit; jika $k = 2$, C_i disebut klausa biner. Suatu klausa dikatakan positif jika tidak kosong dan hanya berisi literal positif; dikatakan negatif jika tidak kosong dan hanya berisi literal negatif. Suatu klausa adalah tautologi jika berisi sepasang literal komplementer, yaitu, berisi p dan $\neg p$ untuk beberapa variabel p .

Jumlah total klausa, yang literal duplikatnya dihilangkan, pada n variabel proposisional dibatasi oleh 3^n , karena setiap variabel memiliki salah satu dari tiga kasus dalam klausa non-tautologi: hilang, sebagai literal positif, atau sebagai literal negatif.

Secara formal, aturan resolusi didefinisikan oleh skema berikut, di mana A adalah variabel proposisional, α dan β adalah literal sisa dari kedua klausa sebagai premis, dan hasilnya adalah klausa baru $(\alpha | \beta)$.

$$\frac{(A | \alpha) \quad (\neg A | \beta)}{(\alpha | \beta)}$$

Alternatifnya, kita dapat menulis aturan resolusi sebagai berikut:

$$(A | \alpha), (\neg A | \beta) \vdash (\alpha | \beta)$$

Klausa $(\alpha | \beta)$ yang dihasilkan oleh aturan resolusi disebut resolven dengan meresolusi A dari C_1 dan C_2 . C_1 dan C_2 adalah induk dari resolven. Aturan resolusi ini juga disebut resolusi biner, karena melibatkan dua klausa sebagai premis.

Aturan resolusi untuk logika proposisional dapat ditelusuri kembali ke Davis dan Putnam (1960). Resolusi diperluas ke logika orde pertama pada tahun 1965 oleh John Alan Robinson, menggunakan algoritma unifikasi. Jika kita menggunakan $\rightarrow$ sebagai pengganti $|$ dalam aturan resolusi, kita memperoleh

$$\frac{(\neg \alpha \rightarrow A) \quad (A \rightarrow \beta)}{(\neg \alpha \rightarrow \beta)}$$

yang merupakan transitivitas dari $\rightarrow$. Jika α kosong, maka kita peroleh MP:

$$\frac{A \quad A \rightarrow \beta}{\beta}$$

Contoh 3.3.1 Misalkan $C = \{(p \mid q), (p \mid \neg q), (\neg p \mid q \mid r), (\neg p \mid \neg q), (\neg r)\}$. Bukti $C \vdash_R \perp$ diberikan di bawah ini:

1. $(p \mid q)$	<i>axiom</i>
2. $(p \mid \neg q)$	<i>axiom</i>
3. $(\neg p \mid q \mid r)$	<i>axiom</i>
4. $(\neg p \mid \neg q)$	<i>axiom</i>
5. $(\neg r)$	<i>axiom</i>
6. $(q \mid r)$	$R, 1, 3, A = p, \alpha = q, \beta = (q \mid r)$
7. $(\neg q)$	$R, 2, 4, A = p, \alpha = \neg q, \beta = \neg q$
8. (r)	$R, 6, 7, A = q, \alpha = r, \beta = \perp$
9. $()$	$R, 8, 5, A = r, \alpha = \perp, \beta = \perp$

Perhatikan bahwa kita mengasumsikan bahwa literal yang duplikat dihapus dan urutan literal tidak relevan dalam sebuah klausa. Dengan demikian, resolven yang dihasilkan dari klausa 1 dan 3 adalah $(q \mid q \mid r)$, tetapi kita tetap mempertahankannya sebagai $(q \mid r)$ dengan menghapus salinan q . Dari klausa 1 dan 4, kita dapat memperoleh resolven $(q \mid \neg q)$ pada p , atau resolven lain $(p \mid \neg p)$ pada q . Kedua resolven ini adalah tautologi dan tidak berguna dalam pencarian $\perp$. Anda tidak dapat melakukan resolusi pada dua variabel secara bersamaan untuk mendapatkan $()$ secara langsung dari klausa 1 dan 4.

Definisi 3.3.2 Bukti resolusi P dari himpunan klausa masukan C adalah bukti sanggahan dari $C \vdash_R \perp$ dan panjang P adalah jumlah resolven dalam P , yang dilambangkan dengan $|P|$. Himpunan klausa masukan yang muncul dalam P disebut inti dari P , yang dilambangkan dengan $core(P)$.

Panjang bukti resolusi dalam Contoh 3.3.1 adalah 4, yaitu panjang daftar dikurangi jumlah klausa aksioma dalam bukti. Inti dari bukti ini adalah seluruh masukan.

Proposisi 3.3.3 Resolusi R adalah sah, yaitu, jika $C \vdash_R B$, maka $C \models B$.

Bukti Dengan menggunakan tabel kebenaran, mudah untuk memeriksa bahwa $((A \vee \alpha) \wedge (\neg A \vee \beta)) \rightarrow (\alpha \vee \beta)$ adalah tautologi. Dengan demikian, aturan resolusi adalah sah. Proposisi ini berlaku berdasarkan Teorema 3.2.2.

Korolari 3.3.4 Jika $C \vdash_R \perp$, maka C tidak dapat dipenuhi.

Bukti Karena $C \vdash_R \perp$ berarti $C \models \perp$, maka $M(C) = M(C \wedge \perp) = M(C) \cap M(\perp) = \emptyset$, karena $M(\perp) = \emptyset$ (lihat Korolari 2.2.26).

Korolari ini memungkinkan kita untuk merancang pembuktian sanggahan: Untuk menunjukkan bahwa A valid, kita mengubah $\neg A$ menjadi himpunan klausa C yang setara dan menunjukkan bahwa $C \vdash_R \perp$, di mana C disebut klausa masukan. Karena jumlah total klausa terbatas, grafik inferensi R terbatas. Ini berarti bahwa prosedur pembuktian akan berakhir tetapi tidak berarti grafiknya kecil. Sebelum kita menetapkan kelengkapan R , mari kita pertimbangkan bagaimana membangun pembuktian resolusi yang efektif.

Implementasi pembuktian resolusi yang sukses membutuhkan integrasi strategi pencarian yang menghindari banyak resolven yang tidak perlu. Beberapa strategi menghilangkan klausa redundan segera setelah muncul dalam derivasi. Beberapa strategi menghindari pembuatan klausa redundan sejak awal. Beberapa strategi mengorbankan kelengkapan demi efisiensi.

Strategi Resolusi

Berikut adalah beberapa strategi resolusi yang terkenal yang menambahkan batasan pada penggunaan aturan resolusi.

- *Resolusi unit*: Salah satu dari dua klausa induk adalah klausa unit.
- *Resolusi masukan*: Salah satu dari dua klausa induk adalah klausa masukan (diberikan sebagai aksioma).
- *Resolusi terurut*: Diberikan urutan pada variabel proposisional, variabel yang diresolusi harus maksimal di kedua klausa induk.
- *Resolusi positif*: Salah satu dari dua klausa induk bersifat positif.
- *Resolusi negatif*: Salah satu dari dua klausa induk bersifat negatif.
- *Himpunan pendukung*: Klausa masukan dipartisi menjadi dua himpunan, S dan T , di mana S disebut himpunan pendukung, dan resolusi tidak diperbolehkan jika kedua klausa induk berada di T .
- *Resolusi linier*: Resolven terbaru digunakan sebagai induk untuk resolusi berikutnya (tidak ada batasan pada resolusi pertama).

Bukti resolusi disebut bukti resolusi X jika setiap resolusi dalam bukti tersebut adalah resolusi X , di mana X adalah “unit,” “input,” “terurut,” “positif,” “negatif,” “himpunan pendukung,” atau “linier.”

Contoh 3.3.5 Misalkan $C = \{1. (p \mid q), 2. (\neg p \mid r), 3. (p \mid \neg q \mid r), 4. (\neg r)\}$. Dalam bukti-bukti berikut, klausa input dihilangkan dan induk dari setiap resolven ditunjukkan sebagai sepasang angka yang mengikuti resolven tersebut.

(a)		(b)		(c)	
5. $(\neg p)$	(2, 4)	5. $(p \mid r)$	(1, 3)	5. $(q \mid r)$	(1, 2)
6. $(p \mid \neg q)$	(3, 4)	6. (r)	(2, 5)	6. $(\neg q \mid r)$	(2, 3)
7. $(\neg q)$	(5, 6)	7. $(\)$	(4, 6)	7. (r)	(5, 6)
8. (q)	(1, 5)			8. $(\)$	(4, 7)
9. $(\)$	(7, 8)				

Perhatikan bahwa (a) adalah bukti resolusi unit dan negatif; (b) adalah bukti resolusi input, positif, dan linier; (c) adalah bukti resolusi terurut, dengan asumsi $p > q > r$. Panjang bukti (a) adalah 5; panjang (b) adalah 3; dan panjang (c) adalah 4. Mudah untuk memeriksa bahwa (a) bukan bukti resolusi input, karena klausa 7 tidak diperoleh dengan resolusi input. (b) bukan bukti resolusi unit. (c) bukan bukti resolusi unit maupun input; (c) bukan bukti resolusi positif maupun negatif.

Efisiensi merupakan pertimbangan penting dalam penalaran otomatis, dan terkadang seseorang mungkin bersedia mengorbankan kelengkapan demi kecepatan. Dalam resolusi unit, salah satu klausa induk selalu merupakan klausa unit; dalam resolusi input, salah satu klausa induk selalu dipilih dari himpunan input. Meskipun efisien, tidak ada strategi yang lengkap.

Misalnya, $C = \{(p \mid q), (p \mid \neg q), (\neg p \mid q), (\neg p \mid \neg q)\}$ tidak dapat dipenuhi, meskipun tidak ada resolusi unit yang tersedia. Kita dapat menggunakan resolusi input untuk mendapatkan p dan $\neg p$ tetapi tidak dapat memperoleh $()$ karena p dan $\neg p$ bukan klausa input.

Resolusi terurut memberlakukan pengurutan total pada variabel proposisional dan memperlakukan setiap klausa bukan sebagai himpunan literal tetapi sebagai daftar literal yang diurutkan. Resolusi terurut efisien dan lengkap.

Resolusi himpunan pendukung adalah salah satu strategi ampuh yang digunakan oleh Wos, Carson, dan Robinson pada tahun 1965, dan kelengkapannya bergantung pada pilihan himpunan pendukung. Untuk membuktikan $A \models B$ dengan resolusi, kita mengubah $A \wedge \neg B$ menjadi klausa. Klausa yang diperoleh dari $\neg B$ pada umumnya berfungsi sebagai himpunan pendukung S dan klausa yang berasal dari A berada di T . Resolusi himpunan pendukung menentukan bahwa induk dari setiap klausa yang diselesaikan tidak keduanya berasal dari T . Motivasi di balik resolusi himpunan pendukung adalah karena A biasanya dapat dipenuhi, mungkin bijaksana untuk tidak menyelesaikan dua klausa yang awalnya berasal dari A .

Perlu dicatat bahwa beberapa strategi meningkatkan aspek-aspek tertentu dari proses deduksi dengan mengorbankan aspek lainnya. Misalnya, suatu strategi dapat mengurangi ukuran ruang pencarian bukti dengan mengorbankan peningkatan panjang bukti terpendek.

Mempertahankan Keterpuasan

Aturan resolusi tidak mempertahankan kesetaraan logis, yaitu, jika D adalah resolven dari C_1 dan C_2 , maka $C_1 \wedge C_2 \models D$, atau $C_1 \wedge C_2 \equiv C_1 \wedge C_2 \wedge D$, tetapi bukan $C_1 \wedge C_2 \equiv D$.

Contoh 3.3.6 Misalkan q adalah resolven dari p dan $\neg p \mid q$. $M(q) = \{\bar{p}q, pq\}$ memiliki dua model, di mana $\bar{p}$ menunjukkan $\neg p$. Namun, $M(p \wedge (\neg p \mid q))$ hanya memiliki satu model, yaitu pq . Dengan demikian, $q \not\models (p \wedge (\neg p \mid q))$. Namun, kedua rumus tersebut dapat dipenuhi.

Definisi 3.3.7 Diberikan dua rumus A dan B , A dan B dikatakan sama-sama dapat dipenuhi atau ekuisatisfiable, jika setiap kali A dapat dipenuhi, maka B juga dapat dipenuhi, dan sebaliknya. Kita menyatakan hubungan ini dengan $A \approx B$.

Jelas, $A \approx B$ berarti $M(A) = \emptyset$ jika dan hanya jika $M(B) = \emptyset$. $A \approx B$ lebih lemah daripada kesetaraan logis $A \equiv B$, yang mensyaratkan $M(A) = M(B)$.

Proposisi 3.3.8 Misalkan $S = C \cup \{(\alpha \mid \beta)\}$ dan $S' = C \cup \{(\alpha \mid x), (\neg x \mid \beta)\}$ adalah dua himpunan klausa, dimana x adalah variabel yang tidak muncul dalam S . Maka $S \approx S'$.

Bukti Jika S memuaskan dan σ adalah model dari S , maka $\sigma(C) = 1$ dan $\sigma(\alpha \mid \beta) = 1$. Jika $\sigma(\alpha) = 0$, maka $\sigma(\beta) = 1$. Kita definisikan $\sigma(x) = 1$ jika $\sigma(\alpha) = 0$; jika tidak $\sigma(x) = 0$. Jadi, $\sigma(\alpha \mid x) = 1$ dan $\sigma(\neg x \mid \beta) = 1$. Jadi σ adalah model dari S' .

Di sisi lain, jika σ adalah model dari S' , maka σ juga merupakan model dari S tanpa modifikasi, karena $(\alpha \mid \beta)$ adalah resolven dari $(\alpha \mid x)$ dan $(\neg x \mid \beta)$.

Dalam proposisi di atas, syarat bahwa x tidak muncul dalam C adalah syarat yang diperlukan. Tanpa syarat tersebut, misalnya, jika $C = \{(\neg p)\}$, $\alpha = p$, $\beta = q$ dan $x = p$, maka $S = \{(\neg p), (q)\}$ dan $S' = \{(\neg p), (p), (\neg p \mid q)\}$. Maka S dapat dipenuhi tetapi S' tidak.

Dengan menggunakan proposisi di atas, kita dapat memecah klausa panjang menjadi serangkaian klausa yang lebih pendek dengan memperkenalkan beberapa variabel baru. Misalnya, klausa $(l_1 \mid l_2 \mid l_3 \mid l_4 \mid l_5)$ dapat ditransformasikan menjadi tiga klausa: $(l_1 \mid l_2 \mid x)$, $(\neg x \mid l_3 \mid y)$, dan $(\neg y \mid l_4 \mid l_5)$. Transformasi ini tidak mempertahankan kesetaraan logis, tetapi mempertahankan kepuasan.

Penggunaan lain dari proposisi di atas adalah menggunakan resolusi untuk menghapus x dari himpunan klausa. Jika kita menggunakan semua resolusi pada x , menyimpan semua resolven dan menghapus semua klausa yang mengandung x , maka himpunan klausa yang dihasilkan akan mempertahankan kepuasan, seperti yang akan kita lihat pada Lemma 3.3.12.

Proposisi 3.3.9 Misalkan B adalah subformula dari A dan x adalah variabel baru. Maka $A \approx (A[B \leftarrow x] \wedge (x \leftrightarrow B))$.

Bukti Jika A dapat dipenuhi dan σ adalah model dari A , maka kita mendefinisikan $\sigma(x) = \sigma(B)$. Ketika menerapkan σ pada pohon formula A dan $A[B \leftarrow x]$, semua node umum memiliki nilai kebenaran yang sama, sehingga $\sigma(A) = \sigma(A[B \leftarrow x])$. Karena $\sigma(x) = \sigma(B)$, maka $\sigma(x \leftrightarrow B) = 1$, sehingga σ adalah model dari $A[B \leftarrow x] \wedge (x \leftrightarrow B)$.

Di sisi lain, jika σ adalah model dari $A[B \leftarrow x] \wedge (x \leftrightarrow B)$, maka σ juga merupakan model dari A . Proposisi di atas memainkan peran penting ketika kita mengubah formula menjadi CNF dan menjaga ukuran CNF sebagai fungsi linier dari ukuran formula masukan.

Ketika kita menggunakan hukum distributif

$$(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$$

Untuk mengubah formula menjadi CNF, C menjadi duplikat dalam hasilnya. Karena itu, ada contoh di mana CNF yang dihasilkan memiliki ukuran eksponensial dari ukuran input. Dengan menggunakan proposisi di atas dan memperkenalkan variabel baru, kita dapat menjaga ukuran CNF sebagai empat kali ukuran input, dan setiap klausa berisi paling banyak tiga literal. CNF yang diperoleh dengan cara ini tidak secara logis setara dengan formula input, tetapi mempertahankan kepuasan formula input: CNF dapat dipenuhi jika dan hanya jika formula input dapat dipenuhi. Cara mengubah formula menjadi CNF ini disebut *transformasi Tseitin*. Mulai sekarang, kita akan menggunakan $\bar{p}$ untuk $\neg p$ jika bentuk yang ringkas lebih disukai.

Contoh 3.3.10 Misalkan A adalah $(p \wedge (q \vee \neg p)) \vee \neg(\neg q \wedge (p \vee q))$. Kita menelusuri pohon formula A dari bawah ke atas untuk mengganti setiap subformula yang simbol teratasnya adalah operator biner dengan variabel baru dan memperkenalkan formula $\leftrightarrow$ untuk setiap variabel baru. Kemudian kita mengubah setiap formula $\leftrightarrow$ menjadi paling banyak tiga klausa dan setiap klausa berisi paling banyak tiga literal.

$$\begin{array}{ll} x_1 \leftrightarrow (q \vee \neg p) & (\bar{x}_1 \mid q \mid \bar{p}), (\bar{q} \mid x_1), (p \mid x_1), \\ x_2 \leftrightarrow (p \wedge x_1) & (\bar{x}_2 \mid p), (\bar{x}_2 \mid x_1), (\bar{p} \mid \bar{x}_1 \mid x_2), \\ x_3 \leftrightarrow (p \vee q) & (\bar{x}_3 \mid p \mid q), (\bar{p} \mid x_3), (\bar{q} \mid x_3), \\ x_4 \leftrightarrow (\neg q \wedge x_3) & (\bar{x}_4 \mid q), (\bar{x}_4 \mid q), (q \mid \bar{x}_3 \mid x_4), \\ x_2 \vee \neg x_4 & (x_2 \mid \bar{x}_4). \end{array}$$

Ukuran input A adalah 14 (tidak termasuk koma dan tanda kurung). Jumlah total literal dari himpunan klausa yang dihasilkan adalah 30.

Contoh 3.3.11 Dalam Contoh 2.3.9, kita telah mengubah $p_1 \leftrightarrow (p_2 \leftrightarrow p_3)$ menjadi empat klausa:

$$(p_1 \mid p_2 \mid p_3), (p_1 \mid p_2 \mid p_3), (p_1 \mid p_2 \mid p_3), (p_1 \mid p_2 \mid p_3)$$

Jika kita ingin mengubah $p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow p_4))$ menjadi klausa, kita dapat mengganti p_3 dengan $(p_3 \leftrightarrow p_4)$ dalam klausa di atas dan kemudian mengubah masing-masingnya menjadi klausa. Artinya, dari $(\bar{p}_1 \mid \bar{p}_2 \mid (p_3 \leftrightarrow p_4))$, $(\bar{p}_1 \mid p_2 \mid (\bar{p}_3 \leftrightarrow \bar{p}_4))$, $(p_1 \mid p_2 \mid (p_3 \leftrightarrow p_4))$, dan $(p_1 \mid \bar{p}_2 \mid (\bar{p}_3 \leftrightarrow \bar{p}_4))$, kita memperoleh delapan klausa, dua kali lipat dari sebelumnya:

$$\begin{array}{l} (\bar{p}_1 \mid \bar{p}_2 \mid \bar{p}_3 \mid p_4), (\bar{p}_1 \mid p_2 \mid p_3 \mid p_4), (\bar{p}_1 \mid \bar{p}_2 \mid p_3 \mid \bar{p}_4), (\bar{p}_1 \mid p_2 \mid \bar{p}_3 \mid \bar{p}_4), \\ (p_1 \mid p_2 \mid \bar{p}_3 \mid p_4), (p_1 \mid \bar{p}_2 \mid p_3 \mid p_4), (p_1 \mid p_2 \mid p_3 \mid \bar{p}_4), (p_1 \mid \bar{p}_2 \mid \bar{p}_3 \mid \bar{p}_4) \end{array}$$

Jika kita mengganti p_4 dengan $(p_4 \leftrightarrow p_5)$, kita akan mendapatkan 16 klausa dengan panjang 5. Secara umum, $p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (\dots (p_{n-1} \leftrightarrow p_n) \dots)))$ akan menghasilkan 2^{n-1} klausa dengan panjang n . Sebaliknya, jika kita memperkenalkan $n - 3$ variabel baru untuk rumus n variabel ini, kita hanya memperoleh $4(n - 1)$ klausa dengan panjang 3. Misalnya, untuk $n = 5$, kita memperkenalkan dua variabel baru, q_1 dan q_2 :

$$(p_1 \leftrightarrow (p_2 \leftrightarrow q_1)) \wedge (q_1 \leftrightarrow (p_3 \leftrightarrow q_2)) \wedge (q_2 \leftrightarrow (p_4 \leftrightarrow q_5))$$

yang akan menghasilkan $3 \times 4 = 12$ klausa dengan panjang 3.

Lemma 3.3.12 Misalkan C adalah himpunan klausa dan x adalah variabel dalam C . Misalkan $D \subset C$ sedemikian sehingga setiap klausa D tidak mengandung x maupun $\bar{x}$. Misalkan $X = \{(\alpha \mid \beta) : (x \mid \alpha), (x \mid \beta) \in C\}$, maka $C \approx D \cup X$.

Bukti Jika C dapat dipenuhi dan σ adalah model dari C , maka $\sigma(C) = 1$ dan $\sigma(D) = 1$. Jika $\sigma(X) = 0$, maka terdapat klausa $(\alpha \mid \beta) \in X$ sedemikian sehingga $\sigma(\alpha) = \sigma(\beta) = 0$. Ini berarti $\sigma(x \mid \alpha) = 0$ atau $\sigma(\bar{x} \mid \beta) = 0$, yang merupakan kontradiksi dengan fakta bahwa $\sigma(C) = 1$, karena baik $(x \mid \alpha)$ maupun $(\bar{x} \mid \beta)$ berada di C . Jadi $\sigma(X) = 1$. Ini berarti σ adalah model dari $D \cup X$.

Di sisi lain, jika $D \cup X$ dapat dipenuhi dan σ adalah modelnya, kita dapat mendefinisikan $\sigma(x) = 1$ jika $\sigma(\alpha) = 0$ untuk beberapa $x \mid \alpha \in C$; dalam hal ini, kita harus memiliki $\sigma(\beta) = 1$ untuk semua $(\bar{x} \mid \beta) \in C$; jika tidak, jika $\sigma(\beta) = 0$ untuk beberapa $(\bar{x} \mid \beta) \in C$, maka $\sigma(\alpha \mid \beta) = 0$, yang merupakan kontradiksi dengan fakta bahwa $\sigma(X) = 1$. Demikian pula, jika $\sigma(\alpha) = 1$ untuk semua $x \mid \alpha \in C$, kita mendefinisikan $\sigma(x) = 0$. Maka σ menjadi model dari C .

Kelengkapan Resolusi

Lemma 3.3.12 memainkan peran penting dalam pembuktian kelengkapan resolusi: Himpunan X dapat diperoleh dari C melalui resolusi. Dari C ke $D \cup X$, kita telah menghilangkan satu variabel dari C . Mengganti C dengan $D \cup X$, kita mengulangi proses yang sama dan menghilangkan variabel lain, hingga tidak ada lagi variabel di C : baik $C = \emptyset$ atau $C = \{\perp\}$. Dari hasil akhir, kita dapat mengetahui apakah himpunan klausa masukan dapat dipenuhi atau tidak. Bukti berikut didasarkan pada gagasan di atas untuk kelengkapan resolusi terurut.

Teorema 3.3.13 Resolusi terurut lengkap dengan urutan total apa pun pada variabel.

Bukti Misalkan ada n variabel dalam klausa masukan S_n dengan urutan $x_n > x_{n-1} > \dots > x_2 > x_1$. Untuk i dari n sampai 1, kita menerapkan Lemma 3.3.12 dengan $C = S_i$ dan $x = x_i$, dan memperoleh $S_{i-1} = D \cup X$. Setiap resolven dalam X diperoleh dengan resolusi terurut pada x_i , yang merupakan variabel maksimal dalam S_i . Akhirnya, kita memperoleh S_0 yang

tidak mengandung variabel apa pun. Karena $S_n \approx S_{n-1} \approx \dots \approx S_1 \approx S_0$, jika S_n tidak dapat dipenuhi, maka $S_0 = \{\perp\}$, di mana $\perp$ menunjukkan klausa kosong; jika S_n dapat dipenuhi, kita sampai pada $S_0 = \emptyset$, yaitu, $S_0 \equiv T$. Dalam hal ini, kita dapat menggunakan ide dalam pembuktian Lemma 3.3.12 untuk menetapkan nilai kebenaran untuk setiap variabel, dari x_1 sampai x_n , untuk memperoleh model S_n .

Contoh 3.3.14 Misalkan $S = \{c_1 : (\bar{a} \mid \bar{b} \mid \bar{c}), c_2 : (a \mid \bar{e}), c_3 : (b \mid \bar{c}), c_4 : (\bar{b} \mid d), c_5 : (c \mid \bar{d} \mid \bar{e}), c_6 : (\bar{d} \mid e)\}$ dan $a > b > c > d > e$. Dengan resolusi terurut, kita mendapatkan $c_7 : (\bar{b} \mid \bar{c} \mid \bar{e})$ dari c_1 dan c_2 ; $c_8 : (\bar{c} \mid d)$ dari c_3 dan c_4 ; $c_9 : (\bar{c} \mid \bar{e})$ dari c_3 dan c_7 ; dan $c_{10} : (\bar{d} \mid \bar{e})$ dari c_5 dan c_9 . Tidak ada klausa lain yang dapat dihasilkan oleh resolusi terurut, yaitu, $c_1 - c_{10}$ jenuh oleh resolusi terurut. Karena klausa kosong tidak dihasilkan, kita dapat membangun model dari $c_1 - c_{10}$ dengan menetapkan nilai kebenaran pada variabel dari yang terkecil hingga yang terbesar. Misalkan S_x adalah himpunan klausa dari $c_1 - c_{10}$ sedemikian sehingga x muncul di setiap klausa S_x sebagai variabel maksimal.

- $S_e = \emptyset$ dan kita dapat menetapkan 0 atau 1 untuk e , misalnya $\sigma_e = \{e\}$.
- $S_d = \{c_6, c_{10}\}$. Dari c_{10} , $\bar{d}$ harus 1, jadi $\sigma_d = \sigma_e \cup \{\bar{d}\} = \{e, \bar{d}\}$.
- $S_c = \{c_5, c_8, c_9\}$. Dari c_8 atau c_9 , $\bar{c}$ harus 1, jadi $\sigma_c = \sigma_d \cup \{\bar{c}\} = \{e, \bar{d}, \bar{c}\}$.
- $S_b = \{c_3, c_4, c_7\}$. Baik c_3 dan c_7 dipenuhi oleh σ_c . Dari c_4 , kita harus menetapkan 0 untuk b , sehingga $\sigma_b = \sigma_c \cup \{\bar{b}\} = \{e, \bar{d}, \bar{c}, \bar{b}\}$.
- $S_a = \{c_1, c_2\}$. Dari c_2 , a harus bernilai 1, sehingga $\sigma_a = \sigma_b \cup \{a\} = \{e, \bar{d}, \bar{c}, \bar{b}, a\}$.

Mudah untuk memeriksa bahwa σ_a adalah model dari S . Oleh karena itu, S dapat dipenuhi.

Contoh di atas mengilustrasikan bahwa jika sekumpulan klausa dijenuhkan oleh resolusi terurut dan klausa kosong tidak ada, kita dapat membangun model dari klausa-klausa ini. Dengan kata lain, kita dapat menggunakan resolusi terurut untuk membangun prosedur pengambilan keputusan untuk menentukan apakah suatu formula A dapat dipenuhi atau tidak.

Algoritma 3.3.15 Algoritma `orderedResolution` akan mengambil himpunan C klausa dan daftar V variabel, $V = (p_1 < p_2 < \dots < p_n)$. Algoritma ini mengembalikan nilai true jika dan hanya jika C dapat dipenuhi.

Algoritma ini menggunakan prosedur $sort(A)$, yang menempatkan literal maksimal dari klausa A sebagai literal pertama dalam A .

```

proc orderedResolution( $C, V$ )
   $C := \{sort(A) \mid A \in C\}$ 

```

```

for  $i := n$  downto 1 do
   $X := \emptyset$ 
  for  $(p_i \mid \alpha) \in C$  do //  $p_i$  adalah maksimal dalam  $(p_i \mid \alpha)$ 
    for  $(\bar{p}_i \mid \beta) \in C$  do //  $p_i$  adalah maksimal dalam  $(\bar{p}_i \mid \beta)$ 
       $A := (\alpha \mid \beta)$ 
      if  $A = ()$  return false
       $X := X \cup \{\text{sort}(A)\}$ 
   $C := C \cup X$ 
return true

```

Sebelum mengembalikan *true* pada baris terakhir, algoritma ini dapat memanggil *createModel* untuk membangun model C ketika C jenuh oleh resolusi terurut. Artinya, C berisi hasil dari setiap kemungkinan resolusi terurut di antara klausa-klausa C .

```

proc createModel( $C$ )
   $\sigma := \emptyset$  // interpretasi kosong
  for  $i := 1$  to  $n$  do
     $v := 0$  // nilai default untuk  $p_i$ 
    for  $(p_i \mid \alpha) \in C$  do //  $p_i$  adalah nilai maksimal
      if  $\text{eval}(\alpha, \sigma) = 0$  do
         $v := 1$  //  $p_i$  harus bernilai 1
     $\sigma := \sigma \cup \{p_i \rightarrow v\}$ 
  return  $\sigma$ 

```

Algoritma resolusi dianggap sebagai prosedur pembuktian gaya deduksi ketika klausa kosong adalah tujuannya. Jika menemukan model adalah tujuan dari algoritma resolusi, itu adalah prosedur pembuktian gaya saturasi, karena himpunan klausa dijenuhkan oleh resolusi terurut, yaitu, semua kemungkinan hasil dari aturan inferensi dihasilkan, sebelum kita dapat membangun model.

Kolala 3.3.16 Resolusi adalah prosedur pengambilan keputusan yang sah dan lengkap untuk logika proposisional.

Bukti Kita telah melihat bahwa resolusi itu sah. Jika C tidak dapat dipenuhi, kita dapat memperoleh bukti resolusi terurut dari C . Bukti ini juga merupakan bukti resolusi. Dengan demikian, $C \models \perp$ menyiratkan $C \vdash_R \perp$. Jika C dapat dipenuhi, kita dapat membangun model untuk C seperti yang kita lakukan untuk resolusi terurut.

Teorema 3.3.17 Resolusi positif lengkap untuk penyangkalan.

Bukti Misalkan C tidak dapat dipenuhi dan ada bukti resolusi $C \vdash_R \perp$. Kita buktikan dengan induksi pada panjang bukti tentang bagaimana mengubah bukti ini menjadi bukti resolusi positif.

Misalkan P adalah bukti resolusi dari $C \vdash_R \perp$ dengan panjang minimal. Jika $|P| = 1$, maka satu-satunya resolusi menggunakan klausa unit positif untuk memperoleh $\perp$ dan itu adalah resolusi positif. Jika $|P| > 1$, misalkan resolusi terakhir dalam P menyelesaikan variabel q . Maka $C \vdash_R q$ dan $C \vdash_R \bar{q}$. Misalkan P_1 dan P_2 adalah bukti dari $C \vdash_R q$ dan $C \vdash_R \bar{q}$ yang disajikan dalam P , masing-masing.

Jika kita menghapus semua kemunculan q dari P_1 , kita memperoleh bukti resolusi P'_1 dan $core(P'_1)$ berisi klausa dari $core(P)$ dengan q yang dihapus. Karena $|P'_1| < |P|$, berdasarkan hipotesis induksi, kita memiliki bukti resolusi positif dari $core(P'_1) \vdash_R \perp$. Jika kita menambahkan q kembali ke $core(P'_1)$, kita memperoleh bukti resolusi positif P''_1 dari $core(P) \vdash_R q$.

Kemudian kita menghapus semua kemunculan $\bar{q}$ dari P_2 dan memperoleh bukti resolusi P'_2 dan $core(P'_2)$ berisi klausa dari $core(P)$ dengan $\bar{q}$ yang telah dihapus. Berdasarkan hipotesis induksi, ada bukti resolusi positif P''_2 dari $core(P'_2) \vdash_R \perp$. Untuk setiap klausa $\alpha \in core(P'_2)$, jika α diperoleh dari $(\alpha | \bar{q}) \in core(P)$, kita menambahkan α sebagai resolven dari $(\alpha | \neq q)$ dan q di awal P''_2 . Akhirnya, kita menggabungkan P''_1 , yang menghasilkan q dengan resolusi positif, dan P''_2 menjadi satu bukti; hasilnya adalah bukti resolusi positif dari $C \vdash_R \perp$.

Hasil berikut dapat dibuktikan menggunakan pendekatan serupa.

Korolari 3.3.18 Resolusi negatif lengkap untuk penyangkalan.

Kelengkapan strategi resolusi penting untuk memastikan bahwa jika klausa kosong tidak ditemukan, klausa masukan dapat dipenuhi. Dalam hal ini, kita harus menghitung semua resolusi yang diizinkan sebelum menyatakan bahwa klausa masukan dapat dipenuhi. Kita mengatakan bahwa himpunan klausa, lama dan baru, jenuh oleh strategi resolusi, jika tidak ada resolven baru yang dapat dihitung oleh strategi resolusi yang diberikan. Resolusi terbatas seperti resolusi terurut, resolusi positif, dan resolusi himpunan pendukung, bekerja jauh lebih baik daripada resolusi tak terbatas, ketika saturasi diperlukan untuk menunjukkan kepuasan dari sekumpulan klausa.

Prosedur Pengambilan Keputusan Berbasis Resolusi

Jika kita ingin menggunakan heuristik dalam prosedur pengambilan keputusan berbasis resolusi, prosedur berikut sesuai dengan tujuan tersebut.

Algoritma 3.3.19 Algoritma resolusi akan mengambil himpunan C klausa dan mengembalikan *true* jika dan hanya jika C dapat dipenuhi. Algoritma ini menggunakan prosedur *resolvable(A,B)*, yang memutuskan apakah klausa A dan B diizinkan untuk melakukan resolusi terbatas, dan jika ya,

$resolve(A, B)$ adalah resolvennya. Algoritma ini menggunakan $pickClause(C)$ untuk memilih klausa sesuai dengan heuristik.

```

proc resolution( $C$ )
   $G := C$  //  $G$ : klausa yang diberikan
   $K := \emptyset$  //  $K$ : klausa yang dipertahankan
  while  $G \neq \emptyset$  do
     $A := pickClause(G)$ 
     $G := G - \{A\}$ 
    for  $B \in K$  if  $resolvable(A, B)$  do
       $res := resolve(A, B)$ 
      if  $res = ()$  if  $false$ 
      if  $res \notin (G \cup K)$ 
         $G := G \cup \{res\}$ 
       $K := K \cup \{A\}$ 
  return  $true$ 

```

Dalam algoritma di atas, kita memindahkan setiap klausa dari G ke K , dan menghitung resolusi antara klausa ini dan semua klausa di K . Ketika G kosong, resolusi antara dua klausa apa pun dari K dilakukan.

Dengan pengecualian resolusi linier, prosedur di atas dapat digunakan untuk mengimplementasikan semua strategi resolusi yang diperkenalkan di bagian ini. Pembatasan akan diimplementasikan dalam prosedur $resolvable$. Untuk resolusi himpunan, implementasi yang lebih baik adalah G diinisialisasi dengan himpunan dukungan dan K diinisialisasi dengan sisa klausa input.

Penghentian prosedur ini dijamin karena hanya ada sejumlah klausa yang terbatas. Ketika prosedur mengembalikan nilai $false$, jawabannya benar karena resolusinya valid. Ketika mengembalikan nilai $true$, kebenaran jawaban bergantung pada kelengkapan strategi resolusi yang diimplementasikan dalam $resolvable$.

Dalam $pickClause$, kita dapat mengimplementasikan berbagai heuristik untuk memilih klausa dalam G untuk melakukan resolusi dengan klausa dalam K . Heuristik populer termasuk lebih menyukai klausa yang lebih pendek atau klausa yang lebih lama. Lebih menyukai klausa yang lebih pendek karena klausa yang lebih pendek lebih kuat sebagai batasan pada penerimaan interpretasi sebagai model. Klausa kosong melarang semua interpretasi sebagai model; klausa unit melarang setengah dari interpretasi sebagai model; klausa biner melarang seperempat dari interpretasi sebagai model; dan seterusnya. Lebih menyukai klausa yang lebih lama saja akan memberi kita strategi *breadth – first*. Jika kita ingin menemukan bukti yang lebih pendek, kita dapat mencampur preferensi ini dengan heuristik lain.

Meskipun hasil bahwa resolusi algoritma dengan strategi lengkap dapat berfungsi sebagai prosedur pengambilan keputusan untuk logika proposisional, dan beberapa strategi

bahkan dapat membangun model ketika klausa masukan dapat dipenuhi, pembukti resolusi pada umumnya tidak efisien untuk logika proposisional, karena jumlah klausa yang mungkin bersifat eksponensial dalam hal jumlah variabel. Komputer dapat dengan cepat kehabisan memori sebelum menemukan solusi untuk masalah aplikasi nyata.

Strategi Penyederhanaan

Ada beberapa strategi yang dapat dengan aman menghilangkan klausa redundan di dalam algoritma resolusi. Strategi tersebut adalah penghapusan tautologi, penghapusan subsumsi, dan penghapusan literal murni.

Suatu klausa adalah tautologi jika mengandung pasangan literal yang saling melengkapi, misalnya, $(p \mid p)$. Klausa tautologi benar dalam setiap interpretasi dan tidak diperlukan dalam bukti resolusi apa pun.

Proposisi 3.3.20 Diberikan himpunan klausa S apa pun, misalkan S dipartisi menjadi $S' \cup T$, di mana T adalah himpunan klausa tautologi dalam S . Maka $S \equiv S'$.

Bukti Untuk interpretasi σ apa pun, $\sigma(S) = \sigma(S') \wedge \sigma(T) = \sigma(S')$ ketika $\sigma(T) = 1$.

Definisi 3.3.21 Suatu klausa C mencakup klausa lain B jika $B = (C \mid \alpha)$ untuk beberapa literal α . Artinya, setiap literal B muncul di C .

Misalnya, $(p \mid q)$ mencakup $(p \mid q \mid r)$. Secara intuitif, resolusi mencoba untuk menghilangkan literal dengan resolusi satu per satu untuk mendapatkan klausa kosong. Jika kita menggunakan $(p \mid q)$ alih-alih $(p \mid q \mid r)$, kita menghindari pekerjaan menghilangkan r .

Proposisi 3.3.22 Jika C mencakup B , maka $S \cup \{C, B\} \equiv S \cup \{C\}$.

Bukti Karena C mencakup B , misalkan B adalah $(C \mid \alpha)$, $M(C) \subseteq M(C) \cup M(\alpha) = M(B)$. $M(S \cup \{C, B\}) = M(S) \cap (M(C) \cap M(B)) = M(S) \cap M(C) = M(S \cup \{C\})$.

Proposisi di atas menunjukkan bahwa jika C mencakup B , maka B dapat dihilangkan sebagai kendala dengan adanya C . Proposisi berikut menunjukkan bahwa B tidak diperlukan dalam bukti resolusi.

Proposisi 3.3.23 Jika C dan B adalah dua klausa yang muncul dalam bukti resolusi dari $S \vdash_R \perp$ sedemikian sehingga C mencakup B , maka B tidak lagi diperlukan setelah C hadir.

Bukti Jika B digunakan dalam bukti resolusi setelah kehadiran C , kita dapat mengganti B dengan C untuk mendapatkan bukti yang lebih pendek atau sama panjangnya, dengan memeriksa semua resolusi yang menghilangkan setiap literal dari B : Misalkan resolusi pertama adalah antara B dan γ dan resolvennya adalah B' . Ada dua kasus yang perlu dipertimbangkan:

(1) Variabel yang diresolusi tidak ada di C . Resolusi ini tidak perlu, karena C mencakup B' , dan kita mengganti B' dengan C . (2) Variabel yang diresolusi ada di C , maka misalkan resolven dari C dan γ adalah C' . Mudah untuk memeriksa bahwa C' mencakup B' , dan kita mengganti B' dengan C' . Kita mengganti keturunan B dengan keturunan C yang sesuai, hingga resolusi terakhir. Bukti resolusi yang dimodifikasi akan memiliki panjang yang lebih pendek atau sama.

Proposisi 3.3.24 Jika ada bukti resolusi yang menggunakan klausa tautologi, maka ada bukti resolusi tanpa klausa tautologi tersebut.

Bukti Misalkan klausa tautologi yang muncul dalam bukti adalah $(p \mid \bar{p} \mid \alpha)$. Kita membutuhkan klausa lain $(\bar{p} \mid \beta)$ (atau $(p \mid \beta)$) untuk menyelesaikan p . Resolven dari $(p \mid \bar{p} \mid \alpha)$ dan $(\bar{p} \mid \beta)$ adalah $(\bar{p} \mid \alpha \mid \beta)$, yang tercakup oleh $(\bar{p} \mid \beta)$. Berdasarkan Proposisi 3.3.23, $(\bar{p} \mid \alpha \mid \beta)$ tidak diperlukan dalam bukti resolusi. Karena setiap resolven dari tautologi tidak diperlukan, kita tidak memerlukan tautologi apa pun dalam bukti resolusi.

Definisi 3.3.25 Suatu literal l dikatakan murni dalam himpunan C klausa jika komplementnya tidak muncul dalam C .

Proposisi 3.3.26 Jika l murni dalam C , misalkan D adalah himpunan klausa yang diperoleh dari C dengan menghapus semua klausa yang mengandung l dari C , maka $C \approx D$.

Bukti Jika σ adalah model dari C , maka σ juga merupakan model dari D . Jika σ adalah model dari D , definisikan $\sigma(l) = 1$, maka σ adalah model dari C .

Misalnya, jika $C = \{c_1 : (p \mid q), c_2 : (\bar{p} \mid r), c_3 : (p \mid \bar{q} \mid r), c_4 : (\bar{p} \mid q)\}$, maka r murni. Menghapus c_2 dan c_3 dari C , q menjadi murni. Menghapus c_1 dan c_4 , kita memperoleh himpunan klausa kosong, yang setara dengan T . Berdasarkan proposisi tersebut, artinya C dapat dipenuhi. Sebuah model dapat diperoleh dengan menetapkan $q = r = 1$ dalam model tersebut.

Proposisi di atas memungkinkan penggunaan strategi penghapusan klausa berikut tanpa perlu khawatir kehilangan bukti:

- *Penghapusan tautologi*: Klausa tautologi dibuang.
- *Penghapusan subsumsi*: Klausa yang disubsumsi dibuang.
- *Penghapusan murni*: Klausa yang berisi literal murni dibuang.

Strategi penghapusan ini dapat diintegrasikan ke dalam resolusi algoritma sebagai berikut.

Algoritma 3.3.27 Resolusi algoritma akan mengambil himpunan S klausa dan mengembalikan nilai benar jika dan hanya jika S dapat dipenuhi. Dalam *praproses*(S), kita dapat menerapkan penghapusan tautologi, pemeriksaan literal murni, atau subsumsi untuk menyederhanakan klausa

masuk sebagai langkah praproses. *subsumedBy(C,S)* memeriksa apakah klausa *C* tercakup oleh klausa di *S*.

```

proc resolution(S)
1   G := preprocessing(S) // G: given clauses
2   K :=  $\emptyset$  // K: kept clauses
3   while G  $\neq \emptyset$  do
4       C := pickClause(G)
5       G := G - {C}
6       N := {} // new clauses from C and K
7       for B  $\in$  K if resolvable(C,B) do
8           res := resolve(C,B)
9           if res = () return false
10          if res is tautology continue
11          if subsumedBy(res, G  $\cup$  K) continue
12          N := N  $\cup$  {res}
13      K := K  $\cup$  {C}
14      for C  $\in$  G if subsumedBy(C,N) do
15          G := G - {C}
16      for B  $\in$  K if subsumedBy(B,N) do
17          K := K - {B}
18      G := G  $\cup$  N
19  return true // K is saturated by resolution.

```

Siklus hidup klausa *C* dalam algoritma resolusi adalah sebagai berikut:

1. Jika *C* adalah klausa masukan, ia melewati pemeriksaan pra-pemrosesan, seperti pemeriksaan literal murni dan pemeriksaan subsumsi, dan masuk ke *G* (baris 1).
2. Jika *C* dipilih dari *G* (baris 4, 5), ia akan dipasangkan dengan setiap klausa *B* di *K* untuk mencoba resolusi pada *C* dan *B*. Setelah ini selesai, *C* akan masuk ke *K* sebagai klausa yang dipertahankan (baris 13).
3. *C* dapat disubsumsi oleh klausa baru ketika berada di *G* (baris 14) atau di *K* (baris 16) dan akan dibuang.
4. Jika *C* adalah klausa baru (baris 8), ia harus melewati pemeriksaan subsumsi (baris 11) dan masuk ke *N* (baris 12).
5. Semua klausa baru dalam *N* akan digunakan untuk memeriksa apakah mereka dapat mencakup beberapa klausa dalam *G* (baris 14, 16), kemudian bergabung ke dalam *G* (baris 18), dan memulai siklus hidup dengan cara yang sama seperti klausa masukan.
6. Jika algoritma berhenti tanpa menemukan klausa kosong (baris 9), himpunan *G* akan kosong (baris 3), dan semua resolusi di antara klausa dalam *K* selesai. Artinya, *K* jenuh dengan resolusi.

Dalam algoritma ini, klausa unit diberi prioritas tinggi untuk melakukan resolusi dan subsumsi unit. Penanganan klausa unit dapat dipisahkan dari loop utama algoritma seperti yang akan kita lihat di bagian selanjutnya.

3.4 PROPAGASI BATASAN BOOLEAN (BCP)

Klausa unit (A) mencakup setiap klausa ($A \mid \alpha$), di mana A muncul sebagai literal. Resolusi unit antara (A) dan ($\bar{A} \mid \beta$), di mana $\bar{A} = p$ jika A adalah $\bar{p}$, menghasilkan resolven (β), yang akan mencakup ($\bar{A} \mid \beta$). Aturan untuk mengganti ($\bar{A} \mid \beta$) dengan (β) dengan adanya (A) disebut penghapusan unit. Artinya, ketika penghapusan subsumsi digunakan dalam resolusi, klausa unit (A) memungkinkan kita untuk menghapus semua kemunculan A (dengan klausa yang mengandung A) dan $\bar{A}$ (hanya $\bar{A}$ dalam klausa yang mengandung A), dengan klausa unit itu sendiri sebagai satu-satunya pengecualian. Klausa unit baru dapat dihasilkan oleh resolusi unit, dan kita dapat terus menyederhanakan klausa dengan klausa unit baru dan seterusnya. Proses ini secara tradisional disebut propagasi kendala Boolean (BCP), atau propagasi unit.

BCP: Prosedur Penyederhanaan

Strategi penghapusan klausa memungkinkan kita untuk menghapus klausa yang tidak perlu selama resolusi. BCP juga memungkinkan kita untuk menyederhanakan klausa dengan menghapus klausa yang tidak perlu dan memperpendek klausa. Berikut ini, kita akan menjelaskan BCP sebagai algoritma yang mengambil himpunan C klausa sebagai input dan mengembalikan pasangan (U, S) , di mana U adalah himpunan klausa unit dan S adalah himpunan klausa non-unit sedemikian sehingga U dan S tidak memiliki variabel yang sama dan $C \equiv U \cup S$. Biasanya, S berisi subset dari C atau klausa dari C dengan menghapus beberapa literal.

Untuk mempermudah pembahasan, kita mengasumsikan bahwa tidak ada klausa yang mengandung literal duplikat (sehingga klausa unit tidak dapat disembunyikan sebagai $(p \mid p)$) dan tidak ada klausa yang merupakan tautologi. Kita juga mengasumsikan bahwa himpunan klausa unit tidak mengandung literal komplementer (sehingga klausa kosong tidak dapat disembunyikan sebagai (p) dan $(\neg p)$).

Algoritma 3.4.1 Algoritma BCP akan mengambil himpunan C klausa dan menerapkan resolusi unit dan penghapusan subsumsi berulang kali, hingga tidak ada lagi klausa baru yang dapat dihasilkan oleh resolusi unit. Fungsi ini akan mengembalikan $\perp$ jika klausa kosong ditemukan; jika tidak, fungsi ini akan mengembalikan serangkaian klausa yang disederhanakan dan setara dengan C .

```

proc BCP( $C$ )
   $S := C$  //  $S$ : simplified clauses
   $U := \emptyset$  //  $U$ : unit clauses
  while  $S$  has a unit clause ( $A$ ) do
     $S := S - \{(A)\}$ 
     $U := U \cup \{A\}$ 
    for  $(A \mid \alpha) \in S$  do
       $S := S - \{(A \mid \alpha)\}$ 
    for  $(\bar{A} \mid \alpha) \in S$  do
      if  $\alpha = ()$  return  $\perp$ 

```

$S := S - \{(\bar{A} \mid \alpha)\} \cup \{(\alpha)\}$
return (U, S)

Contoh 3.4.2 Let $C = \{c_1: (x_2 \mid x_5), c_2: (\bar{x}_1 \mid \bar{x}_4), c_3: (\bar{x}_2 \mid x_4), c_4: (x_1 \mid x_2 \mid \bar{x}_3), c_5: (\bar{x}_5)\}$. Then $BCP(C)$ will return
 $(\{\bar{x}_5, x_2, x_4, \bar{x}_1\}, \emptyset)$

Karena pertama-tama kita menambahkan $\bar{x}_5$ ke U ; c_1 menjadi (x_2) , dan x_2 kemudian ditambahkan ke U ; c_4 dihapus karena $x_2 \in U$; c_3 menjadi (x_4) dan x_4 ditambahkan ke U ; c_2 menjadi $(\bar{x}_1)$ dan $\bar{x}_1$ ditambahkan ke U . Himpunan U memungkinkan kita untuk membangun dua model C :

$$\{x_1 \rightarrow 0, x_2 \rightarrow 1, x_3 \rightarrow v, x_4 \rightarrow 1, x_5 \rightarrow 0\}$$

di mana $v \in \{0, 1\}$ karena x_3 dapat mengambil salah satu nilai.

Proposisi 3.4.3

- (a) Jika $BCP(C)$ mengembalikan $\perp$, maka C tidak dapat dipenuhi.
- (b) Jika $BCP(C)$ mengembalikan (U, S) , maka $C \equiv U \cup S$, di mana U adalah himpunan klausa unit dan S adalah himpunan klausa non-unit, dan U dan S tidak memiliki variabel yang sama.

Bukti Dalam $BCP(C)$, S diinisialisasi dengan C . S diperbarui melalui tiga cara: (1) klausa unit (A) dihapus dari S ke U ; (2) klausa $(A \mid \alpha)$ dihapus dari S ; dan (3) klausa $(\bar{A} \mid \alpha)$ diganti dengan (α) . (α) adalah resolven dari (A) dan $(\bar{A} \mid \alpha)$, sehingga $C \models \alpha$.

- (a) Jika $\alpha = ()$ dalam (3), $\perp$ akan dikembalikan dan $C \equiv \perp$ karena $C \models \perp$.
- (b) Awalnya, $C \equiv U \cup S$. Karena A mencakup $(A \mid \alpha)$, dan α mencakup $(\bar{A} \mid \alpha)$, berdasarkan Proposisi 3.3.22, hal ini membenarkan bahwa baik $(A \mid \alpha)$ maupun $(\bar{A} \mid \alpha)$ dapat dihapus dengan aman, karena mempertahankan $C \equiv U \cup S$ untuk setiap pembaruan U dan S . Ketika (A) dipindahkan dari S ke U , baik A maupun $\bar{A}$ dihapus dari S , sehingga variabel A tidak muncul lagi di S . Prosedur berhenti hanya ketika S tidak memiliki klausa unit.

Prosedur BCP dapat berfungsi sebagai prosedur penyederhanaan yang ampuh ketika klausa unit hadir dalam input. BCP memainkan peran penting dalam memutuskan apakah suatu formula dapat dipenuhi, yang merupakan topik bab berikutnya. Nantinya akan kita tunjukkan bahwa $BCP(C)$ dapat diimplementasikan dalam waktu $O(n)$, di mana n adalah jumlah literal dalam C .

Prosedur Pengambilan Keputusan untuk Klausula Horn

Klausula Horn adalah klausula dengan paling banyak satu literal positif. Klausula Horn dinamai menurut Alfred Horn, yang pertama kali menunjukkan signifikansinya pada tahun 1951. Klausula Horn disebut klausula definit jika memiliki tepat satu literal positif. Dengan demikian, klausula Horn dapat dibagi menjadi klausula definit dan klausula negatif. Klausula definit dapat dibagi lagi menjadi fakta (klausula unit positif) dan aturan (klausula definit non-unit). Klausula Horn memiliki aplikasi penting dalam pemrograman logika, spesifikasi formal, dan teori model, karena memiliki prosedur pengambilan keputusan yang sangat efisien, menggunakan resolusi unit atau resolusi input. Artinya, resolusi unit dan input pada umumnya tidak lengkap, tetapi lengkap untuk klausula Horn.

Teorema 3.4.4 BCP adalah prosedur pengambilan keputusan untuk kepuasan klausula Horn.

Bukti Misalkan H adalah himpunan klausula Horn dan $\perp \notin H$. Cukup untuk menunjukkan bahwa prosedur $BCP(H)$ mengembalikan $\perp$ jika dan hanya jika $H \equiv \perp$. Jika $BCP(H)$ mengembalikan $\perp$, berdasarkan Proposisi 3.4.3, $H \equiv \perp$. Jika $BCP(H)$ mengembalikan (U, S) , berdasarkan Proposisi 3.4.3, $H \equiv U \cup S$, U adalah himpunan klausula unit, S adalah himpunan klausula non-unit, dan U dan S tidak memiliki variabel yang sama. Dalam hal ini, kita membuat penugasan σ di mana setiap literal dalam U bernilai benar, dan setiap variabel dalam S bernilai salah. Perhatikan bahwa setiap variabel dalam U hanya muncul sekali; U dan S tidak memiliki variabel yang sama. Dengan demikian, penugasan ini konsisten dan merupakan model dari $U \cup S$, karena setiap klausula unit dalam U bernilai benar di bawah σ ; untuk setiap klausula dalam S , karena merupakan klausula Horn non-unit dan harus memiliki literal negatif, yang benar di bawah σ . Dengan demikian, H dapat dipenuhi ketika $BCP(H)$ mengembalikan (U, S) .

Proposisi 3.4.5 Resolusi unit adalah sah dan lengkap untuk klausula Horn.

Bukti Resolusi unit adalah sah karena resolusi adalah sah. Teorema sebelumnya menunjukkan bahwa BCP adalah prosedur pengambilan keputusan untuk kepuasan klausula Horn. Satu-satunya aturan inferensi yang digunakan dalam BCP adalah resolusi unit. Setiap klausula yang dihasilkan oleh BCP dapat dihasilkan oleh resolusi unit, dan subsumsi hanya mengurangi beberapa klausula yang tidak perlu dan tidak memengaruhi kelengkapannya.

Resolusi Unit Versus Resolusi Input

Berikut ini, kita akan menunjukkan bahwa resolusi input dan resolusi unit adalah ekuivalen, dalam arti bahwa jika ada bukti resolusi unit, maka ada bukti resolusi input dan sebaliknya.

Teorema 3.4.6 Untuk setiap himpunan klausula C , jika ada bukti resolusi input dari C , maka ada bukti resolusi unit dari C .

Bukti Kita akan membuktikan teorema ini dengan induksi pada panjang bukti. Misalkan P adalah bukti resolusi input dengan panjang minimal dari C . Resolven terakhir adalah $\perp$, kedua induknya harus berupa klausa unit, dan salah satunya harus berupa klausa input, misalnya (A) . Untuk mempermudah, anggap semua klausa input yang digunakan dalam P muncul di awal P dan (A) adalah yang pertama dalam P . Jika $|P| = 1$, maka P harus merupakan bukti resolusi unit dan kita selesai. Jika $|P| > 1$, maka hapus (A) , klausa terakhir dalam P (yaitu, $\perp$), dan semua kemunculan $\bar{A}$ dari P , di mana $\bar{A} = \bar{p}$ jika $A = p$ dan $\bar{A} = p$ jika $A = \bar{p}$. Hasilnya adalah bukti resolusi masukan P' dari C' , di mana C' diperoleh dari C dengan menghapus semua kemunculan $\bar{A}$ dalam C . Artinya, $(\bar{A} \mid \alpha) \in C$ jika dan hanya jika $(\alpha) \in C'$. Karena $|P'| < |P|$, berdasarkan hipotesis induksi, terdapat bukti resolusi unit Q' dari C' . Jika $(\alpha) \in C'$ berasal dari $(\bar{A} \mid \alpha) \in C$, maka ubah alasan $(\alpha) \in Q'$ dari “aksioma” menjadi “resolusi dari (A) dan $(\bar{A} \mid \alpha)$ ”. Misalkan Q' yang dimodifikasi adalah Q , maka Q adalah resolusi unit dari C .

Contoh 3.4.7 Kami mengilustrasikan bukti teorema ini dengan Contoh 3.3.5, di mana

$$C = \{c_1 : (p \mid q), c_2 : (\bar{p} \mid r), c_3 : (p \mid \bar{q} \mid r), c_4 : (\bar{r})\}$$

Misalkan P adalah bukti resolusi masukan (b) dari Contoh 3.3.5. Maka $A = r$, dan C' adalah hasil penghapusan r dari C :

$$C' = \{c_1 : (p \mid q), c'_2 : (\bar{p}), c'_3 : (p \mid \bar{q})\}$$

dan dua resolusi masukan di C' adalah $c_5 : (p)$ dari (c_1, c'_3) dan $c_6 : ()$ dari (c'_2, c'_3) . resolusi unit Q' dari C' akan berisi tiga resolusi unit:

$$c_5 : (q) \text{ from } (c_1, c'_2), c_6 : (q) \text{ from } (c'_2, c'_3), c_7 : () \text{ from } (c_5, c_6)$$

Dengan menambahkan dua resolusi unit sebelum Q' , kita peroleh Q :

$$c'_2 : (\bar{p}) \text{ from } (c_2, c_4), c'_3 : (p \mid \bar{q}) \text{ from } (c_3, c_4), c_5 : (q) \text{ from } (c_1, c'_2), \\ c_6 : (q) \text{ from } (c'_2, c'_3), c_7 : () \text{ from } (c_5, c_6)$$

yang merupakan bukti resolusi unit dari C .

Teorema 3.4.8 Untuk setiap himpunan klausa C , jika ada bukti resolusi unit dari C , maka ada bukti resolusi masukan dari C .

Pembuktian teorema ini diserahkan sebagai latihan.

Kedua teorema di atas menetapkan kesetaraan resolusi masukan dan resolusi unit: Bukti resolusi unit ada jika dan hanya jika bukti resolusi masukan ada. Karena resolusi unit lengkap untuk klausa Horn, kita memiliki hasil berikut.

Korolari 3.4.9 Resolusi masukan sah dan lengkap untuk klausa Horn.

Teorema 3.4.10 Untuk setiap himpunan klausa Horn H , misalkan $H = X \cup Y$, di mana X adalah himpunan semua klausa negatif dan $Y = H - X$. Maka resolusi masukan dan himpunan pendukung sah dan lengkap untuk kepuasan H , ketika X digunakan sebagai himpunan pendukung.

Resolusi bukti tidak mungkin dilakukan di antara klausa-klausa dalam X karena resolusi membutuhkan sepasang literal komplementer dari dua klausa induk. Resolusi himpunan pendukung tidak memungkinkan resolusi di antara klausa-klausa dalam Y . Dengan demikian, resolusi hanya dimungkinkan ketika satu klausa berasal dari X dan satu dari Y , dan resolvennya adalah klausa negatif, karena satu-satunya literal positif dari Y diresolusi. Untuk setiap resolusi antara

X dan Y , kita menambahkan resolven ke dalam X , sampai klausa kosong $\perp$ ditemukan atau tidak ada resolven baru yang dapat dihasilkan. Jika $\perp$ dihasilkan, maka $H \equiv \perp$ karena resolusinya sah.

Jika klausa kosong tidak dihasilkan dan X jenuh (yaitu, tidak ada resolven baru yang dapat dihasilkan), kita membuat penugasan σ sebagai berikut: Panggil BCP pada Y . Karena Y dapat dipenuhi, misalkan $(U, S) = BCP(Y)$. Untuk setiap klausa unit p dalam U , misalkan $\sigma(p) = 1$; Untuk variabel q lain yang belum diberi nilai dalam S , misalkan $\sigma(q) = 0$.

Kami mengklaim bahwa σ adalah model dari H . Pertama, $\sigma(U \cup S) = 1$, karena U dan S tidak memiliki variabel yang sama, dan penugasan akan membuat setiap klausa dalam $U \cup S$ menjadi benar.

Misalkan X adalah himpunan klausa negatif dari H ditambah semua resolven antara X dan Y . Jika $\sigma(X) = 0$, pasti ada klausa $C \in X$ dengan panjang minimal, sehingga $\sigma(C) = 0$. Karena C adalah klausa negatif, misalkan $C = (\overline{p_1} \mid \overline{p_2} \mid \cdots \mid \overline{p_k})$. Karena $\sigma(C) = 0$, untuk $1 \leq j \leq k$, $\sigma(p_j) = 1$ dan haruslah berlaku bahwa $p_j \in U$ karena kita hanya memberikan nilai 1 pada variabel-variabel U .

Sekarang perhatikan p_1 dari C . Jika $p_1 \in Y$, resolven dari p_1 dan C adalah $C' = (\overline{p_2} \mid \cdots \mid \overline{p_k})$. Jika $p_1 \notin Y$, maka $Y \cup \{p_1\}$ tidak dapat dipenuhi dan terdapat bukti resolusi negatif P dari $Y \cup \{p_1\}$. Mengganti $(\overline{p_1})$ dengan C dalam P , kita memperoleh $C' = (\overline{p_2} \mid \cdots \mid \overline{p_k})$ dengan resolusi negatif dari $Y \cup \{C\}$. Mudah dilihat bahwa $\sigma(C') = 0$ dan $C' \in X$. Karena $|C'| < |C|$, ini merupakan kontradiksi dengan asumsi bahwa C memiliki panjang minimal. Dengan demikian, $\sigma(X) = 1$, σ adalah model dari $H = X \cup Y$ dan H dapat dipenuhi.

Perhatikan bahwa resolusi masukan dengan klausa negatif sebagai himpunan pendukung pada klausa Horn juga merupakan resolusi negatif. Jika himpunan pendukung awalnya hanya berisi satu klausa, maka resolusi masukan ini juga merupakan resolusi linier. Jika literal positif maksimal di setiap klausa, maka resolusi ini juga merupakan resolusi terurut.

Akibatnya 3.4.11 Jika H adalah himpunan klausa Horn yang berisi satu klausa negatif, maka resolusi negatif, masukan, dan linier lengkap untuk kepuasan H .

Bukti Jika C adalah satu-satunya klausa negatif, misalkan $X = \{C\}$ dan $Y = H - X$. Dengan menggunakan X sebagai himpunan pendukung, kita akan mendapatkan bukti resolusi input. Karena semua klausa negatif antara X dan Y berasal dari C , bukti tersebut dapat disusun sebagai bukti resolusi linier.

Hasil di atas adalah dasar untuk kelengkapan program Prolog murni, yang akan dibahas di Bab 8.

Literal Kepala/Ekor untuk BCP

Agar klausa unit (A) bernilai benar, nilai kebenaran literal A harus 1. Kita dapat menggunakan interpretasi parsial σ untuk mengingat literal mana yang muncul dalam klausa unit.

Definisi 3.4.12 Interpretasi parsial σ adalah himpunan σ literal, di mana tidak ada variabel yang muncul dalam σ lebih dari sekali, dengan pemahaman bahwa untuk literal A apa pun, jika $A \in \sigma$, maka $\sigma(A) = 1$ dan $\sigma(\bar{A}) = 0$, di mana $\bar{A} = p$ jika A adalah $\bar{p}$. Ketika σ mengandung setiap variabel tepat sekali, itu adalah interpretasi lengkap.

Dari definisi di atas, akan lebih mudah untuk menganggap sekumpulan klausa unit sebagai interpretasi parsial. Misalnya, jika $U = \{p, \neg q, r\}$, itu mewakili interpretasi $\sigma = \{p \rightarrow 1, q \rightarrow 0, r \rightarrow 1\}$. Kita dapat mengevaluasi nilai kebenaran suatu klausa atau rumus dalam interpretasi parsial seperti yang kita lakukan dalam interpretasi lengkap.

Definisi 3.4.13 Diberikan interpretasi parsial σ dan klausa c ,

- c terpenuhi jika satu atau lebih literalnya benar dalam σ .
- c bertentangan jika semua literal c salah dalam σ .
- c adalah unit jika semua literalnya kecuali satu salah dan c tidak terpenuhi.
- c tidak terpecahkan jika c bukan salah satu dari tiga kasus di atas.

Ketika c menjadi unit, kita harus menetapkan 1 ke literal yang belum ditetapkan, misalnya A , dalam c , sehingga c menjadi terpenuhi. Artinya, c adalah alasan literal A menjadi benar, dan kita mencatat ini sebagai $reason(A) = c$. Kita juga mengatakan literal A diimplikasikan oleh c .

Contoh 3.4.14 Diberikan klausa $\{c_1 : (\bar{x}_1), c_2 : (x_3), c_3 : (x_1 | x_2 | \bar{x}_3)\}$, $reason(\bar{x}_1) = c_1$, $reason(x_3) = c_2$, dan $reason(x_2) = c_3$. x_2 diimplikasikan menjadi benar oleh c_3 karena c_3 setara dengan $(\bar{x}_1 \wedge x_3) \rightarrow x_2$.

Dalam *BCP*, kita mulai dengan interpretasi parsial kosong σ . Jika kita telah menemukan klausa unit (A), kita memperluas σ dengan menambahkan A ke dalam σ sehingga klausa unit menjadi terpenuhi. Untuk menghemat waktu, kita tidak menghapus klausa yang mengandung A dari himpunan klausa. Kita tidak menghapus $\bar{A}$ dari klausa mana pun untuk membuat klausa baru. Sebaliknya, kita hanya mengingat bahwa $\bar{A}$ sekarang salah di σ .

Kita perlu mencari tahu klausa mana yang menjadi klausa unit ketika beberapa literal menjadi salah. Klausa c dengan k literal menjadi unit ketika $k - 1$ literal dari c salah. Jika kita tahu bahwa dua literal dari c tidak salah, kita yakin bahwa c tidak terpecahkan. Mari kita sebut kedua literal ini sebagai literal kepala dan ekor dari c , yang merupakan dua literal berbeda dari c ; kita memiliki sifat berikut.

Proposisi 3.4.15 Klausa c yang panjangnya lebih dari satu bukanlah unit maupun bertentangan dalam interpretasi parsial σ jika salah satu literal kepala/ekor dari c bernilai benar atau tidak satupun dari literal tersebut bernilai salah dalam σ .

Bukti Misalkan c berisi k literal dan $k > 1$. Jika c terpenuhi, maka c berisi literal yang bernilai benar dan kita dapat menggunakannya sebagai salah satu literal kepala/ekor. Jika c adalah unit atau bertentangan, maka ada $k - 1$ atau semua literal dari c bernilai salah, dan tidak ada literal dalam c yang bernilai benar. Dalam hal ini, tidak ada dua literal yang tidak bernilai salah untuk literal kepala/ekor. Jika c tidak terpecahkan, maka harus ada dua literal yang belum ditetapkan untuk literal kepala/ekor.

Awalnya, misalkan literal pertama dan terakhir adalah literal kepala/ekor. Ketika literal kepala menjadi salah, kita memindai ke kanan untuk menemukan literal berikutnya dalam c yang tidak bernilai salah dan menjadikannya literal kepala yang baru; Demikian pula, ketika literal ekor menjadi salah, kita memindai ke kiri untuk menemukan literal ekor berikutnya. Jika kita tidak dapat menemukan literal kepala atau ekor baru, kita pasti telah menemukan klausa unit atau klausa yang bertentangan.

Untuk mengimplementasikan ide di atas, kita menggunakan struktur data berikut untuk klausa: Untuk setiap klausa c yang berisi lebih dari satu literal:

- $lits(c)$: array literal yang menyimpan literal dari c
- $head(c)$: indeks literal pertama yang bukan salah dari c dalam $lits(c)$
- $tail(c)$: indeks literal terakhir yang bukan salah dari c dalam $lits(c)$

Misalkan $|c|$ Misalkan kita menyatakan jumlah literal dalam c , maka indeks yang valid untuk $lits(c)$ adalah $\{0, 1, \dots, |c| - 1\}$, dan $head(c)$ diinisialisasi dengan 0 dan $tail(c)$ dengan $|c| - 1$. Berikut adalah struktur data yang terkait dengan setiap literal A :

- $val(A)$: nilai kebenaran A di bawah interpretasi parsial, $val(A) \in \{1, 0, \times\}$, $\times$ ("tidak ditetapkan") adalah nilai awal.
- $cls(A)$: daftar klausa c sedemikian sehingga A ada di c dan ditunjuk oleh $head(c)$ atau $tail(c)$ di $lits(c)$. Kita menggunakan $insert(c, cls(A))$ dan $remove(c, cls(A))$ untuk memasukkan dan menghapus klausa c ke/dari $cls(A)$, masing-masing.

Dengan asumsi variabel global digunakan untuk klausa dan literal, prosedur BCP dapat diimplementasikan sebagai berikut:

```

proc BCP(C)
    // initialization
1   U :=  $\emptyset$  // U: stack of unit clauses
2   for c ∈ C do
3       if c = () return ⊥
4       if (|c| > 1)
5           lits(c) := makeArray(c) // membuat array literal
6           head(c) := 0; insert(c, cls(lits(c)[0]))
7           tail(c) := |c| − 1; insert(c, cls(lits(c)[|c| − 1]))
8       else push(c, U)
9   for each literal A, val(A) := × // × = “belum ditetapkan”
    // pekerjaan utama
10  res := BCPht(U)
    // penyelesaian
11  if (res ≠ "SAT") return ⊥
12  U := {A | val(A) = 1}
12  S := {clean(c) | val(lits(c)[head(c)]) = ×, val(lits(c)[tail(c)]) = ×}
14  return (U, S)
    
```

Perhatikan bahwa prosedur *clean*(*c*) pada baris 13 akan menghapus literal yang salah (di bawah *val*) dari *c* dan mengembalikan T jika salah satu literal dalam *c* benar. Prosedur *BCPht* akan melakukan pekerjaan utama *BCP*.

Algoritma 3.4.16 *BCPht*(*U*) mengasumsikan klausa input $C = U \cup S$, di mana *U* adalah tumpukan klausa unit dan *S* adalah himpunan klausa non-unit yang disimpan menggunakan struktur data head/tail, dan dapat diakses melalui *cls*(*A*). *BCPht* mengembalikan klausa konflik jika klausa kosong ditemukan selama propagasi unit; ia mengembalikan “SAT” jika tidak ada klausa kosong yang ditemukan.

```

proc BCPht(U)
1   while U ≠  $\emptyset$  do
2       A := pop(U)
3       if val(A) = 0 return reason( $\bar{A}$ ) // an empty clause is found
4       else if (val(A) = ×) // × is “unassigned”
5           val(A) := 1; val( $\bar{A}$ ) := 0
6       for c ∈ cls( $\bar{A}$ ) do //  $\bar{A}$  is either head or tail literal of c
7           if ( $\bar{A}$  = lits(c)[head(c)])
8               e1 := head(c); e2 := tail(c); step := 1 // scan from left to right
9           else
10              e1 := tail(c); e2 := head(c); step := −1 // scan from right to left
11          while true do
    
```

```

12       $x := e_1 + step$  //  $x$  takes values from  $e_1 + step$  to  $e_2$ 
13      if  $x = e_2$  break // exit the inner while loop
14       $B := lits(c)[x]$ 
15      if ( $val(B) \neq 0$ ) // new head/tail literal is found
16           $remove(c, cls(\bar{A}))$ ;  $insert(c, cls(B))$ 
17          if ( $step = 1$ )  $head(c) := x$  else  $tail(c) := x$ 
18          break // exit the inner while loop
19      if ( $x = e_2$ ) // no new head/tail available
20           $A := lits(c)[e_2]$  // check if  $c$  is unit or conflicting
21          if  $val(A) = 0$  return  $c$  //  $c$  is conflicting
22          else if ( $val(A) = \times$ ) //  $c$  is unit
23               $push(A, U)$ ;  $reason(A) := c$ ;  $val(A) := 1$ ;  $val(\bar{A}) := 0$ 
24      return "SAT" // no empty clauses are found

```

Contoh 3.4.17 Untuk klausa-klausa dalam Contoh 3.4.2, kita memiliki struktur data berikut setelah inisialisasi (indeks kepala dan ekor diberikan setelah daftar literal di setiap klausa):

$$C = \{c_1 : \langle [x_2, x_5], 0, 1 \rangle, c_2 : \langle [\bar{x}_1, \bar{x}_4], 0, 1 \rangle, c_3 : \langle [\bar{x}_2, x_4], 0, 1 \rangle, c_4 : \langle [x_1, x_2, \bar{x}_3], 0, 2 \rangle\}$$

$U = \{\bar{x}_5\}$; $cls(A)$ contains these clauses :

$cls(x_1) = \{c_4\},$	$cls(\bar{x}_1) = \{c_2\}$
$cls(x_2) = \{c_1\},$	$cls(\bar{x}_2) = \{c_3\}$
$cls(x_3) = \emptyset,$	$cls(\bar{x}_3) = \{c_4\}$
$cls(x_4) = \{c_3\},$	$cls(\bar{x}_4) = \{c_2\}$
$cls(x_5) = \{c_1\},$	$cls(\bar{x}_5) = \emptyset$

dan $val(A) = \times$ (tidak ditetapkan) untuk semua literal A . Perhatikan bahwa setiap klausa muncul dua kali dalam koleksi $cls(A)$.

Di dalam $BCPht(U)$, ketika $\bar{x}_5$ dikeluarkan dari U , kita menetapkan $val(\bar{x}_5) = 1$, $val(x_5) = 0$ (baris 5) dan memeriksa $c_1 \in cls(x_5)$ (baris 6). c_1 menghasilkan klausa unit x_2 dan ditambahkan ke U (baris 23) dengan penetapan $val(x_2) = 1$, $val(\bar{x}_2) = 0$. Ketika x_2 dikeluarkan dari U , kita memeriksa $c_3 \in cls(\bar{x}_2)$. c_3 menghasilkan klausa unit x_4 , kita memasukkan x_4 ke U , dan menetapkan $val(x_4) = 1$, $val(\bar{x}_4) = 0$. Ketika x_4 dikeluarkan dari U , kita memeriksa $c_2 \in cls(\bar{x}_4)$. c_2 menghasilkan klausa unit $\bar{x}_1$; Kita memasukkan $\bar{x}_1$ ke dalam U dan menetapkan $val(\bar{x}_1) = 1$, $val(x_1) = 0$. Ketika $\bar{x}_1$ dikeluarkan dari U , kita memeriksa $c_4 \in cls(x_1)$. c_4 bernilai benar karena $val(x_2) = 1$. Sekarang indeks kepala c_4 diperbarui menjadi 1, dan kita menambahkan $c_4 = ([x_1, x_2, \bar{x}_2], 1, 2)$ ke dalam $cls(x_2)$. Karena klausa kosong tidak ditemukan, $BCPht$ mengembalikan "SAT." Akhirnya, BCP mengembalikan $(U, S) = (\{\bar{x}_5, x_2, x_4, \bar{x}_1\}, \emptyset)$.

Perhatikan bahwa dua penugasan " $val(a) := 1$; $val(\bar{a}) := 0$ " pada baris 23 $BCPht$ dapat dihapus karena pekerjaan yang sama akan dilakukan pada baris 5. Namun, keberadaannya di sini seringkali meningkatkan kinerja $BCPht$. Pada baris 23, kita juga mencatat alasan penugasan literal; Informasi ini dikembalikan pada baris 3 dan berguna jika

kita ingin mengetahui mengapa suatu klausa menjadi bertentangan. *BCPht* dapat sedikit ditingkatkan dengan memeriksa ($val(lits(c)[e_2]) = 1$) setelah baris 10: Jika benar, maka lewati klausa c karena c terpenuhi.

Dalam implementasi *BCP* ini, setelah semua klausa C dibaca, setiap kemunculan literal dari klausa non-unit akan dikunjungi paling banyak sekali di *BCPht*.

Proposisi 3.4.18 *BCP*(C) berjalan dalam waktu $O(n)$, di mana n adalah jumlah total kemunculan literal di semua klausa C .

Bukti Jelas, inisialisasi *BCP* membutuhkan waktu $O(n)$. Untuk *BCPht*, kita mengakses klausa non-unit c ketika literal kepala (atau ekor)nya menjadi salah (baris 6). Kita lewati literal kepala (ekor) (baris 11) dan periksa literal berikutnya b (baris 13); jika salah, kita lanjutkan (baris 14); Jika tidak, kita menghapus c dari $cls(A)$ ke $cls(B)$ (baris 16) dan memperbarui indeks kepala (ekor) (baris 17). Jika kita tidak dapat menemukan literal kepala (ekor) baru (baris 19), kita telah menemukan klausa unit baru (baris 20). Tidak ada literal c yang dikunjungi lebih dari sekali oleh *BCPht*. Secara khusus, kita tidak pernah mengakses klausa dari $cls(A)$ ketika literal A bernilai benar.

Menggabungkan proposisi di atas dengan Teorema 3.4.4, kita memiliki hasil berikut:

Teorema 3.4.19 *BCP* adalah prosedur pengambilan keputusan waktu linear untuk kepuasan klausa Horn.

3.5 LATIHAN SOAL

- Buktikan secara formal bahwa pernyataan-pernyataan berikut ini setara: Untuk setiap dua rumus A dan B , (a) $A \models B$; (b) $A \rightarrow B$ valid; (c) $A \wedge \neg B$ tidak dapat dipenuhi.
- Gunakan metode tabel semantik untuk menentukan apakah rumus-rumus berikut ini valid atau tidak:
 - $(A \leftrightarrow (A \wedge B)) \rightarrow (A \rightarrow B)$
 - $(B \leftrightarrow (A \vee B)) \rightarrow (A \rightarrow B)$
 - $(A \oplus B) \rightarrow ((A \vee B) \wedge (\neg A \vee \neg B))$
 - $(A \leftrightarrow B) \rightarrow ((A \vee \neg B) \wedge (\neg A \vee B))$
- Gunakan metode tabel semantik untuk menentukan apakah relasi implikasi berikut ini benar atau tidak:
 - $(A \wedge B) \rightarrow C \models (A \rightarrow C) \vee (B \rightarrow C)$
 - $(A \wedge B) \rightarrow C \models A \rightarrow C$
 - $A \rightarrow (B \rightarrow C) \models (A \rightarrow B) \rightarrow (A \rightarrow C)$
 - $(A \vee B) \wedge (\neg A \vee C) \models B \vee C$
- Buktikan dengan metode tabel semantik bahwa pernyataan berikut, “batas utang tidak dinaikkan atau pengeluaran tidak meningkat” ($\neg d \vee \neg e$), secara logis mengikuti dari pernyataan dalam Soal 31 Bab 2.

5. Buktikan bahwa jika semua aksioma adalah tautologi dan semua aturan inferensi adalah sah, maka semua rumus turunan dari aksioma dan aturan inferensi adalah tautologi.
6. Tentukan aturan pengenalan dan penghapusan untuk operator $\oplus$ dan $\leftrightarrow$ dalam sistem deduktif alami.
7. Perkirakan batas atas yang ketat dari ukuran grafik inferensi untuk resolusi, jika klausa masukan berisi n variabel.
8. Diberikan klausa non-unit c , kita dapat membagi c menjadi dua klausa dengan memperkenalkan variabel baru: Misalkan c adalah $(\alpha \mid \beta)$, di mana α dan β adalah daftar literal, dan $S = \{(\alpha \mid \gamma), (\bar{\gamma} \mid \beta)\}$, di mana γ adalah variabel baru. Tunjukkan bahwa $S \models c$ dan $S \approx c$. Dengan menggunakan teknik ini, tunjukkan bahwa klausa dengan $n \geq 4$ literal dapat dibagi menjadi himpunan equisatisfiable $n - 2$ klausa yang masing-masing memiliki 3 literal (3CNF).
9. Gunakan pengkodean Tseitin untuk mengubah rumus-rumus berikut menjadi 3CNF (CNF di mana setiap klausa memiliki paling banyak tiga literal):
 - (a) $(A \leftrightarrow (A \wedge B)) \rightarrow (A \rightarrow B)$
 - (b) $(A \oplus B) \rightarrow (A \vee B)$
 - (c) $((A \wedge B) \rightarrow C) \rightarrow (A \rightarrow C)$
 - (d) $((A \vee B) \wedge (\neg A \vee C)) \rightarrow (B \vee C)$
10. Gunakan metode resolusi untuk menentukan apakah rumus-rumus berikut valid atau tidak. Jika rumus tersebut valid, berikan bukti resolusi; jika tidak valid, berikan interpretasi yang membuktikan bahwa rumus tersebut salah.
 - (a) $(A \leftrightarrow (A \wedge B)) \rightarrow (A \rightarrow B)$
 - (b) $(B \leftrightarrow (A \vee B)) \rightarrow (A \rightarrow B)$
 - (c) $(A \oplus B) \rightarrow ((A \vee B) \wedge (\neg A \vee \neg B))$
 - (d) $((\tilde{A} \downarrow B) \uparrow \tilde{C}) \rightarrow (A \uparrow (B \downarrow C))$
11. Gunakan metode resolusi untuk menentukan apakah relasi implikasi berikut ini benar atau salah. Jika relasi tersebut berlaku, berikan bukti resolusi; jika tidak, berikan interpretasi sedemikian rupa sehingga premisnya benar, tetapi konsekuensi dari implikasi tersebut salah.
 - (a) $(A \rightarrow C) \wedge (B \rightarrow D) \models (A \vee B) \rightarrow (C \vee D)$
 - (b) $(A \wedge B) \rightarrow C \models (A \rightarrow C)$
 - (c) $A \rightarrow (B \rightarrow C) \models (A \rightarrow B) \rightarrow (A \rightarrow C)$
 - (d) $(A \vee B) \wedge (\neg A \vee C) \models B \vee C$
12. Terapkan strategi penghapusan klausa pada setiap himpunan klausa berikut dan peroleh himpunan yang lebih sederhana dan setara.

$$S_1 = \{(p \mid q \mid r), (\bar{p} \mid q \mid \bar{r} \mid s), (p \mid \bar{q} \mid s), (\bar{q} \mid r), (q \mid \bar{r})\}$$

$$S_2 = \{(p \mid q \mid r \mid s), (\bar{p} \mid q \mid \bar{r} \mid \bar{s}), (p \mid q \mid s), (q \mid \bar{r}), (q \mid s \mid \bar{r})\}$$

$$S_3 = \{(p \mid q \mid s), (\bar{p} \mid q \mid \bar{r} \mid \bar{s}), (p \mid r \mid s), (\bar{p} \mid r), (\bar{p} \mid q \mid r)\}$$
13. Diberikan himpunan klausa S , di mana $S = \{1. (t \mid \bar{e} \mid d), 2. (\bar{t} \mid c), 3. (e \mid \bar{d} \mid i), 4. (\bar{g} \mid \bar{d} \mid i), 5. (t \mid c \mid \bar{d} \mid g), 6. (\bar{c}), 7. (\bar{t} \mid \bar{g}), 8. (c \mid d), 9. (c \mid e)\}$, temukan bukti

- resolusi berikut untuk S : (a) resolusi unit, (b) resolusi input, (c) resolusi positif, dan (d) resolusi terurut menggunakan urutan $t > i > g > e > d > c$.
14. Terapkan berbagai strategi resolusi untuk membuktikan bahwa himpunan klausa S berikut ini menyiratkan: resolusi unit, resolusi input, resolusi negatif, resolusi positif, dan resolusi terurut menggunakan urutan $p > q > r > s$.
 $S = \{1. (p \mid q \mid r), 2. (p \mid q \mid s), 3. (p \mid q \mid r), 4. (p \mid q \mid r), 5. (p \mid q \mid r), 6. (q \mid r), 7. (r \mid s)\}$
 Harap tunjukkan bukti dari strategi resolusi ini.
 15. Tentukan dengan resolusi terurut apakah himpunan klausa S dapat dipenuhi atau tidak, di mana $S = \{1. (a \mid c \mid \bar{d} \mid f), 2. (a \mid d \mid \bar{e}), 3. (\bar{a} \mid c), 4. (b \mid \bar{d} \mid e), 5. (b \mid d \mid \bar{f}), 6. (\bar{b} \mid \bar{f}), 7. (c \mid \bar{e}), 8. (c \mid e), 9. (\bar{c} \mid \bar{d})\}$, dan urutannya adalah $a > b > c > d > e > f$. Jika himpunan klausa dapat dipenuhi, harap tampilkan semua resolven dari resolusi terurut dan buat model dari klausa-klausa ini. Jika S tidak dapat dipenuhi, harap tampilkan pohon resolusi terurut dari bukti tersebut.
 16. Buktikan dengan resolusi bahwa $S \models \neg f$, di mana S adalah himpunan klausa yang diberikan pada soal sebelumnya. Harap tampilkan pohon resolusi dari bukti ini.
 17. Jika literal kepala/ekor digunakan untuk mengimplementasikan BCP, literal mana dari himpunan klausa S pada masalah sebelumnya yang akan diberi nilai kebenaran selama resolusi unit sebelum klausa kosong ditemukan? Dalam urutan apa?
 18. Suatu himpunan C klausa dikatakan sebagai $2CNF$ jika setiap klausa berisi paling banyak dua literal. Masalah $2SAT$ adalah untuk memutuskan apakah suatu formula $2CNF$ dapat dipenuhi atau tidak. Tunjukkan bahwa *orderedResolution* membutuhkan waktu $O(n^3)$ dan ruang $O(n^2)$ untuk menyelesaikan $2SAT$, di mana n adalah jumlah variabel proposisional dalam $2CNF$.
 19. Buktikan Teorema 3.4.8: Untuk setiap himpunan C klausa, jika ada bukti resolusi unit dari C , maka ada bukti resolusi input dari C .

BAB 4

KETERPUASAN PROPOSISIONAL

Keterpuasan proposisional (SAT) adalah masalah memutuskan apakah suatu formula proposisional dapat dipenuhi, yaitu, apakah formula tersebut memiliki model. Program perangkat lunak untuk menyelesaikan SAT disebut pemecah SAT. SAT telah lama menikmati status khusus dalam ilmu komputer. Dari sisi teoritis, ini adalah masalah NP-lengkap pertama yang pernah ditemukan. Masalah NP adalah masalah yang solusinya dapat diverifikasi dalam waktu polinomial (waktu), dan masalah NP-lengkap adalah masalah NP yang paling sulit.

Dalam kasus terburuk, waktu komputasi dari algoritma yang diketahui untuk masalah NP-lengkap meningkat secara eksponensial dengan ukuran input. Dari sisi praktis, karena setiap masalah NP dapat ditransformasikan menjadi SAT, pemecah SAT yang efisien dapat digunakan untuk menyelesaikan banyak masalah praktis. SAT menemukan beberapa aplikasi penting dalam desain dan verifikasi sistem perangkat keras dan perangkat lunak, dan di banyak bidang kecerdasan buatan dan riset operasi. Dengan demikian, ada motivasi yang kuat untuk mengembangkan pemecah SAT yang bermanfaat secara praktis. Namun, NP-completeness menjadi penyebab pesimisme, karena kecil kemungkinan kita dapat menskalakan solusi tersebut ke contoh praktis yang besar.

Secara teori, prosedur pengambilan keputusan yang disajikan pada bab sebelumnya, seperti tabel semantik atau resolusi, dapat digunakan sebagai pemecah SAT. Namun, prosedur tersebut sebagian besar berfungsi sebagai latihan akademis dengan sedikit harapan untuk penggunaan praktis. Resolusi sebagai pembuktian sanggahan mencari klausa kosong. Tabel semantik meniru transformasi formula menjadi DNF. Tidak satu pun dari prosedur pengambilan keputusan ini bertujuan untuk mencari model.

Untungnya, dalam dua dekade terakhir, beberapa perkembangan penelitian telah memungkinkan kita untuk menangani contoh SAT dengan puluhan ribu variabel dan jutaan klausa. Para peneliti SAT memperkenalkan teknik seperti pembelajaran klausa berbasis konflik, heuristik percabangan baru, dan propagasi unit yang efisien. Teknik-teknik ini membentuk dasar dari banyak pemecah SAT modern. Dengan menggunakan ide-ide ini, pemecah SAT kontemporer seringkali dapat menangani contoh SAT yang sangat besar.

Kompetisi tahunan pemecah SAT mempromosikan kemajuan berkelanjutan dari pemecah SAT. Pemecah SAT adalah mesin dari alat pengecekan model modern. Teknik verifikasi otomatis kontemporer seperti pengecekan model terbatas, abstraksi berbasis bukti, dan pengecekan model berbasis interpolasi, semuanya didasarkan pada pemecah SAT dan perluasannya. Alat-alat ini digunakan untuk memeriksa kebenaran desain perangkat lunak dan perangkat keras.

4.1 ALGORITMA DPLL

Pada tahun 1960, Martin Davis dan Hilary Putnam mengusulkan aturan inferensi resolusi untuk logika proposisional. Untuk masalah realistis, jumlah klausa yang dihasilkan oleh

resolusi tumbuh dengan cepat. Pada tahun 1962, untuk menghindari ledakan ini, Davis, George Logemann, dan Donald Loveland menyarankan untuk mengganti aturan resolusi dengan pemisahan kasus: Pilih variabel p dan pertimbangkan dua kasus $p \rightarrow 1$ dan $p \rightarrow 0$ secara terpisah. Algoritma yang dimodifikasi ini umumnya disebut sebagai algoritma DPLL, di mana *DPLL* adalah singkatan dari Davis, Putnam, Logemann, dan Loveland.

DPLL dapat dianggap sebagai prosedur pencarian: Ruang pencarian adalah semua interpretasi, parsial atau penuh. Interpretasi parsial dapat direpresentasikan oleh himpunan literal, di mana setiap literal dalam himpunan tersebut diberi nilai benar dan setiap literal di luar himpunan tersebut diasumsikan tidak diberi nilai. *DPLL* dimulai dengan interpretasi kosong dan mencoba memperluas interpretasi dengan menambahkan p atau $\neg p$ melalui pemisahan kasus. *DPLL* berhenti ketika setiap variabel diberi nilai tanpa kontradiksi dan interpretasi parsial yang dipertahankannya menjadi sebuah model.

Versi Rekursif DPLL

Berikut ini, kami memberikan versi rekursif *DPLL*. Panggilan pertama ke *DPLL* dilakukan dengan himpunan C klausa masukan dan interpretasi kosong $\sigma = \{\}$. Ini adalah salah satu prosedur backtracking tertua. *DPLL* banyak menggunakan prosedur bernama *BCP* (Boolean constraint propagation) untuk menangani klausa unit. Dari bab sebelumnya, kita tahu bahwa *BCP* mengimplementasikan resolusi unit dan penghapusan subsumsi. Diberikan himpunan klausa C , *BCP*(C) mengembalikan $\perp$ jika klausa kosong ditemukan melalui resolusi unit; jika tidak, *BCP*(C) mengembalikan (U, S) , sedemikian sehingga $C \equiv (U \cup S)$, U adalah himpunan klausa unit, S tidak mengandung klausa unit apa pun, dan U dan S tidak memiliki variabel yang sama. Perhatikan bahwa *BCP*(C) membutuhkan waktu $O(n)$ untuk dijalankan, di mana n adalah jumlah kemunculan literal dari C .

Algoritma 4.1.1 *DPLL*(C, σ) menerima input berupa himpunan klausa C dan interpretasi parsial σ , di mana C dan σ tidak memiliki variabel yang sama. Algoritma ini mengembalikan model C jika C dapat dipenuhi; mengembalikan $\perp$ jika C tidak dapat dipenuhi. Prosedur *pickLiteral*(S) akan memilih literal yang muncul dalam S .

```

proc DPLL( $C, \sigma$ )
1   $res := BCP(C)$ 
2  if ( $res = \perp$ ) return  $\perp$  //  $C$  has no models
   // It must be the case  $res = (U, S)$ 
3   $\sigma := \sigma \cup U$ 
4  if ( $S = \emptyset$ ) return  $\sigma$  //  $\sigma$  is a model of  $C$ 
5   $A := pickLiteral(S)$ 
6   $res := DPLL(S \cup \{(A)\}, \sigma)$ 
7  if ( $res \neq \perp$ ) return  $res$  // a model is found.
8  return DPLL( $S \cup \{(\bar{A})\}, \sigma$ ) // either  $\perp$  or a model is returned.

```

Dalam prosedur di atas, kita berasumsi lagi bahwa $\bar{A} = p$ jika A adalah literal $\bar{p}$. Jika $BCP(C)$ tidak mengembalikan $\perp$, maka ia menyederhanakan C menjadi $U \cup S$, di mana U adalah himpunan literal, S adalah himpunan klausa non-unit, dan U dan S tidak berbagi variabel apa pun. Berdasarkan Proposisi 3.4.3, $C \equiv U \cup S$. Kita mengidentifikasi dua jenis literal dalam U .

Definisi 4.1.2 Dalam $DPLL$, literal $A \in U$ dikatakan sebagai literal keputusan jika A adalah literal yang dikembalikan oleh $pickLiteral(S)$ pada baris 5 $DPLL$; jika tidak, A dikatakan sebagai literal tersirat.

Karena U adalah himpunan literal tanpa literal komplementer, U dianggap sebagai interpretasi parsial σ sedemikian sehingga $\sigma(U) = 1$. Karena U dapat dipenuhi dan $C \equiv U \cup S$, kita menyimpulkan bahwa $C \approx S$. Lebih lanjut, pemeriksaan kepuasan S diubah secara rekursif menjadi pemeriksaan $S \wedge A$ dan $S \wedge \neg A$ secara terpisah, karena $S \equiv (S \wedge A) \vee (S \wedge \neg A)$ untuk literal A yang muncul di S . Transformasi ini, yang diterapkan secara rekursif, menghasilkan prosedur keputusan lengkap untuk kepuasan. Eksekusi $DPLL$ dapat digambarkan dengan pohon yang akarnya adalah panggilan pertama ke $DPLL$ dan setiap simpul non-akar adalah panggilan rekursif ke $DPLL$. Selanjutnya, pohon rekursi ini akan disebut pohon keputusan, karena literal keputusan terlibat dalam setiap panggilan rekursif.

Contoh 4.1.3 Misalkan $C = \{c_1 : (x_2 | x_3 | x_5), c_2 : (\bar{x}_1 | \bar{x}_4), c_3 : (\bar{x}_2 | x_4), c_4 : (\bar{x}_1 | x_2 | \bar{x}_3), c_5 : (\bar{x}_5)\}$. Jika $pickLiteral$ memilih variabel dalam urutan $x_1, x_2, \dots, x_5$, maka:

$DPLL(C, \{\})$ calls

$BCP(C)$, which returns $(\{\bar{x}_5\}, C_1 = \{c'_1 : (x_2 | x_3), c_2, c_3, c_4\})$;

x_1 is picked as the decision literal of level 1;

$DPLL(C_1 \cup \{(x_1)\}, \{\bar{x}_5\})$, which calls

$BCP(C_1 \cup \{(x_1)\})$, which returns $\perp$;

and returns $\perp$

$DPLL(C_1 \cup \{(\bar{x}_1)\}, \{\bar{x}_5\})$, which calls

$BCP(C_1 \cup \{(\bar{x}_1)\})$, which returns $(\{x_1\}, C_2 = \{c'_1, c_3\})$;

x_2 is picked as the decision literal of level 2;

$DPLL(C_2 \cup \{(x_2)\}, \{\bar{x}_1, \bar{x}_5\})$, which calls

$BCP(C_2 \cup \{(x_2)\})$, which returns $(\emptyset, \{\bar{x}_1, x_2, x_4, \bar{x}_5\})$

and returns $\{\bar{x}_1, x_2, x_4, \bar{x}_5\}$

and returns $\{\bar{x}_1, x_2, x_4, \bar{x}_5\}$

and returns $\{\bar{x}_1, x_2, x_4, \bar{x}_5\}$

Dengan demikian, model C dapat dibangun dari $\{\bar{x}_1, x_2, x_4, \bar{x}_5\}$ dengan menambahkan $x_3 \rightarrow v$ untuk nilai apa pun $v \in \{0, 1\}$.

Dalam contoh di atas, tinggi pohon keputusan adalah dua, x_1 dan x_2 adalah literal keputusan, dan x_4 dan $\bar{x}_5$ adalah literal tersirat. Kita telah memanggil BCP empat kali: satu

panggilan di akar (level 0); dua panggilan di node level satu; dan satu panggilan pada node level dua. Jika kita ingin mencari semua model, kita dapat memilih cabang dengan $x_2 \rightarrow 0$, seperti yang ditunjukkan pada Gambar 4.1.

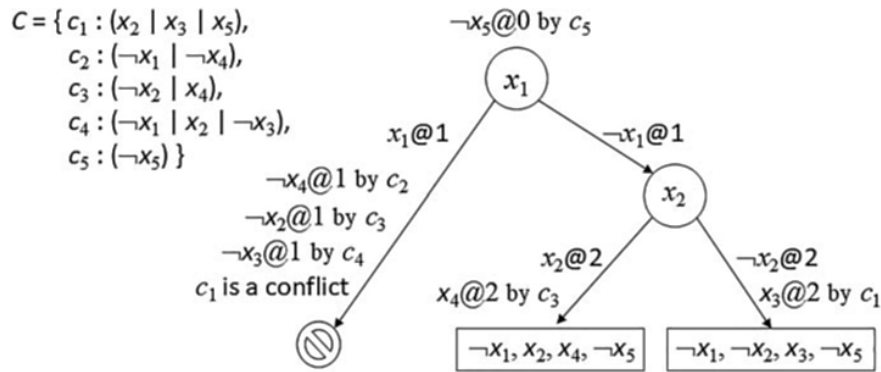
Definisi 4.1.4 Level dari variabel yang ditetapkan adalah kedalaman node dari pohon keputusan $DPLL(C, \{\})$ di mana variabel tersebut diberi nilai kebenaran. Kita menulis $x@l$ jika $x \rightarrow 1$ pada level l , dan $\bar{x}@l$ jika $x \rightarrow 0$ pada level l .

Dalam contoh di atas, untuk model yang dikembalikan oleh $DPLL$, kita dapat menuliskannya sebagai

$$\{\bar{x}_1@1, x_2@2, x_4@2, \bar{x}_5@0\}$$

Teorema 4.1.5 $DPLL(C, \{\})$ adalah prosedur pengambilan keputusan untuk kepuasan C .

Bukti Dengan asumsi $\perp \notin C$. Untuk membuktikan teorema ini, kita hanya perlu menunjukkan bahwa $DPLL$ adalah sah dan berakhir. $DPLL(C, \sigma)$ akan berakhir karena untuk setiap panggilan rekursif, jumlah variabel dalam C berkurang setidaknya satu. Dalam $DPLL(C, \sigma)$, karena σ dan C tidak memiliki variabel yang sama, $(C \cup \sigma) \approx C$. Jadi, dalam bukti kesahihan $DPLL$, kita mengabaikan keberadaan σ .



Gambar 4.1 Pohon keputusan $DPLL$ untuk Contoh 4.1.3. Hasil BCP ditunjukkan di sepanjang tautan pohon.

Kita akan membuktikan dengan induksi pada struktur pohon keputusan $DPLL$. Sebagai kasus dasar, jika $DPLL(C, \sigma)$ mengembalikan $\perp$, itu berarti bahwa $\perp$ dihasilkan oleh resolusi unit di dalam $BCP(C)$. Berdasarkan Proposisi 3.4.3, $C \equiv \perp$. Jika $BCP(C, \sigma)$ mengembalikan (U, S) , berdasarkan Proposisi 3.4.3, $C \equiv U \cup S$. Sebagai kasus dasar lainnya, jika $S = \emptyset$, maka $C \equiv U$ dan U adalah model untuk himpunan klausa unit U jika U dianggap sebagai interpretasi, sehingga C dapat dipenuhi. Jika $S \neq \emptyset$, maka karena

$$S \equiv (S \wedge A) \vee (S \wedge \neg A)$$

Untuk setiap literal A dari S , jika $DPLL(S \cup \{p\}, \sigma)$ atau $DPLL(S \cup \{\bar{p}\}, \sigma)$ mengembalikan sebuah model, berdasarkan hipotesis induksi, itu berarti $(S \wedge p) \vee (S \wedge \neg p)$ dapat dipenuhi, begitu pula S . Jika $DPLL(S \cup \{p\}, \sigma)$ dan $DPLL(S \cup \{\bar{p}\}, \sigma)$ keduanya mengembalikan $\perp$, berdasarkan hipotesis induksi, itu berarti $(S \wedge p) \vee (S \wedge \neg p)$ tidak dapat dipenuhi, begitu pula S .

Pemecah All-SAT dan Incremental SAT

Diberikan himpunan klausa yang dapat dipenuhi, $DPLL$ mengembalikan satu penugasan yang memuaskan. Namun, beberapa aplikasi mengharuskan kita untuk menghitung semua penugasan yang dapat dipenuhi dari suatu formula, dan masalah ini disebut *All – SAT*. Sangat mudah untuk memodifikasi $DPLL$ sehingga semua model dapat ditemukan. Artinya, kita mengganti baris 4 dari $DPLL$ sebagai berikut:

4: **if** ($S = \emptyset$) *save*(σ); **return** $\perp$; // *simpan model dan lanjutkan*

Ketika $DPLL$ yang dimodifikasi selesai, semua model disimpan. Perhatikan bahwa model-model ini adalah model parsial. Satu model parsial mewakili 2^k model lengkap, di mana k adalah jumlah variabel yang belum ditetapkan dalam model parsial.

Selama eksekusi $DPLL$, daftar literal keputusan (berlabel 1) dan komplemennya (berlabel 2) menentukan posisi saat ini dalam pohon keputusan $DPLL$. Daftar literal ini, yang disebut jalur panduan, dapat sangat berguna untuk merancang pemecah *SAT* inkremental: Pemecah mengambil jalur panduan sebagai masukan, menggunakan literal dalam jalur panduan untuk melewati cabang yang telah dieksplorasi, dan mengeksplorasi cabang baru dalam pohon keputusan untuk model baru.

Contoh 4.1.6 Dalam Contoh 4.1.3, sebuah model ditemukan dengan literal keputusan $\bar{x}_1$ dan x_2 . Jalur panduan untuk model ini adalah $[\bar{x}_1 : 2, x_2 : 1]$, di mana label 2 setelah $\bar{x}_1$ berarti ini adalah cabang kedua dari x_1 . Label 1 setelah x_2 berarti ini adalah cabang pertama dari x_1 . Jika model berikutnya diperlukan, model tersebut harus berada di subpohon dengan jalur panduan awal $[\bar{x}_1 : 2, x_2 : 2]$. Dengan menggunakan $[\bar{x}_1 : 2, x_2 : 2]$, pemecah SAT menghindari ruang pencarian yang telah dilaluinya sebelumnya.

Karena pemecah SAT modern sering dirancang sebagai kotak hitam, akan sulit untuk menggunakannya sebagai pemecah inkremental untuk menghitung setiap model. Kita dapat memaksa pemecah SAT untuk menemukan model baru dengan memblokir semua model yang telah ditemukan sebelumnya. Sebuah model dapat dilihat sebagai konjungsi literal yang benar, satu untuk setiap variabel. Negasi dari konjungsi ini adalah klausa, yang disebut klausa pemblokiran. Jika klausa ini ditambahkan ke dalam himpunan klausa input, model akan diblokir oleh klausa ini dan tidak dapat dihasilkan oleh pemecah SAT.

Karena klausa pemblokiran berisi setiap variabel, kemungkinan besar tidak akan menjadi unit dan mahal untuk menyimpan semua klausa pemblokiran. Oleh karena itu,

diinginkan untuk mengurangi ukuran klausa pemblokiran, yaitu, untuk membangun klausa yang lebih pendek yang masih memblokir model. Jika kita mengetahui jalur panduan model ini, negasi dari konjungsi semua literal keputusan dalam jalur panduan akan memblokir model.

Banyak aplikasi pemecah SAT memerlukan penyelesaian urutan instance dengan menambahkan lebih banyak klausa. Pemecah SAT inkremental dengan jalur panduan dapat mendukung aplikasi ini tanpa modifikasi apa pun, karena ruang pencarian yang dilewati oleh jalur panduan tidak berisi model baru jika lebih banyak klausa ditambahkan.

Dalam banyak skenario aplikasi, bermanfaat untuk dapat melakukan beberapa pemeriksaan SAT pada klausa input yang sama di bawah interpretasi parsial paksa yang berbeda, yang disebut "*asumsi*." Misalnya, orang mungkin tertarik pada pertanyaan seperti "Apakah rumus F dapat dipenuhi dengan asumsi $x \rightarrow 1$ dan $y \rightarrow 0$?" Dalam aplikasi seperti itu, rumus masukan hanya dibaca sekali; pengguna mengimplementasikan loop iteratif yang memanggil instansiasi solver yang sama di bawah berbagai set asumsi. Panggilan tersebut dapat bersifat adaptif, yaitu, asumsi panggilan solver SAT di masa mendatang dapat bergantung pada hasil panggilan solver sebelumnya.

Solver SAT dapat mempertahankan keadaan internalnya dari panggilan sebelumnya ke panggilan berikutnya. *DPLL* dengan jalur panduan dapat dengan mudah dimodifikasi untuk mendukung aplikasi tersebut, memperlakukan asumsi sebagai serangkaian klausa unit khusus yang dapat ditambahkan atau dihapus dari solver. Jalur panduan memungkinkan kita untuk menelusuri kembali ke ruang pencarian baru tetapi tidak pernah mengunjungi kembali ruang pencarian yang telah dikunjungi sebelumnya.

Untuk mendapatkan solver SAT berkinerja tinggi, kita dapat menggunakan teknik penghapusan yang disajikan di Bagian 3.3 sebagai teknik pra-pemrosesan. Hal ini memungkinkan kita untuk mengurangi ukuran formula sebelum meneruskannya ke *DPLL*.

Di dalam badan *DPLL*, ia memanggil dirinya sendiri dua kali, sekali dengan $S \cup \{(A)\}$ dan sekali dengan $S \cup \{(A)\}$. Struktur data persisten untuk S akan cepat kehabisan memori karena S sangat besar dan pohon keputusan sangat besar untuk contoh SAT praktis. Pendekatan yang lebih baik adalah menggunakan struktur data destruktif untuk S : Kita mengingat semua operasi yang dilakukan pada S di dalam panggilan pertama, dan kemudian membatalkannya, jika perlu, ketika kita memasuki panggilan kedua.

BCPw: Penggunaan Literal Pengawas dalam DPLL

Dalam Algoritma 3.4.16, kami menyajikan implementasi *BCP* yang efisien sebagai prosedur pengambilan keputusan untuk klausa Horn. Versi *BCP* ini disebut *BCPht*, yang menggunakan struktur data kepala/ekor untuk setiap klausa non-unit. Secara alami, *BCP* ini dapat digunakan untuk mendukung *DPLL* dengan sedikit modifikasi. Artinya, *BCP* dapat mengimplementasikan ide literal pengawas, yang didasarkan pada literal kepala/ekor dan digunakan dalam pemecah SAT chaff oleh Matthew Moskewicz dkk. di Universitas Princeton.

Fungsi literal pengawas sama dengan literal kepala/ekor: Menggunakan dua literal untuk memutuskan apakah suatu klausa menjadi unit atau bertentangan, seperti yang dinyatakan dalam Proposisi 3.4.15.

Literal kepala/ekor adalah literal pengawas; ketika literal kepala/ekor diizinkan muncul di mana saja dalam sebuah klausa, literal kepala/ekor identik dengan literal pengawas. Sebagai literal pengawas, kita tidak tahu literal dalam sebuah klausa sebelum atau sesudah literal kepala/ekor bernilai salah atau tidak ketika *BCP* digunakan untuk mendukung *DPLL*. Dengan demikian, kita perlu mencari seluruh klausa untuk menemukan literal kepala/ekor berikutnya.

Karena ide literal pengawas didasarkan pada literal kepala/ekor, implementasi literal pengawas mudah jika kita memiliki implementasi literal kepala/ekor. Dalam *BCPht*, ketika literal kepala atau ekor dari c menjadi salah, kita mencari penggantinya pada indeks x , di mana $head(c) < x < tail(c)$. Ada dua cara alternatif untuk memodifikasi *BCPht*. Salah satu solusinya adalah mengizinkan $x \in \{0, 1, \dots, |c| - 1\} - \{head(c), tail(c)\}$; kita akan memperkenalkan implementasi solusi ini sebentar lagi. Solusi lainnya adalah dengan menetapkan $head(c) = 0$ dan $tail(c) = 1$ (or $tail(c) = |c| - 1$) dan menukar elemen $lits(c)$ jika perlu.

Solusi kedua menghemat ruang untuk $head(c)$ dan $tail(c)$ tetapi mungkin membutuhkan biaya lebih untuk mengunjungi literal $lits(c)$. Misalnya, jika sebuah klausa memiliki k literal yang menjadi salah dalam urutan dari kiri ke kanan selama *BCP*. Solusi kedua mungkin membutuhkan waktu $O(k^2)$ untuk mengunjungi literal klausa, sedangkan solusi pertama membutuhkan waktu $O(k)$. Akibatnya, *BCP* tidak dapat menjadi prosedur pengambilan keputusan waktu linier untuk klausa Horn jika solusi kedua digunakan. Tentu saja, kita dapat memperkenalkan pointer dalam klausa untuk mengingat posisi literal terakhir yang dikunjungi, sehingga solusi kedua masih memberi kita algoritma waktu linier.

Algoritma 4.1.7 *BCPw(U)* mendukung implementasi *BCP(C)* berdasarkan struktur data head/tail. *BCPw(U)* bekerja dengan cara yang sama seperti *BCPht(U)*, di mana U adalah tumpukan klausa unit dan klausa non-unit disimpan menggunakan struktur data head/tail. Seperti *BCPht*, *BCPw* mengembalikan klausa yang bertentangan jika klausa kosong dihasilkan; ia mengembalikan "SAT" jika tidak. Tidak seperti *BCPht*, nilai $head(c)$ dan $tail(c)$ dapat berupa indeks valid apa pun dari $lits(c)$.

```

proc BCPw(U)
1  while  $U \neq \{\}$  do
2     $A := pop(U)$ 
3    if  $val(A) = 0$  return  $reason(\bar{A})$  // a conflicting clause is found.
4    else if ( $val(A) \neq 1$ )
5       $val(A) := 1$ ;  $val(\bar{A}) := 0$ 
6    for  $c \in cls(\bar{A})$  do //  $\bar{A}$  is either head or tail of  $c$ .
7      if ( $\bar{A} = lits(c)[head(c)]$ )
8         $e_1 := head(c)$ ;  $e_2 := tail(c)$ ;  $step := 1$  // scan from left to right.
9      else
10        $e_1 := tail(c)$ ;  $e_2 := head(c)$ ;  $step := -1$  // scan from right to left.
11     while true do
12        $x := (x + step) \bmod |c|$  //  $x$  takes any valid index of  $lits(c)$ .

```



```

13      if  $x = e_1$  break // exits the inner while loop.
13'     if  $x = e_2$  continue // go to line 12.
14      $B := lits(c)[x]$ 
15     if ( $val(B) \neq 0$ )
16          $remove(c, cls(\bar{A})); insert(c, cls(B))$ 
17         if ( $step = 1$ )  $head(c) := x$  else  $tail(c) := x$ 
18         break // exit the inner while loop.
19     if ( $x = e_1$ ) // no new head or tail found.
20      $A := lits(c)[e_2]$  //  $c$  is unit or conflicting.
21     if  $val(A) = 0$  return  $c$  //  $c$  is conflicting.
22     else if ( $val(A) \neq 1$ ) //  $c$  is unit.
23          $push(A, U); reason(A) := c; val(A) := 1; val(\bar{A}) := 0$ 
24     return "SAT" // no empty clauses are found

```

Algoritma di atas berasal dari *BCPht* (Algoritma 3.4.16) dengan modifikasi berikut, yang memungkinkan nilai $head(c)$ dan $tail(c)$ menjadi indeks valid apa pun dari $lits(c)$:

- Pada baris 12, " $x := x + step$ " diganti dengan " $x := (x + step) \bmod |c|$."
- Pada baris 13, kondisi " $x = e_2$ " diganti dengan " $x = e_1$."
- Pada baris 13 ditambahkan untuk melewati literal yang diawasi oleh e_2 ; ini adalah literal yang akan berada di dalam klausa unit jika klausa tersebut menjadi unit.
- Pada baris 19, kondisi " $x = e_2$ " diganti dengan " $x = e_1$." Ketika $x = e_1$ benar, setiap literal dari klausa tersebut salah, kecuali literal yang diawasi oleh e_2 .

Dalam *DPLL*, pencatatan yang diperlukan untuk mendeteksi kapan suatu klausa menjadi unit dapat melibatkan beban komputasi yang tinggi jika diimplementasikan secara naif. Karena cukup untuk mengamati dua literal yang belum diberi nilai di setiap klausa, pemberian nilai pada literal yang tidak diamati dapat diabaikan dengan aman. Ketika variabel p diberi nilai 1, pemecah SAT hanya perlu mengunjungi klausa yang diamati oleh $\bar{p}$.

Setiap kali salah satu literal yang diamati menjadi salah, pemecah memilih salah satu literal yang belum diberi nilai untuk diamati. Jika ini tidak memungkinkan, klausa tersebut pasti merupakan unit, atau bertentangan, atau sudah terpenuhi di bawah pemberian nilai parsial saat ini. Setiap urutan pemberian nilai yang membuat klausa menjadi unit akan mencakup pemberian nilai salah satu literal yang diamati. Beban komputasi dari strategi ini relatif rendah: Dalam formula dengan m klausa dan n variabel, $2m$ literal perlu diamati, dan m/n klausa dikunjungi per pemberian nilai rata-rata. Keuntungan ini diwarisi dari ide literal kepala/ekor.

Keuntungan utama dari literal yang diamati dibandingkan literal kepala/ekor adalah bahwa literal yang diamati tidak perlu diperbarui saat melakukan backtracking di *DPLL*. Artinya, keunggulan utama *BCPw* dibandingkan *BCPht* adalah penggunaannya di dalam *DPLL*: Saat Anda melakukan backtracking dari panggilan rekursif *DPLL*, Anda tidak perlu membatalkan operasi pada $head(c)$ dan $tail(c)$, karena dua posisi c mana pun sudah cukup untuk memeriksa apakah c menjadi unit.

Implementasi Iteratif DPLL

Karena kompiler akan mengkonversi prosedur rekursi menjadi prosedur iteratif menggunakan tumpukan, untuk menghindari biaya konversi ini, dalam praktiknya, prosedur

DPLL tidak diimplementasikan melalui rekursi tetapi secara iteratif menggunakan tumpukan untuk interpretasi parsial σ . Jika struktur data *head/tail* digunakan, σ dapat diperoleh dari $val(a)$, tetapi kita masih membutuhkan tumpukan literal untuk mencatat waktu ketika literal diberi nilai. Tumpukan ini disebut *trail*, yang juga dapat berfungsi sebagai interpretasi parsial. Dengan menggunakan *trail*, kita melacak literal, baik literal keputusan maupun literal implisit, dari akar pohon keputusan hingga simpul saat ini. Setiap kali literal diberi nilai, baik oleh keputusan maupun implikasi di dalam BCP, literal tersebut dimasukkan ke dalam *trail* dan levelnya dalam pohon keputusan diingat.

Suatu jalur dapat mengarah ke jalan buntu, yaitu, menghasilkan klausa yang bertentangan, dalam hal ini kita harus menjelajahi cabang alternatif dari pemisahan kasus. Ini sesuai dengan penelusuran balik yang membalikkan nilai kebenaran dari literal keputusan. Ketika kita menelusuri balik dari panggilan rekursif, kita mengeluarkan literal tersebut dari jalur dan menjadikannya sebagai "tidak ditetapkan" (dilambangkan dengan $\times$). Ketika menelusuri balik cukup banyak kali, algoritma pencarian akhirnya akan menghabiskan semua cabang pohon keputusan, baik menemukan semua model atau menyatakan tidak ada model yang tersedia.

Untuk mengimplementasikan ide di atas, kita menggunakan jalur σ yang merupakan tumpukan literal dan berfungsi sebagai interpretasi parsial. σ akan menyimpan semua literal yang ditetapkan. Untuk setiap literal A , kita menggunakan $lvl(A)$ untuk mencatat level di mana A diberi nilai kebenaran. Ini sebenarnya dilakukan pada baris 5 dari *BCPw*. Baris 5 dari *BCPw* sekarang dibaca sebagai berikut:

$$5 : val(A) := 1; val(\bar{A}) := 0; lvl(A) := lvl(\bar{A}) := level; push(A, \sigma)$$

di mana val , lvl , σ , dan $level$ adalah variabel global.

Dalam praktiknya, lebih mudah untuk menetapkan level anak kanan sama dengan level induknya dalam pohon keputusan, untuk menunjukkan bahwa anak kanan tidak memiliki alternatif untuk dipertimbangkan. Dalam implementasi *DPLL* berikut, kami telah menggunakan ide ini.

Singkatnya, $BCP(C)$ yang digunakan dalam *DPLL* (Algoritma 4.1.1) menyederhanakan C menjadi (U, S) , di mana U adalah himpunan klausa unit dan S adalah himpunan klausa non-unit. Sekarang, klausa dalam S disimpan dalam struktur data kepala/ekor dan klausa dalam U disimpan dalam jejak σ . Kita mengganti $BCP(C)$ dengan $BCPw(X)$ untuk mendapatkan versi *DPLL* berikut, di mana X awalnya berisi klausa unit dari input dan berisi klausa unit yang dibuat oleh setiap literal keputusan selama eksekusi *DPLL*.

Algoritma 4.1.8 *DPLL(C)* non-rekursif mengambil input berupa himpunan C klausa dan mengembalikan model C jika C dapat dipenuhi; ia mengembalikan $\perp$ jika C tidak dapat dipenuhi. Prosedur *initialize(C)* akan mengembalikan klausa unit C dan menyimpan klausa lainnya dalam struktur data *head/tail*. Variabel global val , lvl , σ , dan $level$ digunakan di semua prosedur.

```

proc DPLL(C)
1  U := initialize(C) // initialize the head/tail data structure
2   $\sigma$  := {}; level := 0 // initialize  $\sigma$  (partial model of C) and level
3  while (true) do
4    if (BCPw(U) =  $\perp$ )
5      if (level = 0) return  $\perp$  // C is unsatisfiable
6      level := level – 1 // level of the last decision literal
7      A := undo() // A is the last decision literal picked at line 10
8      U := { $\bar{A}$ } // the second branch of A
9    else
10     A := pickLiteral() // pick an unassigned A as new decision literal
11     if (A = nil) return  $\sigma$  // all literals assigned, a model is found
12     level := level + 1 // new level in the first branch of A
13     U := {A} // create a new unit clause from A

proc undo()
  // Backtrack to level, undo all assignments to literals of higher levels.
  // Return the last literal of level + 1 in the trail  $\sigma$ , or nil.
1  A := nil
2  while ( $\sigma \neq \{\}$ )  $\wedge$  (lvl(top( $\sigma$ )) > level) do
3    A := pop( $\sigma$ )
4    val(A) := val( $\bar{A}$ ) :=  $\times$  //  $\times$  means “unassigned”
5  return A

```

Prosedur *undo*() akan membatalkan semua penetapan (yang dilakukan pada baris 5 dari *BCP*) ke literal *level* yang lebih tinggi dari *level* (termasuk komplemennya), dan mengembalikan literal terakhir dari *level* + 1 di σ , yang harus merupakan literal keputusan terakhir *A* yang dipilih pada baris 10 dari *DPLL*.

Prosedur *pickLiteral*() (baris 10) adalah tempat untuk mengimplementasikan berbagai heuristik strategi pencarian terbaik. Prosedur ini akan memilih literal yang belum ditetapkan dan menetapkan nilai kebenaran. Seperti yang kita ketahui dari Teorema 4.1.5, pilihan ini tidak berdampak pada kelengkapan algoritma pencarian. Namun, hal ini memiliki dampak signifikan pada kinerja pemecah masalah, karena pilihan ini berperan penting dalam memangkas ruang pencarian. Kita akan membahas masalah ini nanti.

4.2 PEMBELAJARAN KLAUSA BERBASIS KONFLIK (CDCL)

Bagian ini memperkenalkan klausa yang bertentangan sebagai informasi sumber untuk mencegah eksplorasi berulang dari ruang pencarian yang sia-sia. Kita mulai dengan contoh yang memotivasi.

Contoh 4.2.1 Misalkan *C* adalah himpunan klausa atas himpunan variabel $\{x_1, x_2, \dots, x_n\}$ dan *C* memiliki lebih dari satu juta model. Misalkan $C' = C \cup S$, di mana $S = \{(y_1 | y_2), (y_1 | \bar{y}_2), (\bar{y}_1 | y_2), (\bar{y}_1 | \bar{y}_2)\}$. Maka *C'* tidak memiliki model, karena empat klausa tambahan tidak dapat dipenuhi. Sekarang kita

memasukkan C' ke $DPLL$ dan mengasumsikan bahwa *pickLiteral* akan lebih menyukai x_i daripada y_1 dan y_2 . Setiap kali tepat sebelum $DPLL$ memilih y_1 untuk pemisahan kasus, sebuah model C ditemukan pada titik tersebut. Jika y_1 benar, maka salah satu dari $(\overline{y_1} | y_2)$ dan $(\overline{y_1} | \overline{y_2})$ akan saling bertentangan. Jika y_1 salah, maka salah satu dari $(y_1 | y_2)$ dan $(y_1 | \overline{y_2})$ akan saling bertentangan. Kedua nilai y_1 akan menghasilkan klausa yang bertentangan dalam S , sehingga $DPLL$ akan melakukan penelusuran balik. Karena C memiliki lebih dari satu juta model, ini berarti $DPLL$ akan mencoba menunjukkan bahwa S tidak dapat dipenuhi lebih dari satu juta kali.

Jika kita dapat belajar dari klausa-klausa yang bertentangan dalam S , kita dapat menghindari pengulangan pekerjaan yang sama berulang kali. Inilah motivasi dari teknik pembelajaran klausa berbasis konflik (*CDCL*): Pelajari informasi baru dari klausa-klausa yang bertentangan dan gunakan informasi baru tersebut untuk menghindari upaya pencarian yang sia-sia. Dengan menggunakan *CDCL*, informasi baru diperoleh dalam bentuk klausa yang kemudian ditambahkan kembali ke dalam prosedur pencarian. Untuk contoh di atas, ketika y_1 bernilai benar berdasarkan pemisahan kasus, $(\overline{y_1} | y_2)$ menjadi satu, sehingga y_2 harus diatur ke 1. Sekarang $(\overline{y_1} | \overline{y_2})$ menjadi bertentangan. Resolusi antara kedua klausa ini pada y_2 akan menghasilkan klausa baru, yaitu $(\overline{y_1})$.

Klausul baru ini tidak ada hubungannya dengan penugasan x_i , jadi kita dapat kembali ke level 0 $DPLL$ dan BCP selanjutnya akan menemukan klausa kosong dari $S \wedge y_1$ dan $DPLL$ mengembalikan $\perp$, sehingga menghindari pencarian model di C jutaan kali.

Menghasilkan Klausa dari Klausa yang Bertentangan

Ingat bahwa dalam Algoritma 4.1.7, ketika klausa non-unit c menjadi unit, di mana c berisi literal a yang belum diberi nilai, kita melakukan $reason(a) := c$ pada baris 23, yaitu, c adalah alasan mengapa literal a yang belum diberi nilai menjadi benar. Ketika klausa yang bertentangan ditemukan dalam $DPLL$, semua literalnya salah, dan kita memilih literal a , yang diberi nilai terakhir dalam c , dan melakukan resolusi pada a antara klausa ini dan $reason(a)$, dengan harapan resolven akan berguna untuk memangkas ruang pencarian.

Karena resolusi bersifat valid, klausa baru yang dihasilkan dengan cara ini merupakan konsekuensi logis dari input dan aman untuk dipertahankan. Untuk alasan praktis, kita harus melakukan resolusi secara selektif dan menyimpan paling banyak satu klausa per klausa yang bertentangan.

Algoritma 4.2.2 *conflictAnalysis(c)* mengambil input berupa klausa yang bertentangan c dalam interpretasi parsial saat ini pada *level* saat ini, menghasilkan klausa baru melalui resolusi, dan mengembalikannya sebagai output. Prosedur *latestAssignedLiteral(c)* akan mengembalikan literal a dari c sedemikian sehingga a adalah literal terakhir yang memberikan nilai kebenaran pada c . Prosedur *countLevel(c, lvl)* akan mengembalikan jumlah literal dalam

c yang diberi nilai pada level lvl . Prosedur $resolve(c_1, c_2)$ mengembalikan resolven dari c_1 dan c_2 .

```

proc conflictAnalysis( $c$ )
    //  $c$  is conflicting in the partial interpretation  $val()$ 
    1   $\alpha := c$ 
    2  while ( $|\alpha| > 1$ ) do // as  $\alpha$  has more than one literal
    3     $a := latestAssignedLiteral(\alpha)$  //  $a$  is assigned last in  $\alpha$ 
    4     $\alpha := resolve(\alpha, reason(\bar{a}))$  // resolution on  $a$ 
    5    if ( $countLevel(\alpha, level) \leq 1$ ) return  $\alpha$ 
    6  return  $\alpha$ 

```

Contoh 4.2.3 Misalkan klausa input $C = \{c_1 : (x_2 | x_3), c_2 : (x_1 | x_4), c_3 : (x_2 | x_4), c_4 : (x_1 | x_2 | x_3)\}$. Jika kita memilih x_1 sebagai literal keputusan pertama dalam $DPLL$, maka akan menghasilkan penugasan berikut:

<i>assignment</i>	<i>reason</i>
$x_1@1$	(x_1)
$\bar{x}_4@1$	$c_2 : (x_1 x_4)$
$\bar{x}_2@1$	$c_3 : (x_2 x_4)$
$\bar{x}_3@1$	$c_1 : (x_2 x_3)$

Klausul c_4 menjadi bertentangan dan berisi tiga literal level 1.

Kita sebut $conflictAnalysis(c_4)$: Misalkan α_0 adalah c_4 ; resolusi berikut dilakukan:

1. $\alpha_1 : (\bar{x}_1 | x_2) = resolve(\alpha_0, c_1)$.
2. $\alpha_2 : (\bar{x}_1 | x_4) = resolve(\alpha_1, c_3)$.
3. $\alpha_3 : (\bar{x}_1) = resolve(\alpha_2, c_2)$.

α_3 akan dikembalikan oleh $conflictAnalysis(c_4)$.

Proposisi 4.2.4 Dengan asumsi c bertentangan dalam interpretasi saat ini:

- (a) Prosedur $conflictAnalysis(c)$ akan berakhir dan mengembalikan klausa α yang merupakan implikasi dari klausa input.
- (b) α adalah klausa yang bertentangan dalam interpretasi saat ini.
- (c) Jika kita melakukan backtracking dari $level$ saat ini, α akan menjadi klausa unit dalam interpretasi setelah backtracking.

Bukti

- a. (a): Kita hanya menetapkan sejumlah literal yang terbatas dengan nilai kebenaran pada level saat ini. Literal yang kita pilih untuk diselesaikan berada dalam urutan terbalik ketika ditetapkan. Setelah diselesaikan, literal ini tidak akan pernah kembali ke klausa baru; dengan demikian, tidak ada kemungkinan terjadinya loop. Karena α dihasilkan oleh resolusi dari klausa input, α harus merupakan implikasi dari klausa input.

- b. (*b*): Faktanya, setiap resolven dalam *conflictAnalysis(c)* adalah klausa yang bertentangan dalam interpretasi saat ini. Karena *c* adalah klausa yang bertentangan, setiap literal bernilai salah dalam interpretasi saat ini. *c* diselesaikan dengan klausa alasan yang hanya memiliki satu literal yang benar dalam interpretasi, dan literal yang benar ini diselesaikan selama resolusi. Resolven yang dihasilkan hanya akan berisi literal yang salah dalam interpretasi saat ini. α adalah resolven terakhir yang dihasilkan dalam prosedur.
- c. (*c*): Prosedur *conflictAnalysis(c)* akan berakhir ketika satu literal, misalnya *B*, dari level saat ini tetap ada di α . Jika kita melakukan backtracking dari level saat ini, *B* akan tidak diberi nilai dan literal lainnya akan bernilai false menurut (*b*); dengan demikian, α adalah klausa unit dalam interpretasi setelah backtracking.

DPLL dengan CDCL

Prosedur *DPLL* baru dengan *CDCL* (pembelajaran klausa berbasis konflik) dapat dijelaskan sebagai berikut: Kita akan mengganti *BCPw(U)* dengan *BCP(U)* yang merupakan modifikasi kecil dari *BCPw(U)*: Alih-alih mengembalikan $\perp$ ketika ditemukan klausa yang bertentangan, cukup kembalikan klausa yang bertentangan tersebut.

Perbedaan utama dari algoritma *DPLL* baru terletak pada baris 7–8: Dalam Algoritma 4.1.8, klausa unit baru dibuat dari literal keputusan terakhir; dalam versi baru, klausa baru dibuat dari klausa yang bertentangan dan klausa baru ini akan menjadi unit setelah operasi undo.

Algoritma 4.2.5 *DPLL(C)* mengambil sebagai input himpunan *C* klausa dan mengembalikan model *C* jika *C* dapat dipenuhi; ia mengembalikan $\perp$ jika *C* tidak dapat dipenuhi. Prosedur *initialize(C)* akan mengembalikan klausa unit dari *C* dan menyimpan klausa sisanya dalam struktur data *head/tail*. Variabel global *val*, *lvl*, σ , dan level digunakan seperti pada Algoritma 4.1.8.

proc DPLL(C)

```

1  U := initialize(C) // initialize the head/tail data structure
2   $\sigma$  := {}; level := 0 // initialize  $\sigma$  (partial model of C) and level
3  while (true) do
4      res := BCP(U) // U is a set of unit clauses.
5      if (res  $\neq$  "SAT") // res contains a conflicting clause
6          if (level = 0) return  $\perp$  // C is unsatisfiable.
7          U := insertNewClause(conflictAnalysis(res)) // new level is set
8          undo() // undo to the new level
9      else
10         A := pickLiteral() // pick an unassigned A as new decision literal
11         if (A = nil) return  $\sigma$  // all literals assigned, a model is found
12         level := level + 1 // new level in the first branch of A
13         U := {A} // create a new unit clause from A
```

proc insertNewClause(α)

```

1  if ( $|\alpha| = 1$ ) //  $\alpha$  is a unit clause
2       $level := 0$ 
3       $A := literal(\alpha)$ ;  $reason(A) := \alpha$  // assume  $\alpha = (A)$ 
4      return  $\{A\}$ 
5  else // insert  $\alpha$  into the head/tail data structure for clauses
6       $lits(\alpha) := makeArray(\alpha)$  // create an array of literals
7       $secondHigh := 0$  // look for the second highest level in  $\alpha$ 
8      for  $x := 0$  to  $|\alpha| - 1$  do
9          if ( $lvl(lits(\alpha)[x]) = level$ )  $head(\alpha) := x$ ;  $b := lits(\alpha)[x]$  //  $lvl(b) = level$ 
10         else if ( $lvl(lits(\alpha)[x]) > secondHigh$ )  $secondHigh := lvl(lits(\alpha)[x])$ ;  $t := x$ 
11          $tail(\alpha) := t$ 
12          $level := secondHigh$ 
13          $reason(b) := \alpha$ 
14     return  $\{b\}$  // the head literal of  $\alpha$ 

```

Prosedur *insertNewClause* akan mengembalikan satu set literal yang berfungsi sebagai klausa unit baru dan mengatur ulang variabel *level* dari *DPLL*. Jika klausa baru α yang dihasilkan oleh *conflictAnalysis* adalah klausa unit, klausa ini akan dikembalikan, dan level diatur ke 0. Jika klausa baru bukan unit, berdasarkan Proposisi 4.2.4, hanya ada satu literal dari level saat ini di α , dan literal ini akan menjadi literal kepala dari α ketika level diturunkan. Kemudian kita memilih satu literal dari level tertinggi selain literal kepala sebagai literal ekor. Level akan diatur ulang ke level literal ekor. Ketika kita kembali ke level ini, berdasarkan Proposisi 4.2.4, α menjadi unit dan merupakan alasan literal kepala, yang belum ditetapkan pada titik ini, untuk ditetapkan benar.

Contoh 4.2.6 Perhatikan himpunan klausa C dari klausa-klausa berikut:

$$\begin{array}{ll}
 c_1 : (x_1 | x_2) & c_2 : (x_1 | x_3 | \overline{x_7}) \\
 c_3 : (\overline{x_2} | \overline{x_3} | x_4) & c_4 : (\overline{x_4} | x_5 | x_6) \\
 c_5 : (\overline{x_3} | \overline{x_4}) & c_6 : (x_5 | \overline{x_6}) \\
 c_7 : (\overline{x_2} | x_3) & c_8 : (\overline{x_1} | x_3) \\
 c_9 : (\overline{x_1} | x_2 | \overline{x_3}) &
 \end{array}$$

Jika tiga literal pertama yang dipilih oleh *pickLiteral* di *DPLL* adalah $\overline{x_7}$, $\overline{x_6}$, dan $\overline{x_5}$, maka satu-satunya literal yang tersirat adalah $\overline{x_4}$ @3 oleh c_4 . Literal keputusan berikutnya adalah $\overline{x_1}$, yang menyiratkan penugasan berikut: x_2 @4 oleh c_1 dan $\overline{x_3}$ @4 oleh c_3 . c_7 menjadi bertentangan pada level 4.

Dengan memanggil *conflictAnalysis*(c_7), kita memperoleh $(\overline{x_2} | x_4)$ melalui resolusi antara c_7 dan c_3 , yang berisi x_2 sebagai satu-satunya literal pada level 4. Kita menambahkan $(\overline{x_2} | x_4)$ ke *DPLL* sebagai c_{10} .

Level *DPLL* diatur ke 3 karena $lvl(x_4) = 3$. Jadi, kita melakukan backtrack ke level 3, dan $c_{10} = (\overline{x_2} | x_4)$ menjadi unit. Panggilan berikutnya ke *BCP* akan membuat penugasan

berikut pada *level* 3: $\bar{x}_2@3$ oleh $c_{10}, x_1@3$ oleh $c_1, x_3@3$ oleh c_8 , dan c_9 menjadi bertentangan.

Dengan memanggil *conflictAnalysis*(c_9), kita memperoleh $c_{11} = (x_2)$ melalui dua resolusi antara c_8, c_9 , dan c_1 . Klausa baru adalah unit dan kita menetapkan *level* = 0. Dengan melakukan backjumping ke *level* 0 dengan undo, panggilan berikutnya ke *BCP* akan menyiratkan penugasan berikut: $x_2@0$ oleh $c_{11}, x_3@0$ oleh $c_7, x_4@0$ oleh c_{10} , dan c_5 menjadi bertentangan.

Dengan memanggil *conflictAnalysis*(c_5), kita memperoleh $c_{12} = (\bar{x}_2)$ melalui dua resolusi antara c_5, c_{10} , dan c_7 . Klausa c_{11} dan c_{12} akan menghasilkan klausa kosong. Dengan demikian, klausa input tidak dapat dipenuhi.

Pembelajaran klausa dengan analisis konflik tidak mengurangi kelengkapan algoritma pencarian: Bahkan jika klausa yang dipelajari dihapus di kemudian hari selama pencarian, jejaknya menjamin bahwa pemecah masalah tidak pernah berulang kali memasuki tingkat keputusan dengan penugasan parsial yang sama. Kami telah menunjukkan kebenaran pembelajaran klausa dengan menunjukkan bahwa setiap klausa yang bertentangan tersirat oleh rumus aslinya.

Ide *CDCL* pertama kali digunakan dalam pemecah masalah grasp karya Marques-Silva dan Sakallah, dan pemecah masalah relsat karya Bayardo dan Schrag. Dalam pemecah masalah mereka, mereka secara independen memperkenalkan mekanisme baru untuk menganalisis konflik yang ditemui selama pencarian penugasan yang memuaskan. Ada banyak cara untuk menghasilkan klausa baru dari klausa yang bertentangan dan sebagian besar didasarkan pada grafik implikasi. Metode pembelajaran klausa baru melalui resolusi adalah yang paling mudah untuk disajikan.

CDCL menghadirkan perubahan revolusioner pada *DPLL* sehingga *DPLL* bukan lagi prosedur pencarian mundur sederhana, karena *level* yang direset oleh *insertNewClause* mungkin bukan *level* — 1 dan *DPLL* akan melompat mundur ke *level* yang lebih rendah dari *level* — 1 (ke node yang lebih dekat ke akar pohon keputusan *DPLL*), sehingga menghindari ruang pencarian yang tidak perlu. Jika klausa baru yang dipelajari berisi literal tunggal, *level* baru diatur ke 0. *DPLL* dengan *CDCL* dapat belajar dari kegagalan dan melompat mundur ke *level* tempat sumber kegagalan berasal. Beberapa teknik baru diusulkan berdasarkan *CDCL*, termasuk menghasilkan bukti resolusi ketika klausa input tidak dapat dipenuhi, atau memulai ulang pencarian secara acak.

Inti yang Tidak Dapat Dipenuhi

Keberadaan bukti resolusi untuk sekumpulan klausa yang tidak dapat dipenuhi dijamin oleh kelengkapan resolusi. Menemukan bukti resolusi adalah pekerjaan yang sulit. Telah ditunjukkan bahwa *CDCL* seperti yang dipraktikkan dalam pemecah SAT saat ini sesuai dengan sistem pembuktian yang jauh lebih ampuh daripada resolusi. Secara khusus, setiap langkah pembelajaran sebenarnya adalah urutan langkah resolusi, di mana klausa yang dipelajari adalah resolven terakhir; sebaliknya, bukti resolusi dapat diperoleh dengan menganggap prosedur pembelajaran sebagai prosedur resolusi terpandu. Dalam Contoh 4.2.6, kita telah

melihat bagaimana klausa kosong dihasilkan. Bukti resolusi yang mengarah ke klausa kosong dapat ditampilkan sebagai berikut:

$c_1 : (x_1 \mid x_2)$	Assumed
$c_3 : (\overline{x_2} \mid \overline{x_3} \mid x_4)$	Assumed
$c_5 : (\overline{x_3} \mid \overline{x_4})$	Assumed
$c_7 : (\overline{x_2} \mid x_3)$	Assumed
$c_8 : (\overline{x_1} \mid x_3)$	Assumed
$c_9 : (\overline{x_1} \mid x_2 \mid \overline{x_3})$	Assumed
$c_{10} : (\overline{x_2} \mid x_4)$	resolvent of c_7, c_3
$c_{11} : (\overline{x_1} \mid x_2)$	resolvent of c_8, c_9
$c_{12} : (x_2)$	resolvent of c_{11}, c_1
$c_{13} : (\overline{x_2} \mid \overline{x_3})$	resolvent of c_5, c_{10}
$c_{14} : (\overline{x_2})$	resolvent of c_{13}, c_7
$c_{15} : ()$	resolvent of c_{12}, c_{14}

Contoh ini menunjukkan ide umum untuk mendapatkan bukti resolusi ketika *DPLL* menemukan sekumpulan klausa yang tidak dapat dipenuhi. Selama pencarian di dalam *DPLL*, klausa yang bertentangan dihasilkan karena literal keputusan. Prosedur *conflictAnalysis* menghasilkan klausa baru untuk menunjukkan mengapa keputusan sebelumnya salah dan kita naik dalam pohon keputusan sesuai dengan klausa baru ini. *DPLL* mengembalikan $\perp$ ketika akhirnya menemukan klausa yang bertentangan di level 0. Dalam hal ini, bukti resolusi dihasilkan oleh *conflictAnalysis*, dan bukti ini menggunakan klausa masukan atau klausa yang dihasilkan sebelumnya oleh *conflictAnalysis*.

Diberikan himpunan klausa C yang tidak dapat dipenuhi, kita dapat menggunakan semua langkah resolusi di dalam prosedur *conflictAnalysis* untuk membangun bukti resolusi. Jelas, bukti tersebut memberikan bukti untuk ketidakpuasan C . Klausa yang digunakan sebagai premis bukti adalah subset dari klausa C dan disebut inti yang tidak dapat dipenuhi dari bukti tersebut. Perhatikan bahwa suatu formula biasanya tidak memiliki inti yang tidak dapat dipenuhi yang unik. Setiap subset C yang tidak dapat dipenuhi adalah inti yang tidak dapat dipenuhi. Bukti resolusi dan inti yang tidak dapat dipenuhi memiliki aplikasi dalam verifikasi perangkat keras.

Inti yang tidak dapat dipenuhi (*unsatisfiable core*) bersifat minimal jika penghapusan klausa apa pun dari inti tersebut membuat klausa yang tersisa dalam inti tersebut dapat dipenuhi. Kita dapat menggunakan definisi ini untuk memeriksa apakah setiap klausa dalam inti tersebut diperlukan, dan ini jelas merupakan pekerjaan yang sulit ketika intinya sangat besar. Inti dari contoh di atas berisi setiap klausa input kecuali c_2 , c_4 , dan c_6 ; ini adalah inti minimal.

Restart Acak

Misalkan kita mencari model C dengan *DPLL*(C) dan literal keputusan pertama adalah A . Sayangnya, $C \wedge A$ adalah contoh yang sulit dipenuhi dan *DPLL* terjebak di sana tanpa keberhasilan. Memulai ulang *DPLL* secara acak mungkin satu-satunya pilihan Anda. Dengan

"restart," yang kami maksud adalah *DPLL* membuang semua literal keputusan sebelumnya (ini dapat dengan mudah dilakukan dengan $level := 0$; $undo()$ di *DPLL*). Dengan "acak," yang kami maksud adalah ketika Anda memulai ulang pencarian, literal keputusan pertama kemungkinan besar akan berbeda dari A . Secara intuitif, memulai ulang secara acak berarti ada peluang untuk menghindari nasib buruk dan mendapatkan keberuntungan lebih dalam menebak penugasan literal yang tepat yang akan mengarah pada solusi cepat.

Tanpa *CDCL*, memulai ulang secara acak membuat *DPLL* tidak lengkap. Dengan *CDCL*, klausa input dan yang dihasilkan mencegah *DPLL* memilih urutan literal keputusan yang sama dan tidak dapat menghasilkan klausa yang sama lagi. Karena jumlah klausa yang mungkin terbatas, *DPLL* tidak dapat berjalan selamanya. Ini adalah pandangan teoretis. Dalam praktiknya, kita tidak dapat menyimpan semua klausa yang dihasilkan karena prosedur akan kehabisan memori. Mengelola klausa yang dihasilkan dalam *DPLL* dengan *CDCL* adalah masalah penting yang tidak dibahas dalam buku ini.

Intuisi yang sama menunjukkan bahwa memulai ulang secara acak akan jauh lebih efektif ketika contoh masalah sebenarnya dapat dipenuhi. Hasil eksperimen menunjukkan bahwa memulai ulang secara acak juga membantu ketika klausa input tidak dapat dipenuhi. Jika restart sering dilakukan, ini mengasumsikan penyimpangan dari praktik standar, *CDCL* dapat sekuat resolusi umum, sementara *DPLL* tanpa restart diketahui sesuai dengan resolusi seperti pohon yang lebih lemah. Justifikasi teoritis untuk percepatan ini adalah bahwa restart memungkinkan pencarian untuk mendapatkan manfaat dari pengetahuan yang diperoleh tentang variabel yang terus-menerus bermasalah dan saling bertentangan lebih cepat daripada jika menggunakan backjumping yang memungkinkan penugasan parsial untuk direset dengan cara yang sama. Pada intinya, restart dapat memungkinkan penemuan bukti ketidakpuasan yang lebih pendek.

Untuk mengimplementasikan strategi restart, ketika *DPLL* telah menemukan sejumlah kegagalan (klausa yang saling bertentangan) tanpa keberhasilan (sebuah model), saatnya untuk melakukan restart. Saat ini, pemecah *SAT* sering menggunakan kebijakan restart berikut: Misalkan $r_i = r_0 \gamma^{i-1}$ adalah jumlah kegagalan untuk restart ke- i , di mana r_0 adalah bilangan bulat tertentu, dan $1 \leq \gamma < 2$. Jika $r_0 = 300$ dan $\gamma = 1,2$, artinya restart pertama terjadi ketika *DPLL* mengalami 300 kegagalan, restart kedua terjadi setelah 360 kegagalan lagi, dan seterusnya. Jika $\gamma = 1$, artinya *DPLL* melakukan restart setelah sejumlah kegagalan tetap.

Mengingat penggunaan umum fitur restart dalam solver *SAT* pembelajaran klausa saat ini, tugas memilih kebijakan restart yang baik tampak menarik. Meskipun hasil eksperimen menunjukkan bahwa tidak ada kebijakan restart yang lebih baik daripada yang lain untuk berbagai macam masalah, solver *SAT* pembelajaran klausa dapat memperoleh manfaat besar dari kebijakan restart yang dirancang dengan cermat. Secara khusus, eksperimen menunjukkan bahwa kebijakan restart yang tidak sepele berkinerja jauh lebih baik daripada jika fitur restart dinonaktifkan dan menunjukkan kinerja yang sangat berbeda satu sama lain. Hal ini memberikan motivasi untuk merancang kebijakan restart yang lebih baik, terutama yang dinamis berdasarkan jenis masalah dan statistik pencarian.

Heuristik Percabangan untuk DPLL

Baik menggunakan CDCL atau tidak, setiap implementasi *DPLL* membutuhkan fungsi untuk memilih literal untuk pemisahan kasus. Kita berasumsi bahwa ketika sebuah literal dipilih untuk pemisahan kasus, literal tersebut akan diberi nilai true pada cabang pertama *DPLL*. Sebelum pengembangan teknik CDCL, heuristik percabangan adalah metode utama yang digunakan untuk mengurangi ukuran ruang pencarian. Oleh karena itu, tampaknya peran heuristik percabangan kemungkinan akan berubah secara signifikan untuk algoritma yang memangkas ruang pencarian menggunakan pembelajaran klausa.

Sudah menjadi pengetahuan umum bahwa menguntungkan untuk menetapkan terlebih dahulu variabel yang paling ketat batasannya, yaitu variabel yang muncul dalam sejumlah besar klausa. Salah satu perwakilan dari strategi pemilihan tersebut dikenal sebagai aturan MOMS, yang bercabang pada literal yang memiliki kemunculan maksimum dalam klausa dengan ukuran minimum. Jika klausa tersebut berisi klausa biner, maka aturan MOMS akan memilih literal dalam klausa biner. Dengan menetapkan nilai benar pada literal ini, kemungkinan besar sejumlah klausa biner maksimal akan terpenuhi.

Di sisi lain, jika kriteria utama untuk pemilihan literal cabang adalah memilih literal yang memungkinkan terjadinya kaskade propagasi unit (hasil dari kaskade tersebut adalah submasalah yang lebih kecil), kita akan menetapkan nilai salah pada literal yang dipilih oleh aturan MOMS dalam klausa biner; menetapkan nilai salah pada literal ini kemungkinan akan menciptakan sejumlah klausa unit maksimal dari semua klausa biner. MOMS memberikan perkiraan kasar namun mudah dihitung terhadap jumlah klausa unit (yaitu, literal tersirat) yang mungkin disebabkan oleh penetapan variabel tertentu.

Sebagai alternatif, seseorang dapat memanggil BCP beberapa kali pada sekumpulan variabel yang menjanjikan secara bergantian dan menghitung jumlah pasti klausa unit yang akan disebabkan oleh pilihan percabangan. Setiap variabel yang dipilih ditetapkan sebagai benar dan salah secara bergantian dan BCP dieksekusi untuk setiap pilihan. Jumlah pasti propagasi unit yang disebabkan kemudian digunakan untuk mengevaluasi kemungkinan pilihan percabangan. Tidak seperti heuristik MOMS, aturan ini jelas tepat dalam upayanya untuk menilai jumlah klausa unit yang disebabkan oleh penugasan variabel potensial. Sayangnya, perhitungannya juga jauh lebih mahal karena beberapa eksekusi BCP dan pembatalannya. Telah ditunjukkan bahwa, dengan menggunakan MOMS untuk memilih sejumlah kecil kandidat yang menjanjikan, yang masing-masing kemudian dievaluasi secara tepat menggunakan BCP, ia mengungguli heuristik lain pada masalah yang dihasilkan secara acak.

Strategi lain adalah melakukan percabangan pada variabel yang kemungkinan besar merupakan literal tulang punggung. Literal tulang punggung adalah literal yang harus benar di semua model klausa input. Kemungkinan bahwa literal tertentu adalah literal tulang punggung didekati dengan menghitung kemunculan literal tersebut dalam klausa yang terpenuhi selama eksekusi *DPLL*. Heuristik ini mengungguli heuristik yang dibahas pada paragraf sebelumnya pada banyak contoh.

Pengembangan teknik CDCL memungkinkan pemecah masalah untuk mengatasi masalah yang lebih terstruktur dan realistis. Tidak ada studi formal yang membandingkan heuristik yang dibahas sebelumnya pada masalah terstruktur ketika teknik CDCL digunakan. Teknik CDCL menciptakan banyak klausa baru dan membuat penghitungan kemunculan setiap literal menjadi sulit atau kurang signifikan. Teknik percabangan dan pembelajaran sangat terkait, dan penambahan pembelajaran pada implementasi *DPLL* akan memiliki efek signifikan pada efektivitas strategi percabangan apa pun. Saat klausa baru dipelajari, jumlah klausa unit yang disebabkan oleh suatu penugasan dapat diharapkan bervariasi; sebaliknya juga benar bahwa pilihan literal keputusan dapat memengaruhi klausa yang dihasilkan.

Heuristik percabangan yang dirancang untuk berfungsi dengan baik dalam konteks pembelajaran klausa umumnya mencoba untuk bercabang pada literal di antara klausa baru yang telah dipelajari baru-baru ini. Hal ini cenderung memungkinkan eksekusi *DPLL* untuk terus "membuat kemajuan" pada satu bagian ruang pencarian daripada berpindah dari satu area ke area lain; manfaat tambahannya adalah bahwa klausa yang dipelajari cenderung tetap relevan, menghindari ketidakefisienan yang terkait dengan hilangnya informasi yang ada dalam klausa yang dipelajari yang menjadi tidak relevan dan dibuang.

Salah satu heuristik percabangan populer untuk *DPLL* dengan pembelajaran klausa disebut "heuristik jumlah individu terbesar dinamis (*DLIS*)" dan berperilaku mirip dengan aturan MOMS. Pada setiap titik keputusan, ia memilih penugasan yang memenuhi klausa yang paling tidak terpenuhi. Secara formal, misalkan p_x adalah jumlah klausa yang belum terselesaikan yang mengandung x dan n_x adalah jumlah klausa yang belum terselesaikan yang mengandung $\bar{x}$. Selain itu, misalkan x adalah variabel yang p_x -nya maksimal, dan misalkan y adalah variabel yang n_y -nya maksimal. Jika $p_x > n_y$, pilih x sebagai literal keputusan berikutnya; jika tidak, pilih $\bar{y}$. Kerugian dari strategi ini adalah overhead komputasinya tinggi: Algoritma perlu mengunjungi semua klausa yang berisi literal yang telah diatur ke benar untuk memperbarui nilai p_x dan n_x untuk semua variabel yang terdapat dalam klausa ini. Selain itu, proses tersebut perlu dibalik saat melakukan backtracking.

Heuristik yang umum digunakan dalam pemecah SAT kontemporer lebih menyukai literal dalam klausa konflik yang baru ditambahkan. Setiap literal dikaitkan dengan sebuah hitungan, yang diinisialisasi dengan jumlah kemunculan literal tersebut dalam himpunan klausa. Ketika klausa yang dipelajari ditambahkan, hitungan yang terkait dengan setiap literal dalam klausa tersebut akan bertambah. Secara berkala, semua penghitung dibagi dengan konstanta yang lebih besar dari 1, menghasilkan peluruhan yang menyebabkan bias terhadap percabangan pada variabel yang muncul dalam klausa yang baru dipelajari. Pada setiap titik keputusan, pemecah kemudian memilih literal yang belum ditetapkan dengan hitungan tertinggi (di mana jika terjadi seri, maka akan dipecah secara acak secara default).

Pendekatan ini, yang dikenal sebagai heuristik *variable state independent decaying sum* (*VSIDS*), pertama kali diimplementasikan dalam zchaff. Zchaff mempertahankan daftar literal yang belum ditetapkan yang diurutkan berdasarkan hitungan. Daftar ini hanya diperbarui ketika klausa yang dipelajari ditambahkan, sehingga menghasilkan overhead yang sangat rendah. Keputusan dapat dibuat dalam waktu konstan.

Heuristik yang digunakan dalam berkmin dibangun di atas ide ini tetapi merespons lebih dinamis terhadap klausa yang baru dipelajari. Heuristik berkmin lebih memilih untuk bercabang pada literal yang belum ditetapkan dalam klausa yang paling baru dipelajari yang belum terpenuhi.

Penekanan pada variabel yang terlibat dalam klausa yang bertentangan baru-baru ini mengarah pada pencarian berbasis lokalitas, yang secara efektif berfokus pada subruang. Subruang yang diinduksi oleh strategi pengambilan keputusan ini cenderung menyatu, menghasilkan lebih banyak peluang untuk penyelesaian klausa yang bertentangan, karena sebagian besar variabel bersifat umum. Merepresentasikan hitungan menggunakan variabel bilangan bulat menyebabkan sejumlah besar hitungan yang sama. Solver minisat menghindari masalah ini dengan menggunakan bilangan floating-point untuk merepresentasikan bobot. Strategi lain yang mungkin (tetapi jauh lebih kompleks) adalah hanya berkonsentrasi pada klausa yang bertentangan yang belum terselesaikan dengan mempertahankan tumpukan klausa yang bertentangan.

Ketika restart digunakan, heuristik percabangan harus merupakan kombinasi dari pemilihan acak dan heuristik lainnya, sehingga *DPLL* akan memilih literal cabang yang berbeda setelah setiap restart. Secara keseluruhan, ada banyak heuristik percabangan yang bersaing untuk solver kepuasan, dan masih banyak yang harus dilakukan dalam mengevaluasi efektivitas relatifnya dengan pembelajaran klausa pada masalah terstruktur yang realistis.

4.3 PENGGUNAAN PEMECAH SAT

Terdapat implementasi *DPLL* yang sangat baik dan Anda dapat menemukan pemecah SAT ini di halaman web kompetisi SAT internasional (satcompetition.org). Sebagian besar tersedia secara gratis, seperti minisat, glucose, lingeling, maple_LCM, dan beberapa lainnya. Pemecah SAT ini juga disebut generator model tujuan umum, karena mereka menerima CNF sebagai input dan banyak masalah dapat ditentukan dalam CNF karena SAT adalah NP-lengkap.

Jika Anda memiliki masalah pencarian yang sulit untuk dipecahkan, baik memecahkan teka-teki atau menemukan solusi di bawah batasan tertentu, Anda dapat menulis program tujuan khusus atau mendeskripsikan masalah Anda dalam CNF dan menggunakan salah satu generator model tujuan umum ini. Meskipun alat tujuan khusus dan generator model tujuan umum sama-sama bergantung pada pencarian menyeluruh, ada perbedaan mendasar.

Untuk yang terakhir, setiap masalah memiliki representasi internal yang seragam (yaitu, klausa untuk pemecah SAT). Representasi seragam ini dapat menimbulkan redundansi dan inefisiensi. Namun, karena mesin pencari tunggal digunakan untuk semua masalah, setiap peningkatan pada mesin pencari tersebut sangat signifikan. Terlebih lagi, kita telah mengumpulkan pengetahuan selama tiga dekade tentang bagaimana membuat mesin pencari seragam tersebut efisien. Penggunaan generator model tujuan umum untuk memecahkan masalah kombinatorial yang sulit patut mendapat perhatian setidaknya karena dua alasan.

- Jauh lebih mudah untuk menentukan masalah dalam CNF daripada menulis program khusus untuk menyelesaikannya. Demikian pula, menyempurnakan spesifikasi jauh lebih mudah daripada menyempurnakan program khusus.
- Generator model tujuan umum dapat memberikan hasil yang kompetitif dan komplementer untuk masalah kombinatorial tertentu. Beberapa contoh akan diberikan berikut ini.

Menentukan Instans SAT dalam Format DIMACS

Saat ini, pemecah SAT menggunakan format yang disarankan oleh DIMACS (Pusat Matematika Diskrit dan Ilmu Komputer Teoretis) bertahun-tahun yang lalu untuk rumus CNF. Format DIMACS menggunakan file teks: Jika karakter pertama dari sebuah baris adalah "c," itu berarti komentar. CNF dimulai dengan baris " p cnf n m ," di mana n adalah jumlah variabel proposisional dan m adalah jumlah klausa. Sebagian besar pemecah SAT hanya mensyaratkan m sebagai batas atas untuk jumlah klausa. Setiap literal direpresentasikan oleh bilangan bulat: Bilangan bulat positif adalah literal positif dan bilangan bulat negatif adalah literal negatif. Klausa adalah daftar bilangan bulat yang berakhiran 0.

Contoh 4.3.1 Untuk merepresentasikan $C = \{(x_1 | x_2 | x_3), (x_1 | x_2), (x_2 | x_3), (x_2 | x_3)\}$ dalam format DIMACS, file teks akan terlihat seperti berikut: n

```
c a tiny example
1 2 -3 0
-1 2 0
-2 -3 0
2 3 0
```

Untuk menggunakan pemecah SAT saat ini, Anda perlu menyiapkan contoh SAT Anda dalam format DIMACS. Ini biasanya dilakukan dengan menulis program kecil yang disebut encoder. Setelah model ditemukan oleh pemecah SAT, Anda mungkin memerlukan program lain, yang disebut decoder, yang menerjemahkan model ke dalam format yang diinginkan. Misalnya, encoder untuk teka-teki Sudoku akan menghasilkan rumus A dalam CNF dari teka-teki tersebut dan decoder akan mengambil model A dan menghasilkan solusi Sudoku.

2		5		1		9	
8			2		3		6
	3			6			7
		1				6	
5	4						1 9
		2				7	
	9			3			8
2			8		4		7
	1		9		7		6

Unsolved Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	5
9	7	1	3	8	5	6	2	4
5	4	3	7	2	6	8	1	9
6	8	2	1	4	9	7	5	3
7	9	4	6	3	2	5	8	1
2	6	5	8	1	4	9	3	7
3	1	8	9	5	7	4	6	2

Solved Sudoku

Gambar 4.2 Contoh tipikal teka-teki Sudoku dan solusinya

Teka-Teki Sudoku

Teka-teki Sudoku standar adalah mengisi papan 9×9 dengan angka 1 hingga 9, sedemikian rupa sehingga setiap baris, setiap kolom, dan setiap dari sembilan blok 3×3 berisi semua angka dari 1 hingga 9. Contoh tipikal teka-teki Sudoku dan solusinya ditunjukkan pada Gambar 4.2.

Untuk menyelesaikan masalah ini dengan menggunakan pemecah SAT, kita perlu menentukan teka-teki dalam bentuk CNF. Pertama, kita perlu mendefinisikan variabel proposisional. Misalkan $p_{i,j,k}$ adalah variabel proposisional sedemikian rupa sehingga $p_{i,j,k} = 1$ jika dan hanya jika angka k berada di baris i dan kolom j dari papan Sudoku. Karena $1 \leq i, j, k \leq 9$, maka ada $9^3 = 729$ variabel.

Kita dapat mengkodekan $p_{i,j,k}$ sebagai $81(i - 1) + 9(j - 1) + k$, maka $p_{1,1,1} = 1$ dan $p_{9,9,9} = 729$. Pengkodean lain adalah $p_{i,j,k} = 100i + 10j + k$, maka $p_{1,1,1} = 111$ dan $p_{9,9,9} = 999$. Yang terakhir membuang beberapa variabel, tetapi memungkinkan kita untuk melihat dengan jelas nilai i, j , dan k dari bilangan bulat. Kita akan menggunakan yang terakhir; karena ini adalah masalah SAT yang sangat mudah, sedikit pemborosan tidak masalah. Tiga baris pertama dari file CNF dapat dibaca sebagai

```
c Sudoku puzzle
p cnf 999 1000000
```

di mana 1000000 adalah perkiraan untuk jumlah klausa dari batasan pada teka-teki.

Batasan umum teka-teki dapat dinyatakan sebagai berikut:

- Setiap sel berisi tepat satu digit dari nilai apa pun.
- Setiap baris berisi setiap angka tepat satu kali, yaitu, tidak ada salinan angka yang sama dalam satu baris.
- Setiap kolom berisi setiap angka tepat satu kali.
- Setiap kotak 3×3 berisi setiap angka tepat satu kali.

Solusi untuk papan tertentu akan menyatakan angka mana yang harus ditempatkan di sel mana.

Kendala pertama di atas dapat dibagi menjadi dua bagian: (1) Setiap sel berisi setidaknya satu angka, dan (2) setiap sel berisi paling banyak satu angka. Jika setiap sel dalam satu baris menyumbangkan satu angka dan setiap angka muncul paling banyak satu kali, maka itu berarti setiap angka muncul tepat satu kali di baris ini. Alasan yang sama berlaku untuk kolom dan kotak, sehingga tiga kendala berikutnya dapat disederhanakan dengan menghilangkan kendala seperti "setiap baris berisi setiap angka setidaknya satu kali."

1. Setiap sel berisi setidaknya satu angka.
2. Setiap sel berisi paling banyak satu angka.
3. Setiap baris berisi setiap angka paling banyak satu kali.
4. Setiap kolom berisi setiap angka paling banyak satu kali.
5. Setiap grid 3×3 memuat setiap angka paling banyak satu kali.

Sekarang mudah untuk mengubah batasan di atas menjadi klausa:

1. Untuk $1 \leq i, j \leq 9$, $(p_{i,j,1} \mid p_{i,j,2} \mid \dots \mid p_{i,j,9})$.
2. Untuk $1 \leq i, j \leq 9$, $1 \leq k < c \leq 9$, $(\overline{p_{i,j,k}} \mid \overline{p_{i,j,c}})$.
3. Untuk $1 \leq i, k \leq 9$, $1 \leq j < b \leq 9$, $(\overline{p_{i,j,k}} \mid \overline{p_{i,b,k}})$.
4. Untuk $1 \leq j, k \leq 9$, $1 \leq i < a \leq 9$, $(\overline{p_{i,j,k}} \mid \overline{p_{a,j,k}})$.
5. Untuk $0 \leq a, b \leq 2$, $1 \leq i < i' \leq 3$, $1 \leq j < j' \leq 3$, $1 \leq k \leq 9$,
 $(\overline{p_{3a+i,3b+j,k}} \mid \overline{p_{3a+i',3b+j',k}})$.

Terakhir, kita perlu menambahkan konfigurasi awal papan sebagai klausa. Jika angka c ditempatkan pada baris a dan kolom b , kita membuat klausa unit $(p_{a,b,c})$ untuknya. Setiap teka-teki Sudoku harus memiliki setidaknya 17 angka yang ditempatkan awalnya di papan.

Anda dapat menulis program kecil untuk mengkonversi klausa di atas ke format DIMACS dan menggunakan salah satu pemecah SAT yang tersedia di internet untuk memecahkan teka-teki Sudoku. Pemecah SAT tidak membutuhkan waktu lama untuk menemukan solusinya. Jika Anda ingin mendesain teka-teki Sudoku untuk teman Anda, Anda dapat menggunakan pemecah SAT untuk memeriksa apakah teka-teki Anda tidak memiliki solusi atau memiliki lebih dari satu solusi.

Masalah Persegi Latin

Definisi 4.3.2 Diberikan himpunan S yang terdiri dari n elemen, persegi Latin L di atas S adalah matriks $n \times n$ sedemikian sehingga setiap baris dan setiap kolom L merupakan permutasi dari S . Ukuran S , $|S|$, disebut orde L .

Teka-teki Sudoku adalah persegi Latin berorde 9 di atas $S = \{1, 2, \dots, 9\}$. Orang biasanya menggunakan $S = Z_n = \{0, 1, \dots, n-1\}$ untuk persegi Latin berorde n .

Setiap persegi Latin dapat dianggap sebagai definisi fungsi $*$: $S \times S \rightarrow S$ sedemikian sehingga $(a * b) = c$ jika dan hanya jika sel pada baris a dan kolom b berisi c . Pasangan $(S, *)$ adalah struktur khusus dalam aljabar abstrak, yang disebut *kuasigroup*.

Persegi Latin dengan sifat khusus telah banyak dipelajari dalam teori desain dan pemecah SAT telah digunakan untuk menyelesaikan banyak masalah terbuka di sana. Sebagai contoh, pemecah SAT telah berhasil digunakan untuk menunjukkan tidak adanya persegi Latin berordo $n = \{9, 10, 12, 13, 14, 15\}$ yang memenuhi kendala $((y * x) * y) * y = x$ untuk $x, y \in Z_n$. Masih menjadi pertanyaan terbuka apakah ada persegi Latin berordo 18 seperti itu.

Untuk merepresentasikan kendala $((y * x) * y) * y = x$ dalam bentuk CNF, kita perlu “meratakan” kendala tersebut dengan memperkenalkan variabel baru: Kendala $((y * x) * y) * y = x$ setara dengan

$$(y * x = z) \wedge (z * y = w) \rightarrow w * y = x$$

Jika kita mengganti $y * x = z$ dengan $p_{y,x,z}$, dan seterusnya, kita akan mendapatkan klausa berikut:

$$(\overline{p_{y,x,z}} \mid \overline{p_{z,y,w}} \mid p_{w,y,x})$$

untuk $x, y, z, w \in Z_n$. Karena persegi Latin dapat ditentukan dalam CNF seperti yang kita lakukan untuk teka-teki Sudoku, sekarang mencari persegi Latin berordo n yang memenuhi $((y * x) * y) * y = x$ sama dengan mencari model klausa berikut:

1. Untuk $0 \leq i, j < n, (p_{i,j,0} \mid p_{i,j,2} \mid \dots \mid p_{i,j,n-1})$.
2. Untuk $0 \leq i, j < n, 0 \leq k < c < n, (\overline{p_{i,j,k}} \mid \overline{p_{i,j,c}})$.
3. Untuk $0 \leq i, k < n, 0 \leq j < b < n, (\overline{p_{i,j,k}} \mid \overline{p_{i,b,k}})$.
4. Untuk $0 \leq j, k < n, 0 \leq i < a < n, (\overline{p_{i,j,k}} \mid \overline{p_{a,j,k}})$.
5. Untuk $0 \leq x, y, z, w < n, (\overline{p_{y,x,z}} \mid \overline{p_{z,y,w}} \mid p_{w,y,x})$.

Empat set klausa pertama disalin dari teka-teki Sudoku, dan set terakhir adalah spesifikasi dari $((y * x) * y) * y = x$ dalam logika proposisional.

Masalah Graf

Banyak masalah aplikasi dapat dimodelkan sebagai masalah graf, seperti merepresentasikan masalah penjadwalan ujian sebagai masalah pewarnaan graf. Dalam studi teori NP-completeness, kita belajar untuk mereduksi SAT ke beberapa masalah graf. Artinya, kita menggunakan algoritma untuk masalah graf untuk menyelesaikan SAT. Sekarang, karena kita memiliki pemecah SAT, reduksinya dibalik: Masalah graf direduksi menjadi SAT, yaitu, kita menggunakan pemecah SAT untuk menyelesaikan masalah graf. Tantangannya adalah memastikan bahwa reduksi menghasilkan CNF dengan ukuran minimal, atau setidaknya, ukuran CNF adalah polinomial dari ukuran input.

Definisi 4.3.3 (Pewarnaan Graf) Diberikan graf sederhana tak berarah $G = (V, E)$ dan bilangan bulat positif K , setiap bilangan $k, 1 \leq k \leq K$, mewakili sebuah warna. Masalah pewarnaan graf adalah untuk menetapkan warna k , untuk setiap $v \in V$ sedemikian sehingga kedua ujung setiap sisi E menerima warna yang berbeda.

Misalkan $V = \{v_1, v_2, \dots, v_n\}$. Kita memperkenalkan kn variabel proposisional $p_{i,k,1} \leq i \leq n, 1 \leq k \leq K$, dengan arti bahwa $p_{i,k} = 1$ jika dan hanya jika v_i menerima warna k . Batasan untuk pewarnaan graf diberikan di bawah ini:

1. Setiap simpul menerima tepat satu warna.
2. Kedua ujung setiap sisi menerima warna yang berbeda.

Batasan ini diterjemahkan ke dalam klausa-klausa berikut:

1. Untuk $1 \leq i \leq |V|, (p_{i,1} \mid p_{i,2} \mid \dots \mid p_{i,K})$.
2. Untuk $1 \leq i \leq |V|, 1 \leq j < k \leq K, (\overline{p_{i,j}} \mid \overline{p_{i,k}})$.
3. Untuk $(v_i, v_j) \in E, 1 \leq k \leq K, (\overline{p_{i,k}} \mid \overline{p_{j,k}})$.

Dua set klausa pertama menentukan “setidaknya satu” dan “paling banyak satu” untuk batasan “tepat satu,” seperti yang kita lakukan pada teka-teki Sudoku. Ukuran set klausa pertama adalah $K|V|$ (menghitung jumlah literal); ukuran set kedua adalah $(K - 1)K/2|V|$;

ukuran set ketiga adalah $2K|E|$. Secara tegas, ini bukan reduksi polinomial jika K bukan konstanta. Jika K besar, kita dapat mengasumsikan $K = s * t$ untuk beberapa bilangan bulat $s, t > 1$ dan memperkenalkan s variabel baru $q_{i,j,0} \leq j < s$, sedemikian sehingga $q_{i,j}$ benar jika simpul i menerima warna c untuk $j * t < c \leq (j + 1) * t$, dan menggunakan variabel-variabel ini untuk mengurangi ukuran himpunan klausa kedua.

Definisi 4.3.4 (Klik) Diberikan graf sederhana tak berarah $G = (V, E)$ dan bilangan bulat positif k , masalah klik adalah menemukan himpunan bagian $S \subseteq V$ sedemikian sehingga $|S| = k$ dan untuk setiap dua simpul x, y dari S , $(x, y) \in E$.

Misalkan $V = \{v_1, v_2, \dots, v_n\}$. Sebagai percobaan pertama, mari kita definisikan n variabel proposisional, $x_{i,1} \leq i \leq n$, dengan pemahaman bahwa $x_i = 1$ jika dan hanya jika $v_i \in S$. Mudah untuk menentukan jika $v_i, v_j \in S$, maka $(v_i, v_j) \in E$. Artinya, jika $(v_i, v_j) \notin E$, maka kita membuat klausa $(\bar{x}_i | \bar{x}_j)$. Namun, akan tidak efisien untuk menentukan bahwa $|S| = k$. Untuk memastikan bahwa $|S| = k$, kita dapat mempertimbangkan himpunan bagian $T \subset V$ dengan $n - k + 1$ simpul. Salah satunya harus berada di S . Untuk menentukan batasan ini dalam logika, kita memiliki klausa berikut:

$$\text{Untuk } T \subset V \text{ dan } |T| = n - k + 1, (\bigvee_{v_i \in T} x_i).$$

Ini adalah himpunan klausa yang sangat besar dan sangat panjang. Jelas reduksi dari klik ke CNF bukanlah polinomial. Misalnya, jika $n = 100$ dan $k = 5$, maka ada $3.921.225 (= 100!/(4! 96!))$ klausa dengan panjang 96. Jelas, pemecah SAT mana pun akan kesulitan untuk menanganinya.

Solusi alternatif adalah dengan menganggap bahwa S disimpan sebagai daftar L yang terdiri dari k simpul. Kita memperkenalkan tambahan $k * n$ variabel proposisional, $p_{i,j,1} \leq i \leq n, 1 \leq j \leq k$, dengan pemahaman bahwa $p_{i,j} = 1$ jika dan hanya jika v_i berada di posisi j dari L . Kemudian kita memiliki batasan berikut:

1. Terdapat tepat satu simpul di setiap posisi L .
2. Jika $v_i \in L$, maka $v_i \in S$.
3. Jika $v_i \in S$, maka $v_i \in L$ (opsional).

Batasan-batasan di atas, ditambah batasan lama, dapat ditentukan dalam CNF sebagai berikut:

- (a_1) Untuk $1 \leq j \leq k, (p_{1,j} | p_{2,j} | \dots | p_{n,j})$.
- (a_2) Untuk $1 \leq j \leq k, 1 \leq i < i' \leq n, (\overline{p_{i,j}} | \overline{p_{i',j}})$.
- (a_3) Untuk $1 \leq j \leq k, 1 \leq i \leq n, (\overline{p_{i,j}} | x_i)$.
- (a_4) Untuk $1 \leq i \leq n, (\bar{x}_i | p_{i,1} | p_{i,2} | \dots | p_{i,k})$.
- (a_5) Untuk $1 \leq i, j \leq n, \text{jika } (v_i, v_j) \notin E, (\bar{x}_i | \bar{x}_j)$.

Tanpa mengurangi keumuman, kita dapat mengasumsikan bahwa L adalah daftar yang diurutkan dengan menambahkan lebih banyak klausa, untuk mengurangi beberapa solusi simetris.

Jika $n = 100, k = 5, (a1)$ menghasilkan 5 klausa dengan panjang 100; $(a2)$ menghasilkan 24.750 $(= 5 * 100 * 99/2)$ klausa dengan panjang 2; $(a3)$ menghasilkan 500 klausa dengan panjang 2; dan $(a4)$ menghasilkan 100 klausa dengan panjang 6, dengan total 25.355 klausa. Ini merupakan pengurangan yang substansial dari 3.921.225 dengan biaya memperkenalkan 500 variabel baru.

4.4 KEPUASAN MAKSIMUM

Diberikan himpunan C klausa dan interpretasi lengkap σ , kita dapat menghitung berapa banyak klausa yang salah di bawah σ . Misalkan

$$f(C, \sigma) = |\{c : c \in C, \sigma(c) = 0\}|,$$

Maka $0 \leq f(C, \sigma) \leq |C|$, di mana $|C|$ adalah jumlah klausa dalam C . Ketika $f(C, \sigma) = 0$, σ adalah model dari C . Masalah kepuasan maksimum MAX-SAT dengan demikian merupakan masalah optimasi di mana f berfungsi sebagai fungsi tujuan: Temukan σ sedemikian sehingga $f(C, \sigma)$ minimal (atau $|C| - f(C, \sigma)$ maksimal).

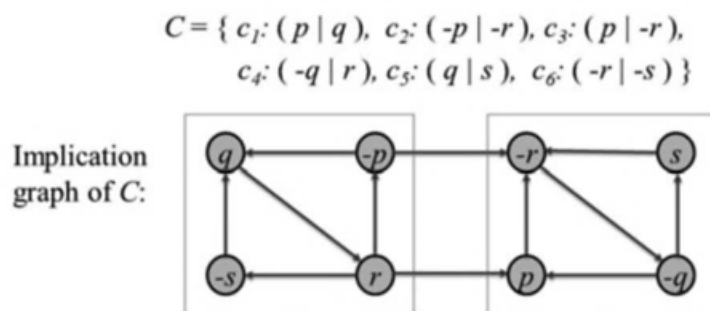
2SAT Versus Max2SAT

Misalnya, Max-kSAT menunjukkan versi keputusan dari MaxSAT yang mengambil bilangan bulat positif m dan himpunan C klausa sebagai masukan, di mana setiap klausa memiliki tidak lebih dari k literal, dan mengembalikan nilai benar jika ada interpretasi σ sedemikian sehingga setidaknya m klausa terpenuhi di bawah σ . Ketika $m = |C|$, maka Max-kSAT menjadi masalah kepuasan kSAT. Sudah diketahui bahwa 2SAT dapat diselesaikan dalam waktu polinomial, sedangkan Max2SAT adalah NP-lengkap. Dengan demikian, MaxSAT jauh lebih sulit daripada SAT.

Teorema 4.4.1 Terdapat prosedur pengambilan keputusan waktu linear untuk 2SAT.

Bukti Misalkan C berada dalam 2CNF tanpa literal murni. Kita membangun graf berarah $G = (V, E)$ untuk C sebagai berikut: V berisi dua literal untuk setiap variabel yang muncul di C . Untuk setiap klausa $(A | B) \in C$, kita menambahkan dua sisi, $(\bar{A}, B)$ dan $(\bar{B}, A)$, ke dalam E . Jelas, sisi-sisi tersebut mewakili relasi implikasi di antara literal, karena $(A | B) \equiv (\bar{A} \rightarrow B) \wedge (\bar{B} \rightarrow A)$. Karena relasi implikasi bersifat transitif dan $(A \rightarrow B) \equiv (\bar{B} \rightarrow \bar{A})$, maka tiga pernyataan berikut ini setara untuk setiap pasangan literal x dan y dalam G :

1. $C \models (x \rightarrow y)$.
2. Terdapat jalur dari x ke y dalam G .
3. Terdapat jalur dari $\bar{y}$ ke $\bar{x}$ dalam G .



Gambar 4.3 Grafik implikasi untuk 2CNF C dan SCC

Kemudian kita menjalankan algoritma untuk komponen yang terhubung kuat (SCC), yang dapat dilakukan dalam waktu linear, pada G . Mudah untuk melihat bahwa $C \models (x \leftrightarrow y)$ untuk setiap dua literal x dan y dalam komponen yang sama, karena x dan y berbagi siklus di G . Jika suatu variabel, misalnya p , dan komplementnya, $\neg p$, muncul dalam komponen yang sama, maka C tidak dapat dipenuhi, karena $C \models (p \leftrightarrow \neg p)$. Jika tidak ada kasus seperti itu, maka C harus dapat dipenuhi. Sebuah model dapat diperoleh dengan memberikan nilai kebenaran yang sama kepada setiap node dalam komponen yang sama, mengikuti urutan topologi komponen yang diinduksi oleh G : Selalu berikan nilai 0 kepada suatu komponen, kecuali jika salah satu literal dalam komponen tersebut telah menerima nilai 1.

Contoh 4.4.2 Diberikan $C = \{c_1: (p \mid q), c_2: (\bar{p} \mid \bar{r}), c_3: (p \mid \bar{r}), c_4: (\bar{q} \mid r), c_5: (q \mid s), c_6: (\bar{r} \mid \bar{s})\}$, grafik implikasinya ditunjukkan pada Gambar 4.3. Terdapat dua SCCs: $S_1 = \{\bar{p}, q, r, \bar{s}\}$ dan $S_2 = \{p, \bar{q}, \bar{r}, s\}$. Kita harus memberikan nilai 0 kepada setiap node di S_1 (dan 1 kepada setiap node di S_2) untuk mendapatkan model C .

Teorema 4.4.3 Max2SAT adalah NP-lengkap.

Bukti NP adalah kelas masalah keputusan yang solusinya dapat diperiksa dalam waktu polinomial. Max2SAT berada di NP karena kandidat solusi adalah interpretasi σ dan kita dapat memeriksa berapa banyak klausa yang benar di bawah σ dalam waktu polinomial.

Untuk menunjukkan bahwa Max2SAT adalah NP-hard, kita mereduksi 3SAT ke Max2SAT. Artinya, jika ada algoritma A untuk menyelesaikan Max2SAT, kita dapat membangun algoritma B , yang menggunakan A , untuk menyelesaikan 3SAT, dan waktu komputasi A dan B berbeda dengan polinomial dari ukuran input.

Diberikan instance C dari 3SAT, kita mengasumsikan bahwa setiap klausa C memiliki tepat tiga literal. Untuk setiap klausa $c_i = (l_1 \mid l_2 \mid l_3) \in C$, di mana l_1, l_2 , dan l_3 adalah literal sembarang, kita membuat 10 klausa di Max2SAT:

$$(l_1), (l_2), (l_3), (x_i), (\bar{l}_1 \mid \bar{l}_2), (\bar{l}_1 \mid \bar{l}_3), (\bar{l}_2 \mid \bar{l}_3), (l_1 \mid \bar{x}_i), (l_2 \mid \bar{x}_i), (l_3 \mid \bar{x}_i)$$

di mana x_i adalah variabel baru. Misalkan C' adalah kumpulan klausa-klausa ini yang dibuat dari klausa-klausa dalam C , maka $|C'| = 10|C|$ dan pernyataan-pernyataan berikut ini benar:

- Jika suatu interpretasi memenuhi c_i , maka kita dapat memenuhi tujuh dari sepuluh klausa.
- Jika suatu interpretasi memalsukan c_i , maka paling banyak enam dari sepuluh klausa dapat dipenuhi.

Pernyataan-pernyataan di atas dapat diverifikasi dengan tabel kebenaran pada l_1, l_2, l_3 , dan x_i :

l_1	l_2	l_3	x_i	$(\overline{l_1} \overline{l_2})$	$(\overline{l_1} \overline{l_3})$	$(\overline{l_2} \overline{l_3})$	$(l_1 \overline{x_i})$	$(l_2 \overline{x_i})$	$(l_3 \overline{x_i})$	sum
0	0	0	0	1	1	1	1	1	1	6
0	0	0	1	1	1	1	0	0	0	4
0	0	1	0	1	1	1	1	1	1	7
0	0	1	1	1	1	1	0	0	1	6
0	1	1	0	1	1	0	1	1	1	7
0	1	1	1	1	1	0	0	1	1	7
1	1	1	0	0	0	0	1	1	1	6
1	1	1	1	0	0	0	1	1	1	7

Empat kolom pertama berfungsi untuk nilai kebenaran $\{l_1, l_2, l_3, x_i\}$ dan empat klausa pertama dari sepuluh klausa. Dua baris pertama tabel menunjukkan bahwa ketika $(l_1 | l_2 | l_3)$ salah, jumlah nilai kebenaran dari sepuluh klausa (kolom terakhir) paling banyak adalah 6. Sisa tabel menunjukkan jumlah nilai kebenaran dari sepuluh klausa ketika satu, atau dua, atau tiga literal dari $\{l_1, l_2, l_3\}$ benar (beberapa kasus diabaikan karena simetri l_1, l_2 , dan l_3). Jelas dari tabel bahwa ketika $(l_1 | l_2 | l_3)$ benar, kita dapat memilih nilai x_i untuk mendapatkan tujuh klausa yang benar dari sepuluh klausa. Maka C dapat dipenuhi jika dan hanya jika C' memiliki interpretasi yang memenuhi $K = 7|C|$ klausa dari C' . Jika kita memiliki algoritma untuk menyelesaikan C' dengan $K = 7|C|$, maka output algoritma akan memberi tahu apakah C dapat dipenuhi atau tidak.

MaxSAT Berbobot dan Hibrida

Definisi 4.4.4 Diberikan himpunan C klausa dan fungsi $w: C \rightarrow N, w(c)$ disebut bobot (atau biaya) dari $c \in C$. (C, w) disebut CNF berbobot (WCNF). Diberikan (C, w) , masalah MaxSAT berbobot adalah menemukan interpretasi lengkap σ sedemikian sehingga $\sum_{c \in C, \sigma(c)=0} w(c)$ minimal (atau setara, $\sum_{c \in C, \sigma(c)=0} w(c)$ maksimal). Setiap klausa dalam C disebut klausa berbobot dan dilambangkan dengan $(c; w(c))$.

Contoh 4.4.5 Misalkan C adalah sebuah instance MaxSAT berbobot, di mana $C = \{c_1 : (x_1; 5), c_2 : (\overline{x_1} | x_2; 4), c_3 : (x_1 | \overline{x_2} | x_3; 3), c_4 : (\overline{x_1} | \overline{x_2}; 2), c_5 : (x_1 | x_2 | \overline{x_3}; 4), c_6 : (\overline{x_1} | x_3; 1), c_7 : (\overline{x_1} | \overline{x_2} | \overline{x_3}; 2)\}$.

Maka $\sigma = \{x_1, x_2, \bar{x}_3\}$ adalah interpretasi optimal untuk C , di mana c_4 dan c_6 bernilai salah, dengan total bobot klausa yang salah adalah 3; Jumlah bobot klausa yang benar adalah 18.

Definisi 4.4.6 Diberikan WCNF (C, w) , misalkan (H, S) adalah partisi dari himpunan C , di mana setiap klausa $c \in H$ memiliki bobot $w(c) = \infty$, solusi dari (H, S) adalah interpretasi penuh σ sedemikian sehingga setiap klausa H benar dalam σ . Masalah MaxSAT hibrida adalah menemukan solusi optimal sedemikian sehingga $\sum_{c \in S, \sigma(c)=0} w(c)$ minimal. Klausa dalam H disebut klausa keras dan klausa dalam S disebut klausa lunak.

MaxSAT hibrida juga disebut MaxSAT parsial berbobot. Definisi MaxSAT hibrida mengasumsikan bahwa MaxSAT hibrida adalah kasus khusus dari MaxSAT berbobot. MaxSAT berbobot juga dapat dilihat sebagai kasus khusus dari MaxSAT hibrida ketika $H = \emptyset$. Dengan kata lain, kedua definisi tersebut setara dalam daya ekspresif. Dalam praktiknya, ∞ dapat digantikan oleh jumlah semua bobot dalam S . Memisahkan H dari S akan memungkinkan kita untuk menangani H secara efisien, menggunakan teknik canggih untuk SAT.

Karena MaxSAT adalah kasus khusus dari MaxSAT berbobot atau MaxSAT hibrida, MaxSAT berbobot atau MaxSAT hibrida juga harus NP-lengkap. Bahkan, sangat mudah untuk mereduksi masalah NP-lengkap menjadi MaxSAT berbobot atau MaxSAT hibrida.

Masalah klik adalah masalah NP-lengkap yang terkenal dalam teori graf. Diberikan graf $G = (V, E)$, misalkan $V = \{v_1, v_2, \dots, v_n\}$. Kita mendefinisikan n variabel proposisional x_i , $1 \leq i \leq n$, dengan arti bahwa $x_i = 1$ jika dan hanya jika v_i berada dalam solusi dari clique maksimum. Kemudian kita dapat mereduksi masalah clique ke masalah MaxSAT berbobot dengan mengubah $G = (V, E)$ menjadi rumus 2WCNF, yang terdiri dari dua himpunan klausa berbobot berikut.

- Untuk $1 \leq i \leq n$, buat $(x_i; 1)$.
- Untuk $v_i, v_j \in V, i \neq j$ dan $(v_i, v_j) \in E$, buat $(\bar{x}_i | \bar{x}_j; n)$.

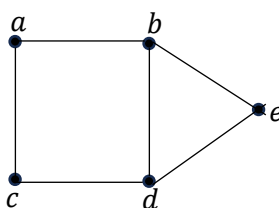
Himpunan klausa pertama menyatakan bahwa jika sebuah simpul dipilih dalam solusi, ada imbalan 1 (atau, jika simpul tersebut tidak ada dalam solusi, ada penalti 1). Rangkaian klausa kedua menentukan sifat dari sebuah klik: Jika dua simpul dipilih dalam solusi dan tidak ada sisi di antara keduanya, maka ada penalti sebesar $|V|$.

Contoh 4.4.7 Untuk grafik pada Gambar 4.4, klausa-klausanya adalah

$$C = (\bar{x}_a | \bar{x}_e; 5), (\bar{x}_b | \bar{x}_c; 5), (\bar{x}_c | \bar{x}_e; 5), (x_a; 1), (x_b; 1), (x_c; 1), (x_d; 1), (x_e; 1)\}.$$

Salah satu solusi optimal adalah $\sigma = \{x_a \rightarrow 1, x_b \rightarrow 1, x_c \rightarrow 0, x_d \rightarrow 1, x_e \rightarrow 0\}$, di mana dua klausa, yaitu (x_c) dan (x_e) , bernilai salah dan total penaltinya adalah 2.

Pertimbangkan interpretasi σ di mana satu variabel x_i bernilai benar; semua variabel lainnya bernilai salah. Dengan σ , semua klausa dalam himpunan kedua akan bernilai benar dan semua klausa dalam himpunan pertama akan bernilai salah, kecuali satu. Artinya, jumlah bobot klausa yang bernilai salah adalah $n - 1$ di bawah σ . Dengan demikian, total bobot semua klausa yang bernilai salah dalam solusi optimal apa pun tidak mungkin lebih besar dari $n - 1$. Dengan kata lain, semua klausa dalam himpunan kedua harus bernilai benar dalam solusi optimal. Ini menyiratkan bahwa harus ada sisi antara dua simpul dalam solusi tersebut. Solusi optimal memilih jumlah x_i maksimal dan dengan demikian memberi kita clique maksimal. Wajar untuk menentukan himpunan klausa kedua sebagai klausa keras dalam format MaxSAT hibrida.



Gambar 4.4 Menemukan klik maksimal dalam sebuah grafik

Metode Pencarian Lokal

Metode pencarian lokal banyak digunakan untuk menyelesaikan masalah optimasi. Dalam metode ini, pertama-tama kita mendefinisikan ruang pencarian, dan untuk setiap titik dalam ruang pencarian, kita mendefinisikan tetangga dari titik tersebut. Dimulai dengan titik mana pun (seringkali dipilih secara acak), kita berulang kali mengganti titik saat ini dengan salah satu tetangganya, biasanya yang lebih baik di bawah fungsi tujuan, sampai titik tersebut sesuai dengan kriteria pencarian, atau kita kehabisan waktu.

Untuk SAT atau MAX-SAT, ruang pencarian dari metode pencarian lokal umum adalah himpunan interpretasi lengkap. Jika C berisi n variabel, maka ukuran ruang pencarian adalah $O(2^n)$. Sebagai perbandingan, untuk prosedur pencarian menyeluruh seperti DPLL, ukuran ruang pencarian adalah $O(3^n)$, karena semua interpretasi parsial dipertimbangkan. Untuk kenyamanan, kita mengasumsikan bahwa interpretasi lengkap σ diwakili oleh himpunan n literal (atau setara, produk lengkap), di mana setiap variabel muncul tepat sekali. Tetangga dari σ diperoleh dengan membalik nilai kebenaran dari satu variabel dalam σ . Dengan demikian, setiap σ memiliki n tetangga:

$$neighbors(\sigma) = \{\sigma - \{A\} \cup \{\neg A\} : A \in \sigma\}$$

Selman dkk. mengusulkan prosedur pencarian lokal serakah yang disebut GSAT untuk SAT. Waktu eksekusi GSAT dikendalikan oleh dua parameter: MaxTries adalah jumlah maksimum titik awal acak, dan MaxFlips adalah jumlah maksimum pembalikan yang diizinkan untuk setiap titik awal.

Algoritma 4.4.8 GSAT(C) mengambil input berupa himpunan C klausa dan mengembalikan model C jika menemukannya; ia mengembalikan "tidak diketahui" jika berhenti tanpa menemukan model.

```

proc GSAT( $C, \sigma$ )
1  for  $i := 1$  to  $MaxTries$  do
2       $\sigma :=$  a random full interpretation of  $C$  // a starting point
3      for  $j := 1$  to  $MaxFlips$  do
4          if  $\sigma(C) = 1$  return  $\sigma$  // a model is found.
5          pick  $A \in \sigma$  // pick a literal of  $\sigma$  for flipping.
6           $\sigma := \sigma - \{A\} \cup \{\neg A\}$ 
7           $\sigma := \sigma$ 
8  return "unknown"

```

Contoh 4.4.9 Misalkan $C = \{c_1: (x_1 | x_3 | x_4), c_2: (\bar{x}_1 | \bar{x}_4), c_3: (\bar{x}_2 | \bar{x}_4), c_4: (\bar{x}_1 | x_2 | \bar{x}_3), c_5: (x_2 | \bar{x}_4)\}$. Jika interpretasi awalnya adalah $\sigma_0 = \{x_1, x_2, x_3, x_4\}$, maka $f(C, \sigma_0) = 2$ karena $\sigma_0(c_2) = \sigma_0(c_3) = 0$. Jika x_1 dipilih untuk dibalik, kita peroleh $\sigma_1 = \{\bar{x}_1, x_2, x_3, x_4\}$ dan $f(C, \sigma_1) = 1$ karena $\sigma_1(c_3) = 0$. Jika x_2 dipilih selanjutnya untuk dibalik, maka $\sigma_2 = \{\bar{x}_1, \bar{x}_2, x_3, x_4\}$ dan $f(C, \sigma_2) = 1$ karena $\sigma_2(c_5) = 0$. Jika x_4 dipilih selanjutnya untuk dibalik, maka $\sigma_3 = \{\bar{x}_1, \bar{x}_2, x_3, \bar{x}_4\}$ dan $f(C, \sigma_3) = 0$, yaitu, σ_3 adalah model dari C .

Terdapat banyak variasi GSAT, karena ada banyak cara untuk memilih literal pada baris 5. Secara umum, kita perlu memilih literal $A \in neighbors(\sigma)$ sedemikian rupa sehingga interpretasi yang dihasilkan σ' , di sini $\sigma' = \sigma - \{A\} \cup \{\neg A\}$, memenuhi ($f(C, \sigma') \leq f(C, \sigma)$). Kita dapat secara acak memilih salah satu literal tersebut atau memilih yang terbaik sedemikian rupa sehingga $f(C, \sigma')$ minimal.

Ketika $f(C, \sigma') = f(C, \sigma)$, GSAT masih memilih untuk berpindah dari σ ke σ' , dan perpindahan tersebut disebut perpindahan "menyamping"; perpindahan tersebut tidak menambah atau mengurangi jumlah total klausa yang tidak terpenuhi. Untuk menghindari GSAT bolak-balik antara dua interpretasi yang berdekatan dengan nilai f yang sama, diperlukan gagasan pencarian tabu, sehingga interpretasi yang baru saja dikunjungi tidak diizinkan menjadi interpretasi berikutnya. Pencarian GSAT biasanya dimulai dengan penurunan serakah yang cepat menuju penugasan kebenaran yang lebih baik, diikuti oleh urutan panjang gerakan "menyamping". Eksperimen menunjukkan bahwa pada banyak rumus, GSAT menghabiskan sebagian besar waktunya pada gerakan menyamping.

Kita juga dapat mengizinkan gerakan yang meningkatkan nilai f (disebut gerakan ke atas) sesekali, terutama di awal eksekusi. Misalnya, menggunakan ide "simulasi annealing," gerakan ke atas dapat diizinkan dengan probabilitas tertentu, dan probabilitas ini menurun seiring bertambahnya jumlah putaran. Untuk mengimplementasikan ide ini, kita mengganti baris 7 dengan dua baris berikut:

```

7    $x := \text{random}(0, 1)$  //  $x$  is a random number in  $[0, 1)$ 
7'  if  $(x < p \vee (f(C, \sigma') \leq f(C, \sigma)))$   $\sigma := \sigma'$ 

```

Selman dkk. mengusulkan variasi GSAT, yang disebut walksat, yang memilih variabel terbaik untuk dibalik dalam klausa konflik yang dipilih secara acak. Strategi walksat untuk memilih literal yang akan dibalik adalah sebagai berikut:

1. Pilih klausa konflik secara acak, misalnya α .
2. Jika ada literal $B \in \alpha$ sedemikian sehingga pembalikan B tidak mengubah klausa yang saat ini terpenuhi menjadi tidak terpenuhi, kembalikan B sebagai literal yang dipilih.
3. Dengan probabilitas p , pilih $B \in \alpha$ secara acak dan kembalikan B .
4. Dengan probabilitas $1 - p$, pilih $B \in \alpha$ sedemikian sehingga pembalikan B mengubah jumlah klausa yang saat ini terpenuhi menjadi tidak terpenuhi paling sedikit. Jika tidak puas, kembalikan B .

Sejak awal tahun 1990-an, telah ada penelitian aktif tentang perancangan, pemahaman, dan peningkatan metode pencarian lokal untuk SAT. Dalam kerangka GSAT, berbagai ide diusulkan, dan beberapa di antaranya diberikan di bawah ini.

- Saat memilih literal, ambil literal yang paling lama dibalik untuk memecahkan kebuntuan.
- Bobot diberikan kepada setiap klausa, dengan menambah bobot klausa yang tidak puas sebanyak satu untuk setiap interpretasi. Salah satu literal yang muncul di lebih banyak klausa dengan bobot maksimum dipilih.
- Setiap variabel diberi bobot, yang ditingkatkan pada variabel yang sering dibalik. Variabel dengan bobot minimum dipilih.

Selman dkk. menunjukkan melalui eksperimen bahwa GSAT serta walksat secara substansial mengungguli pemecah SAT berbasis DPLL terbaik pada beberapa kelas rumus, termasuk rumus yang dihasilkan secara acak dan masalah pewarnaan graf.

GSAT mengembalikan model atau "tidak diketahui," karena tidak pernah dapat menyatakan bahwa "klausa input tidak dapat dipenuhi." Ini adalah fitur khas dari metode pencarian lokal karena tidak lengkap dalam arti bahwa mereka tidak memberikan jaminan bahwa pada akhirnya akan melaporkan penugasan yang memuaskan atau menyatakan bahwa rumus yang diberikan tidak dapat dipenuhi.

Ada juga upaya pendekatan hibrida yang mengeksplorasi kombinasi ide dari metode DPLL dan teknik pencarian lokal. Ada juga pekerjaan untuk menganalisis metode pencarian lokal secara formal, menghasilkan beberapa algoritma waktu $O(2^n)$ terbaik untuk SAT. Misalnya, waktu eksekusi yang diharapkan, dengan mengabaikan faktor polinomial, dari metode pencarian lokal sederhana dengan restart setelah setiap $3n$ "pembalikan" telah ditunjukkan sebagai $((k + 1)/k)^n$ untuk k instance CNF dari n variabel, di mana setiap klausa berisi paling banyak k literal. Ketika $k = 3$, hasilnya menghasilkan kompleksitas $O((4/3)^n)$ untuk 3SAT.

Algoritma GSAT dapat digunakan untuk menyelesaikan masalah MaxSAT, yang meminta untuk menemukan interpretasi untuk sekumpulan klausa sedemikian rupa sehingga jumlah

klausa maksimal bernilai benar. Karena GSAT tidak lengkap, jawaban yang ditemukan oleh GSAT tidak dijamin optimal. Ini adalah fitur khas dari metode pencarian lokal yang selalu menemukan solusi optimal lokal. GSAT juga dapat digunakan untuk menyelesaikan MaxSAT berbobot dan MaxSAT hibrida.

Algoritma Branch-and-Bound

Untuk mendapatkan solusi optimal dari MaxSAT atau MaxSAT hibrida, pendekatan tradisional adalah menggunakan algoritma branch-and-bound. Dalam algoritma ini, kita sering menganggap MaxSAT sebagai masalah minimisasi: Temukan solusi dengan biaya minimum (optimal). Bagian percabangan dari algoritma branch-and-bound sangat mirip dengan DPLL, melakukan pemisahan kasus pada literal yang dipilih.

Bagian pembatasan dari algoritma memeriksa apakah estimasi solusi saat ini lebih baik atau tidak daripada solusi yang ditemukan sejauh ini. Jika tidak lebih baik, algoritma memotong cabang yang tidak berguna ini dan melakukan backtracking. Jika kita mencari solusi minimal, estimasi menghasilkan batas bawah; untuk solusi maksimal, estimasi menghasilkan batas atas. Misalnya, jika kita mencari solusi minimal, dan sudah memiliki solusi dengan nilai 30, jika estimasi mengatakan batas bawahnya adalah 32, maka tidak ada gunanya untuk melanjutkan, karena nilai solusi dari cabang saat ini tidak mungkin lebih rendah dari 32.

Di sini kami menyajikan algoritma branch-and-bound untuk MaxSAT hibrida dalam gaya DPLL rekursif (Algoritma 4.1.1).

Algoritma 4.4.10 $HyMaxSATBB(H, F, \sigma, soln)$ menerima input berupa himpunan H klausa keras, himpunan F klausa lunak berbobot, interpretasi parsial σ , dan solusi terbaik $soln$ yang ditemukan sejauh ini. H dan σ tidak memiliki variabel yang sama. Algoritma ini mencari interpretasi lengkap σ yang merupakan model dari H dan jumlah bobot klausa palsu dalam F minimal di bawah σ ; jumlah bobot ini akan dikembalikan oleh algoritma. Prosedur $pickLiteral(S)$ akan memilih literal yang muncul di S .

```

proc  $HyMaxSATBB(H, F, \sigma, soln)$ 
1    $res := BCP(H)$ 
2   if ( $res = \perp$ ) return  $\infty$  // hard causes violated
3   // Assume  $res = (U, S)$ 
4    $\sigma := \sigma \cup U$ 
5    $F := simplifySoft(F, \sigma)$ 
6   if  $soln \leq lowerBound(F)$  return  $soln$ 
7    $A := pickLiteral(S \cup F)$ 
8   if ( $A = nil$ ) return  $countFalseClauseWeight(F, \sigma)$ 
9    $soln := min(soln, HyMaxSATBB(S \cup \{A\}, F, \sigma, soln))$ 
10  return  $min(soln, HyMaxSATBB(S \cup \{A\}, F, \sigma, soln))$ 

```

Kita telah melihat BCP dalam Algoritma 4.2.5; empat prosedur berikut dipanggil di dalam $HyMaxSATBB$:

- *simplifySoft*(F, σ) pada baris 5: Sederhanakan klausa lunak F menggunakan interpretasi parsial σ dan pertahankan total bobot klausa salah tidak berubah di bawah interpretasi apa pun. Detailnya akan menyusul.
- *lowerBound*(F) pada baris 6: Perkirakan total bobot minimal klausa salah ketika σ diperluas ke interpretasi penuh dalam semua kemungkinan cara. Detailnya akan dibahas nanti.
- *pickLiteral*($S \cup F$) pada baris 7: Pilih literal yang belum ditetapkan untuk pemisahan kasus; ia mengembalikan nil jika setiap literal memiliki nilai kebenaran. Di sinilah heuristik untuk percabangan diimplementasikan.
- *countFalseClauseWeight*(F, σ) pada baris 8: Hitung jumlah semua bobot klausa salah dalam $F\sigma$. Di sini, σ adalah interpretasi lengkap dan dapat disimpan untuk penggunaan selanjutnya.

Untuk implementasi *HyMaxSATBB* yang efisien, solusi pertama selalu diperoleh dengan algoritma pencarian lokal seperti *GSAT*. Kita dapat menggunakan semua teknik yang digunakan untuk *DPLL*, termasuk prosedur iteratif, struktur data destruktif, dan pembelajaran klausa berbasis konflik untuk klausa keras.

Untuk klausa lunak, perlakuannya berbeda. Misalnya, resolusi unit adalah aturan penyederhanaan yang ampuh untuk klausa keras, tetapi tidak tepat untuk klausa lunak.

Jika $C = \{(\bar{x}_1; 1), (x_1 | x_2; 1), (x_1 | \bar{x}_2; 1), (x_1 | x_3; 1), (x_1 | \bar{x}_3; 1)\}$, solusi optimal memiliki satu klausa salah ketika $x_1 \rightarrow 1$. Jika kita menerapkan resolusi unit pada C untuk mendapatkan $C' = \{(\bar{x}_1; 1), (x_2; 1), (\bar{x}_2; 1), (x_3; 1), (\bar{x}_3; 1)\}$, solusi optimal dari C' akan mengandung dua klausa salah. Tentu saja, jika suatu klausa unit bersifat keras, kita dapat menerapkan resolusi unit pada klausa keras dan lunak.

Aturan Penyederhanaan dan Batas Bawah

Definisi 4.4.11 Suatu aturan inferensi atau aturan penyederhanaan yang berlaku untuk himpunan C klausa berbobot dikatakan sah, jika solusi optimal dari himpunan C' klausa berbobot yang dihasilkan memiliki nilai yang sama dengan solusi optimal dari C .

Dengan kata lain, aturan sah harus mempertahankan nilai solusi optimal. Sebagian besar aturan inferensi logika proposisional tidak sah untuk klausa berbobot karena menghasilkan klausa tambahan. Satu pengecualian adalah ketika klausa keras dapat dihasilkan dari klausa lunak. Misalnya, diberikan klausa lunak ($c; w$), jika c menjadi salah, w akan ditambahkan sebagai bagian dari batas bawah. Jika penambahan w menyebabkan batas bawah melebihi nilai solusi saat ini, maka kita dapat menghasilkan klausa keras c untuk mencegah c menjadi salah. Berikut ini, kita fokus pada aturan penyederhanaan, yang mengganti beberapa klausa dengan klausa lain ketika kita mengubah C menjadi C' . Setiap aturan penyederhanaan dapat dilambangkan dengan

$$C_1, C_2, \dots, C_m \Rightarrow C'_1, C'_2, \dots, C'_n$$

Ketika kita ingin mengganti $\{C_1, C_2, \dots, C_m\} \subseteq C$ dengan $\{C'_1, C'_2, \dots, C'_n\}$, di mana C_i, C'_j adalah klausa berbobot.

Dalam aturan berikut, α dan β menunjukkan disjungsi literal yang sembarang, mungkin kosong.

1. **Eliminasi nol:** Klausa berbobot dengan bobot nol dapat dibuang.

$$(\alpha; 0) \Rightarrow T$$

2. **Penghapusan tautologi:** Klausa yang valid selalu benar dan dapat diabaikan.

$$(p \mid \bar{p} \mid \alpha; w) \Rightarrow T$$

3. **Penggabungan identik:** Klausa identik digabungkan menjadi satu.

$$(\alpha; w_1), (\alpha; w_2) \Rightarrow (\alpha; w_1 + w_2)$$

4. **Bentrokan unit:** Klausa unit dengan literal komplemen dapat direduksi menjadi satu.

$$(A; w_1), (\bar{A}; w_2) \Rightarrow (A; w_1 - w_2), (\perp; w_2)$$

di mana A adalah literal dan $w_1 \geq w_2$. Misalnya, diberikan $(p; 2)$ dan $(\bar{p}; 3)$, jika $p \rightarrow 1$, kita mendapatkan biaya 3 dari $(\bar{p}; 3)$; jika $p \rightarrow 0$, kita mendapatkan biaya 2 dari $(p; 2)$. Hasil yang sama dapat diperoleh dari $(\bar{p}; 1)$ dan $(\perp; 2)$; yang terakhir ditambahkan ke dalam fungsi biaya.

5. **Resolusi tertimbang:**

$$(p \mid \alpha; w_1), (\bar{p} \mid \beta; w_2) \Rightarrow \begin{aligned} &(\alpha \mid \beta; u), (p \mid \alpha, w_1 - u), (\bar{p} \mid \beta; w_2 - u), \\ &(p \mid \alpha \mid \bar{\beta}; u), (\bar{p} \mid \bar{\alpha} \mid \beta; u) \end{aligned}$$

di mana $w_1 > 0, w_2 > 0$, dan $u = \min(w_1, w_2)$. Aturan ini diusulkan oleh Larrosa dkk. (2007). Karena $w_1 - u = 0$ atau $w_2 - u = 0$, aturan di atas tidak dapat digunakan selamanya. Misalnya, dari $(p \mid q; 2)$ dan $(\bar{p} \mid q; 3)$, $\alpha = \beta = q, u = \min(2, 3)$, kita memperoleh lima klausa: $(q; 2)$, $(p \mid q; 0)$, $(\bar{p} \mid q; 1)$, $(p \mid q \mid \bar{q}; 2)$, dan $(\bar{p} \mid \bar{q} \mid q; 2)$. Klausa kedua dihilangkan dengan eliminasi nol, dan dua klausa terakhir dengan eliminasi tautologi, kita memperoleh dua klausa: $(q; 2)$, $(\bar{p} \mid q; 1)$. Perhatikan bahwa $\bar{\beta}$ dan $\bar{\alpha}$ dalam dua klausa terakhir bukanlah disjungsi literal secara umum. Jika demikian, kita perlu mengubah $(p \mid \alpha \mid \bar{\beta}; u)$ (atau $(\bar{p} \mid \bar{\alpha} \mid \beta; u)$) menjadi sekumpulan klausa, masing-masing memiliki bobot u . Dalam praktiknya, aturan ini sering berlaku untuk klausa pendek atau ketika $\alpha = \beta$.

Sangat mudah untuk memeriksa apakah semua aturan penyederhanaan di atas valid untuk klausa berbobot, karena aturan tersebut mempertahankan solusi optimal.

Untuk mengimplementasikan *simplifySoft*(F, σ) secara efisien, F akan disederhanakan terlebih dahulu oleh σ , karena σ mewakili sekumpulan klausa unit keras. Seperti BCP, setiap klausa benar di bawah σ akan dihapus dari F dan setiap literal salah di bawah σ akan dihapus dari klausa di F . Setelah langkah ini, F dan σ tidak berbagi variabel apa pun. Langkah kedua adalah menerapkan aturan penyederhanaan di atas. Beberapa aturan seperti penghapusan nol dan penghapusan tautologi, diterapkan secara langsung. Beberapa aturan seperti resolusi berbobot diterapkan secara selektif.

Untuk mengimplementasikan *lowerBound*(F, soln), pertama-tama, kita mengasumsikan bahwa F telah disederhanakan dan kita menghitung total bobot dari semua klausa yang salah. Jika jumlahnya melebihi solusi saat ini, kita keluar.

Untuk klausa F yang bukan salah, terdapat berbagai teknik untuk memperkirakan batas bawah. Salah satu tekniknya adalah dengan merumuskan F sebagai contoh pemrograman linier bilangan bulat (ILP). Pemecah ILP adalah alat optimasi umum untuk riset operasi. Pemecah ILP memecahkan masalah dengan kendala linier dan fungsi tujuan linier di mana beberapa variabel adalah bilangan bulat.

Pemecah ILP canggih sangat ampuh dan efektif serta dapat langsung menyerang MaxSAT. Dalam praktiknya, pemecah ILP efektif pada banyak masalah optimasi standar, misalnya, penutup simpul, tetapi untuk masalah di mana terdapat banyak kendala Boolean, ILP tidak seefektif itu. Informasi lebih lanjut tentang ILP dapat ditemukan di Bab 12.

Untuk mendapatkan batas bawah ILP, kendala pada solusi bilangan bulat dilonggarkan sehingga ILP menjadi contoh pemrograman linier (LP), yang diketahui memiliki solusi waktu polinomial.

Jika kita memandang T sebagai 1 dan $\perp$ sebagai 0, sudah diketahui bahwa sebuah klausa dapat diubah menjadi pertidaksamaan linear, seperti $(\bar{x}_1 \mid x_2 \mid \bar{x}_3)$ menjadi $(1 - x_1) + x_2 + (1 - x_3) \geq 1$, sehingga klausa tersebut dapat dipenuhi jika dan hanya jika himpunan pertidaksamaan tersebut memiliki solusi 0/1 untuk semua variabel.

Untuk setiap klausa c , misalkan $P(c)$ dan $N(c)$ adalah himpunan literal positif dan literal negatif dalam c , masing-masing. Misalkan $F = \{(c_i; w_i) : 1 \leq i \leq m\}$ atas n variabel, katakanlah $\{x_1, x_2, \dots, x_n\}$. F dapat dirumuskan sebagai contoh ILP berikut:

minimalkan $\sum_{1 \leq i \leq m} w_i(1 - y_i)$ // minimalkan bobot klausa yang dipalsukan dengan syarat $\sum_{x \in P(c_i)} x + \sum_{\bar{x} \in N(c_i)} (1 - x) \geq y_i, 1 \leq i \leq m$, // c_i benar jika dan hanya jika $y_i = 1$
 $y_i \in \{0, 1\}, 1 \leq i \leq m$, // setiap klausa dipalsukan atau dipenuhi
 $x_j \in \{0, 1\}, 1 \leq j \leq n$. // setiap variabel bernilai false atau true di mana y_i adalah variabel baru untuk setiap klausa c_i dalam F .

Secara intuitif, $y_i = 1$ menyiratkan bahwa $\sum_{x \in P(c_i)} x + \sum_{\bar{x} \in N(c_i)} (1 - x) \geq 1$, yaitu, c_i bernilai true. Oleh karena itu, semakin banyak y_i dengan $y_i = 1$, semakin kecil nilai $\sum_{1 \leq i \leq m} w_i (1 - y_i)$

Contoh ILP di atas dapat dilonggarkan menjadi contoh L dari LP :

minimalkan $\sum_{1 \leq i \leq m} w_i (1 - y_i)$ dengan syarat
 $\sum_{x \in P(c_i)} x + \sum_{\bar{x} \in N(c_i)} (1 - x) \geq y_i, 1 \leq i \leq m,$
 $0 \leq y_i \leq 1, 1 \leq i \leq m,$ dan
 $0 \leq x_j \leq 1, 1 \leq j \leq n.$

Jika L memiliki solusi, maka $\sum_{1 \leq i \leq m} w_i (1 - y_i)$ adalah batas bawah untuk solusi dari $F.LP$ dapat digunakan untuk mendapatkan solusi perkiraan untuk F ketika kita membulatkan nilai x_i dari bilangan riil ke bilangan bulat.

Teknik lain untuk batas bawah mencari inkonsistensi yang memaksa beberapa klausa lunak untuk difalsifikasi. Beberapa orang menyarankan untuk mempelajari klausa lunak yang difalsifikasi; beberapa orang menggunakan konsep klon, minibucket, atau BDD dengan pembatasan lebar dalam relaksasi. Misalnya, kita dapat memperlakukan klausa lunak seolah-olah itu adalah klausa keras dan kemudian menjalankan BCP untuk menemukan klausa yang difalsifikasi. Ini dapat membantu kita menemukan subset klausa yang tidak dapat dipenuhi dengan cepat dan menggunakannya untuk memperkirakan total bobot klausa yang difalsifikasi. Teknik-teknik ini dapat efektif pada masalah kombinatorial kecil. Setelah jumlah variabel dalam F mencapai 1.000 atau lebih, teknik batas bawah ini menjadi lemah atau terlalu mahal. Strategi untuk mengaktifkan/menonaktifkan teknik-teknik ini di *HyMaxSATBB* masih menjadi topik penelitian.

Selain algoritma berbasis branch-and-bound, beberapa solver MaxSAT mengubah sebuah instance MaxSAT menjadi urutan instance SAT di mana setiap instance mengkodekan masalah keputusan dalam bentuk, untuk nilai k yang berbeda, “apakah ada interpretasi yang memalsukan klausa lunak dengan total bobot paling banyak k ?”

Misalnya, jika kita mulai dengan nilai k yang kecil, instance SAT akan tidak dapat dipenuhi. Dengan meningkatkan k secara bertahap, k pertama ketika instance SAT menjadi dapat dipenuhi akan menjadi nilai solusi optimal. Tentu saja, strategi linier pada k ini belum tentu memberikan solver MaxSAT yang efektif.

Ada banyak penelitian yang sedang berlangsung ke arah ini. Fokus dari pendekatan ini terutama melakukan dua hal:

1. Mengembangkan cara ekonomis untuk mengkodekan masalah keputusan pada setiap tahap urutan.
2. Memanfaatkan informasi yang diperoleh dari solver SAT pada setiap tahap di tahap berikutnya.

Beberapa solver MaxSAT menggunakan teknik inovatif untuk mendapatkan cara yang lebih efisien dalam mengkodekan dan menyelesaikan masalah keputusan SAT individual. Beberapa

solver menggunakan core yang tidak dapat dipenuhi untuk meningkatkan efisiensi penyelesaian SAT. Solver ini tampak menjanjikan karena efektif pada beberapa masalah MaxSAT besar, terutama yang memiliki banyak klausa keras.

Penggunaan Solver MaxSAT Hibrida

Sebagian besar masalah dunia nyata melibatkan komponen optimasi dan ada permintaan tinggi untuk pendekatan otomatis dalam menemukan solusi yang baik untuk masalah optimasi yang sulit secara komputasi. Kita telah melihat bagaimana cara mengkodekan masalah clique dengan mudah dan ringkas ke dalam sebuah instance MaxSAT. MaxSAT memungkinkan pengkodean berbagai jenis kendala lunak domain terbatas tingkat tinggi secara ringkas.

Terdapat ekstensi format DIMACS untuk WCNF. Untuk menentukan instance MaxSAT hibrida, baris yang dimulai dengan p akan berisi kata kunci wcnf, diikuti oleh jumlah variabel, jumlah klausa, dan label untuk klausa keras. Artinya,

```
p wcnf <# jumlah variabel> <# jumlah klausa> <label klausa keras>
```

Label klausa keras adalah bilangan bulat yang lebih besar dari bobot maksimal klausa lunak. Setiap klausa dimulai dengan nilai bobot, diikuti oleh literal dan diakhiri dengan 0. Untuk contoh clique kecil pada Contoh 4.4.7, file input ke solver MaxSAT akan terlihat seperti:

```
c contoh clique kecil
c xa = 1, xb = 2, xc = 3, xd = 4, xe = 5
p wcnf 5 8 9
9 -1 -5 0
9 -2 -3 0
9 -3 -5 0
1 1 0
1 2 0
1 3 0
1 4 0
1 5 0
```

Suatu klausa dikatakan keras jika bobot klausa tersebut sama dengan label untuk klausa keras. Format standar untuk instance MaxSAT memungkinkan evaluasi tahunan solver MaxSAT. Situs web terbaru untuk evaluasi MaxSAT dapat ditemukan di internet dan situs untuk tahun 2020 diberikan di bawah ini:

<https://maxsat-evaluations.github.io/2020/>

Para peneliti MaxSAT dapat menilai solver MaxSAT terkini dan membuat koleksi contoh benchmark MaxSAT yang tersedia untuk umum.

Pemecah MaxSAT ini dibangun di atas teknik-teknik sukses untuk SAT, terus berkembang dalam teknologi pemecah masalah praktis, dan menawarkan alternatif untuk pendekatan tradisional, misalnya, pemrograman bilangan bulat. Pemecah MaxSAT menggunakan logika proposisional sebagai bahasa deklaratif yang mendasarinya dan sangat cocok untuk masalah optimasi yang secara inheren "sangat Boolean". Kinerja pemecah MaxSAT telah melampaui algoritma khusus pada beberapa domain masalah:

- Pengelompokan korelasi
- Inferensi probabilistik
- Konsistensi kuartet maksimum
- Manajemen paket perangkat lunak
- Lokalisasi kesalahan
- Penalaran atas bionetwork
- Array penutup optimal
- Komputasi lebar pohon
- Pembelajaran struktur jaringan Bayesian
- Penemuan kausal
- Bidang pemotongan untuk IP (pemrograman bilangan bulat)
- Dinamika argumentasi

Kemajuan pemecah SAT dan MaxSAT kontemporer telah mengubah cara kita berpikir tentang masalah NP-lengkap. Mereka telah menunjukkan bahwa, meskipun masalah-masalah ini masih sulit diatasi dalam kasus terburuk, banyak contoh dapat ditangani dengan sukses.

4.5 LATIHAN SOAL

1. Gambarlah empat pohon keputusan lengkap (pohon panggilan rekursif) dari DPLL pada himpunan klausa berikut:

$$\{(p \mid q), (p \mid \bar{r}), (\bar{p} \mid r), (\bar{p} \mid q \mid \bar{r}), (\bar{q} \mid \bar{r})\}$$

menggunakan dua urutan variabel, p, q, r dan r, q, p , masing-masing, untuk pemisahan (nilai mana pun dapat dipilih terlebih dahulu), dan dengan atau tanpa BCP, masing-masing.

2. Gambarlah pohon keputusan lengkap *DPLL* pada himpunan klausa berikut:

$$\{(p \mid q \mid s), (p \mid \bar{r} \mid \bar{s}), (q \mid \bar{r} \mid s), (\bar{p} \mid q \mid \bar{r}), (\bar{q} \mid \bar{r} \mid \bar{s})\}$$

gunakan urutan p, q, r, s untuk pemisahan (nilai 1 digunakan pertama untuk setiap variabel pemisahan) dan laporkan nilai kebenaran variabel pada berbagai level. Berapa banyak model yang ditemukan oleh *DPLL*? Apa jalur panduannya ketika setiap model ditemukan?

3. Jika kita memasukkan himpunan klausa pada masalah sebelumnya ke dalam pemecah SAT yang bekerja sebagai kotak hitam dan hanya dapat menghasilkan satu model pada

satu waktu, klausa apa yang harus ditambahkan ke dalam himpunan klausa agar pemecah SAT dapat menghasilkan model baru?

4. Perkirakan batas atas yang ketat dari ukuran grafik inferensi untuk resolusi, jika klausa masukan berisi n variabel.
5. Berikan pseudo-kode untuk mengimplementasikan *BCPw* berdasarkan *BCPht* sedemikian rupa sehingga ruang untuk $head(c)$ dan $tail(c)$ dibebaskan dengan mengasumsikan $head(c) = 0$ dan $tail(c) = |c| - 1$ untuk setiap klausa non-unit c , dan tidak ada ruang tambahan yang dapat digunakan untuk c . Dalam kode Anda, Anda dapat memanggil $swap(lits(c), i, j)$, yang menukar dua elemen pada indeks i dan j dalam array $lits(c)$.
6. Diberikan himpunan H dari $m + 1$ klausa Horn atas m variabel:

$$H = \{c_1 = (\bar{x}_1 | \bar{x}_2 | \dots | \bar{x}_{m-1} | \bar{x}_m), c_2 = (\bar{x}_1 | x_2), c_3 = (x_2 | x_3), \dots, c_m = (\bar{x}_{m-1} | x_m), c_{m+1} = (x_1)\}$$

Tunjukkan bahwa ukuran H adalah $O(m)$, dan algoritma *BCPw* pada masalah sebelumnya akan membutuhkan waktu $\Omega(m^2)$ pada H , jika semua klausa dalam $cls(A)$ harus diproses sebelum mempertimbangkan $cls(B)$, di mana $\neg A$ diberi nilai benar sebelum $\neg B$.

7. Kita menerapkan DPLL (tanpa CDCL) pada himpunan klausa S berikut: $S = \{1. (a | c | \bar{d} | f), 2. (a | d | \bar{e}), 3. (\bar{a} | c), 4. (b | \bar{d} | e), 5. (b | d | \bar{f}), 6. (\bar{b} | \bar{f}), 7. (c | \bar{e}), 8. (c | e), 9. (\bar{c} | \bar{d})\}$, dengan asumsi bahwa *pickLiteral* memilih literal dalam urutan berikut: $\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e}, \bar{f}$ (false dicoba terlebih dahulu untuk setiap variabel). Harap (a) gambar pohon keputusan DPLL, (b) berikan klausa unit yang diturunkan oleh BCP di setiap simpul pohon keputusan, dan (c) identifikasi literal kepala/ekor dari setiap klausa di akhir DPLL (dengan asumsi awalnya literal pertama dan terakhir adalah literal kepala/ekor).
8. Kami menerapkan GSAT pada himpunan S klausa dalam masalah sebelumnya. Asumsikan bahwa node awal (yaitu, interpretasi) adalah $\sigma_0 = \{\bar{a}, b, \bar{c}, d, \bar{e}, f\}$, dan strategi untuk memilih tetangga untuk menggantikan node saat ini adalah (a) yang terbaik di antara semua tetangga dan (b) tetangga pertama yang lebih baik daripada node saat ini (membalik variabel dalam urutan $a, b, \dots, f$). Pergerakan menyamping diperbolehkan jika tidak ada tetangga yang lebih baik daripada node saat ini. Jika tidak ada tetangga yang lebih baik daripada node saat ini, ditemukan optimum lokal dan Anda perlu memulai ulang GSAT dengan node $\sigma_0 = \{a, \bar{b}, c, \bar{d}, e, \bar{f}\}$.
9. Kita menerapkan DPLL dengan CDCL pada himpunan klausa C : $\{c_1: (x_1 | x_2 | x_5), c_2: (x_1 | x_3 | \bar{x}_4 | x_5), c_3: (\bar{x}_1 | x_2 | x_3 | x_4), c_4: (\bar{x}_3 | x_4 | x_5), c_5: (\bar{x}_2 | x_3 | x_5), c_6: (\bar{x}_3 | \bar{x}_4 | \bar{x}_5), c_7: (\bar{x}_3 | x_4 | \bar{x}_5)\}$. Jika *pickLiteral* memilih literal untuk pemisahan kasus dalam urutan berikut $\bar{x}_5, \bar{x}_4, \dots, \bar{x}_1$ ($x_i = 0$) dicoba terlebih dahulu), silakan jawab pertanyaan berikut:

- (a) Untuk setiap klausa yang bertentangan yang ditemukan dalam $DPLL(C)$, klausa baru apa yang akan dihasilkan oleh *conflictAnalysis*?
 - (b) Apa literal kepala dan ekor jika klausa baru yang dihasilkan di (a) bukan unit? Lalu berapa nilai *level* pada akhir panggilan *insertNewClause*?
 - (c) Gambarkan pohon keputusan $DPLL$ hingga ditemukan model atau $DPLL(C)$ mengembalikan *false*.
10. Masalah sarang merpati adalah menempatkan $n + 1$ merpati ke dalam n lubang sedemikian rupa sehingga setiap merpati harus berada di dalam lubang dan tidak ada lubang yang dapat menampung lebih dari satu merpati. Silakan tulis encoder dalam bahasa pemrograman apa pun yang membaca bilangan bulat n dan menghasilkan rumus CNF untuk masalah sarang merpati dengan $n + 1$ merpati dan n lubang dalam format DIMACS. Simpan output encoder Anda dalam file bernama *pigeonX.cnf*, di mana X adalah n , dan masukkan *pigeonX.cnf* ke pemecah SAT yang menerima CNF dalam format DIMACS. Serahkan baik encoder, hasil dari solver SAT untuk $n = 4, 6, 8, 10$, dan file CNF untuk $n = 4$ (yaitu, *pigeon4.cnf*), ditambah ringkasan output dari solver SAT tentang ukuran input, waktu komputasi, dan jumlah konflik. Apa hubungan antara ukuran input dan waktu komputasi?
 11. Masalah n -ratu adalah menempatkan n ratu pada papan $n \times n$ sedemikian sehingga tidak ada dua ratu yang berada di baris yang sama, kolom yang sama, atau diagonal yang sama. Silakan tulis program bernama encoder dalam bahasa pemrograman apa pun yang membaca bilangan bulat n dan menghasilkan rumus CNF untuk masalah n -ratu dalam format DIMACS. Simpan output dari encoder Anda dalam file bernama *queenX.cnf* dan masukkan *queenX.cnf* ke solver SAT yang menerima CNF dalam format DIMACS. Tulis program lain bernama decoder yang membaca output dari pemecah SAT dan menampilkan solusi n -queen di layar komputer Anda. Serahkan encoder dan decoder, hasil dari pemecah SAT untuk $n = 5, 10, 15$, file CNF untuk $n = 5$ (yaitu, *queen5.cnf*), dan output dari decoder Anda untuk $n = 10$.
 12. Anda diminta untuk menulis dua program kecil, satu bernama encoder dan yang lainnya bernama decoder. Encoder akan membaca teka-teki Sudoku dari sebuah file (beberapa contoh disediakan secara online) dan menghasilkan rumus CNF untuk teka-teki ini dalam format DIMACS. Simpan output dari encoder Anda dalam sebuah file bernama *sudokuN.cnf* dan masukkan *sudokuN.cnf* ke pemecah SAT yang menerima CNF dalam format DIMACS. Decoder akan membaca output dari pemecah SAT dan menampilkan solusi teka-teki Sudoku di layar komputer Anda. Serahkan encoder dan decoder, hasil dari pemecah SAT, dan hasil decoder Anda untuk teka-teki Sudoku yang disediakan.
 13. Sebuah persegi Latin di atas $N = \{0, 1, \dots, n - 1\}$ adalah persegi $n \times n$ di mana setiap baris dan setiap kolom persegi adalah permutasi dari angka-angka dari N . Setiap persegi Latin mendefinisikan fungsi biner $*$: $N, N \rightarrow N$ sedemikian sehingga $(x * y) = z$ jika dan hanya jika entri pada baris ke- x dan kolom ke- y adalah z . Anda diminta untuk menulis program kecil yang disebut encoder. Encoder akan membaca bilangan bulat n

dan menghasilkan rumus CNF untuk persegi Latin berukuran n yang memenuhi batasan $x * x = x$ dan $(x * y) * (y * x) = y$ (disebut hukum ketiga Stein) dalam format DIMACS. Simpan output dari encoder Anda dalam file bernama `latinN.cnf` dan masukkan `latinN.cnf` ke dalam solver SAT yang menerima CNF dalam format DIMACS. Serahkan encoder, dan ringkasan hasil solver SAT untuk $n = 4, 5, \dots, 9$.

14. Diberikan graf sederhana tak berarah $G = (V, E)$, himpunan independen dari G adalah himpunan bagian $X \subseteq V$ sedemikian sehingga untuk setiap dua simpul $X, (x, y) \in E$. Masalah himpunan independen adalah untuk menemukan himpunan independen maksimum dari G . (a) Tentukan masalah himpunan independen dalam MaxSAT hibrida; dan (b) tentukan versi keputusan dari masalah himpunan independen dalam SAT.
15. Diberikan graf sederhana tak berarah $G = (V, E)$, penutup simpul dari G adalah himpunan bagian $X \subseteq V$ sedemikian sehingga untuk setiap sisi $(x, y) \in E, \{x, y\} \cap X \neq \emptyset$. Masalah penutup simpul adalah menemukan penutup simpul minimal dari G . (a) Tentukan masalah penutup simpul dalam MaxSAT hibrida; dan (b) tentukan versi keputusan dari masalah penutup simpul dalam SAT.
16. Diberikan graf sederhana tak berarah $G = (V, E)$, masalah jalur terpanjang adalah menemukan jalur sederhana terpanjang (tanpa simpul berulang) di G . (a) Tentukan masalah jalur terpanjang dalam MaxSAT hibrida; dan (b) tentukan versi keputusan dari masalah jalur terpanjang dalam SAT.

BAB 5

LOGIKA ORDE PERTAMA

Logika proposisional memberikan awal yang baik untuk menggambarkan prinsip-prinsip umum penalaran logis, tetapi tidak mungkin untuk mengekspresikan sifat-sifat umum dari banyak himpunan penting, terutama ketika suatu himpunan bersifat tak terbatas. Misalnya, bagaimana cara menentukan pernyataan-pernyataan berikut: Setiap bilangan asli adalah genap atau ganjil. Suatu bilangan asli adalah genap jika dan hanya jika penerusnya adalah ganjil. Relasi $<$ bersifat transitif atas bilangan asli. Daya ekspresif logika proposisional yang lemah menjelaskan kesederhanaan matematisnya yang relatif, tetapi ini merupakan keterbatasan yang sangat serius, dan diinginkan untuk memiliki logika yang lebih ekspresif. Matematika dan beberapa disiplin ilmu lain seperti ilmu komputer sering mempertimbangkan himpunan elemen di mana relasi dan operasi tertentu dipilih.

Ketika menggunakan bahasa logika proposisional, kemampuan kita untuk berbicara tentang sifat-sifat relasi dan operasi tersebut sangat terbatas. Dengan demikian, perlu untuk menyempurnakan bahasa logika, untuk meningkatkan daya ekspresif logika. Inilah tepatnya yang disediakan oleh kerangka kerja logis yang dikenal sebagai logika orde pertama, yang akan menjadi topik tiga bab berikutnya. Logika orde pertama adalah logika yang jauh lebih kaya daripada logika proposisional, namun memiliki banyak sifat matematika yang bagus, yang banyak di antaranya diwarisi dari logika proposisional.

Dalam logika orde pertama, banyak sifat menarik dan penting tentang berbagai himpunan dapat diungkapkan. Secara teknis, ini dicapai dengan memungkinkan simbol proposisional memiliki argumen yang mencakup elemen-elemen himpunan, untuk mengungkapkan relasi dan operasi yang dimaksud. Simbol proposisional ini disebut simbol predikat. Inilah alasan mengapa logika orde pertama disebut kalkulus predikat.

5.1 SINTAKSIS BAHASA ORDE PERTAMA

Logika adalah kumpulan bahasa formal yang saling terkait erat. Ada bahasa-bahasa tertentu yang disebut bahasa orde pertama, dan bersama-sama membentuk logika orde pertama. Studi kita tentang logika orde pertama akan sejajar dengan studi tentang logika proposisional: Pertama-tama kita akan mendefinisikan sintaksis dan semantik bahasa orde pertama sebagai perluasan dari bahasa proposisional, dan kemudian menyajikan berbagai alat penalaran untuk bahasa-bahasa ini.

Istilah dan Rumus

Pertimbangkan beberapa pernyataan umum tentang bilangan asli:

- Setiap bilangan asli adalah genap atau ganjil, tetapi tidak keduanya.
- Suatu bilangan asli x adalah genap jika dan hanya jika $x + 1$ adalah ganjil.
- Untuk setiap bilangan asli x , $x < x + 1$ dan $\neg(x < x)$.
- Untuk setiap tiga bilangan asli x , y , dan z , jika $x < y$ dan $y < z$, maka $x < z$.
- Untuk setiap bilangan asli x , terdapat bilangan asli y , sedemikian sehingga $x < y$.

Sebelum mencoba membuktikan pernyataan-pernyataan ini, mari kita pertimbangkan bagaimana menuliskannya dalam bahasa formal. Pertama, kita membutuhkan cara untuk menggambarkan himpunan bilangan asli. Seperti logika proposisional, logika orde pertama tidak mengasumsikan keberadaan himpunan khusus apa pun, kecuali beberapa simbol, dan semuanya perlu dibangun dari awal. Untungnya, dari Contoh 1.3.8, kita tahu bahwa himpunan bilangan asli dapat dibangun secara rekursif dari dua simbol, yaitu konstanta 0 dan simbol fungsi uner s : Bilangan asli i secara unik diwakili oleh istilah $s_i(0)$, yang berarti ada i salinan s dalam $s(s(\dots s(0)\dots))$. Oleh karena itu, bahasa orde pertama kita akan memiliki 0 dan s .

Selanjutnya, untuk menyatakan sifat "genap" atau "ganjil", kita dapat memperkenalkan simbol predikat genap dan ganjil, yang merupakan contoh fungsi keputusan yang mengambil bilangan asli dan mengembalikan nilai Boolean. Relasi $<$ digunakan dalam pernyataan di atas dan kita menggunakannya dalam bahasa orde pertama. Karena beberapa variabel seperti x, y , dan z sudah digunakan dalam pernyataan tersebut, kita juga meminjamnya.

Terakhir, untuk menyatakan "untuk setiap" dan "ada" dalam bahasa alami, kita menambahkan dua simbol penting, $\forall$ dan $\exists$, masing-masing, dalam bahasa orde pertama. Simbol tersebut disebut kuantifikasi universal dan disebut kuantifikasi eksistensial. Kedua simbol ini akan dibahas secara luas di subbagian berikutnya.

Definisi 5.1.1 Bahasa orde pertama L dibangun oleh alfabet yang terdiri dari empat himpunan simbol:

- P adalah himpunan simbol predikat, termasuk variabel proposisional, biasanya dilambangkan dengan p, q, r , dengan atau tanpa subskrip, atau beberapa simbol relasi biner populer, seperti $<, \leq, \geq, >$, dll.
- F adalah himpunan simbol fungsi, termasuk konstanta, biasanya dilambangkan dengan a, b, c (untuk konstanta), f, g dan h , dengan atau tanpa subskrip.
- X adalah himpunan variabel, biasanya dilambangkan dengan x, y , dan z , dengan atau tanpa subskrip.
- Op adalah himpunan operator logika seperti $\neg, \wedge, \vee, \rightarrow$, dan $\leftrightarrow$, dan kuantifikasi $\forall$ dan $\exists$.

Untuk setiap simbol dalam $P \cup F$, jumlah argumen untuk simbol tersebut tetap dan ditentukan oleh fungsi aritas $aritas: P \cup F \rightarrow \mathbb{N}$. Jika $aritas(p) = 0$ untuk $p \in P$, p sama dengan variabel proposisional; jika $aritas(f) = 0$ untuk $f \in F$, f adalah konstanta.

Pasangan (P, F) dengan aritas disebut tanda tangan bahasa. Kita menyatakan bahasa L dengan $L = (P, F, X, Op)$.

Secara umum, P atau F bisa kosong; X dan Op tidak pernah bisa kosong. Sesuai konvensi, kita menggunakan $\{p/aritas(p) \mid p \in P\}$ dan $\{f/aritas(f) \mid f \in F\}$ untuk menunjukkan P dan F dengan aritas secara ringkas. Misalnya, $genap/1$ berarti $aritas(genap) = 1$. Secara teori, P atau F atau X bisa tak terbatas, tetapi dalam praktiknya, mereka selalu terbatas. Kita juga mengasumsikan bahwa tidak ada simbol yang muncul lebih

dari sekali dalam keempat himpunan ini. Alih-alih menetapkan satu bahasa tunggal untuk selamanya, tanda tangan yang berbeda menghasilkan bahasa orde pertama yang berbeda dan memungkinkan kita untuk menentukan simbol yang ingin kita gunakan untuk domain minat tertentu.

Contoh 5.1.2 Untuk menentukan sifat-sifat bilangan asli yang tercantum di awal bagian ini, kita menggunakan tanda tangan (P, F) , di mana $P = \{genap/1, ganjil/1, </2\}$, $F = \{0/0, s/1\}$. Arity diberikan setelah simbol / yang mengikuti setiap predikat atau simbol fungsi.

Setelah tanda tangan diberikan, kita menambahkan beberapa variabel X dan operator logika Op , untuk mendapatkan alfabet untuk bahasa orde pertama. Logika orde pertama memungkinkan kita untuk membangun ekspresi kompleks dari alfabet. Dimulai dengan variabel dan konstanta, kita dapat menggunakan simbol fungsi untuk membangun ekspresi majemuk yang disebut istilah.

Definisi 5.1.3 Diberikan bahasa orde pertama $L(P, F, X, Op)$, himpunan istilah L adalah himpunan string yang dibentuk oleh F dan X , yang dapat didefinisikan secara formal oleh tata bahasa BNF berikut:

$$\begin{aligned} \langle Constants \rangle &::= a \text{ if } a \in F, \text{arity}(a) = 0 \\ \langle Variables \rangle &::= x \text{ if } x \in X \\ \langle Terms \rangle &::= \langle Constants \rangle \mid \langle Variables \rangle \mid \\ &\quad f(\langle Terms \rangle_1, \langle Terms \rangle_2, \dots, \langle Terms \rangle_k) \text{ if } f \in F, \text{arity}(f) = k \end{aligned}$$

Himpunan istilah dalam Istilah sering dilambangkan dengan $T(F, X)$. Suatu istilah disebut istilah dasar jika tidak mengandung variabel apa pun. Himpunan semua istilah dasar dilambangkan dengan $T(F)$. Perhatikan bahwa $T(F) = \emptyset$ jika F tidak mengandung konstanta.

Untuk $F = \{0, s\}$ pada contoh sebelumnya, $T(F) = \{0, s(0), s(s(0)), \dots\}$ dan untuk $X = \{x, y\}$, $T(F, X) = \{0, s(0), s(x), s(y), s(s(0)), s(s(x)), s(s(y)), \dots\}$

Secara intuitif, istilah-istilah tersebut menunjukkan objek dalam domain yang dimaksud. Dalam suatu istilah, selain simbol dari F dan X , tanda kurung "(" dan ")" dan koma "," juga digunakan untuk mengidentifikasi strukturnya. Seperti setiap rumus proposisional memiliki pohon rumus (Definisi 2.1.1), untuk setiap istilah kita dapat memiliki pohon istilah di mana tanda kurung dan koma dihilangkan. Kita dapat menetapkan posisi, yang merupakan urutan bilangan bulat positif, untuk setiap simpul pohon istilah dan menggunakan posisi ini untuk mengidentifikasi subistilah yang ditunjukkan oleh simpul tersebut.

Definisi 5.1.4 Diberikan istilah t , posisi p dari t , dengan subistilah pada p , yang dilambangkan dengan t/p , didefinisikan secara rekursif sebagai berikut:

- Jika t adalah konstanta atau variabel, maka $p = \epsilon$, barisan kosong, dan $t/p = t$.
- Jika $t = f(t_1, \dots, t_k)$, maka $p = \epsilon$ dan $t/p = t$, atau $p = i.q$, di mana $1 \leq i \leq k$ dan $t/p = t_i/q$, dengan asumsi $p.\epsilon = \epsilon.p = p$ untuk setiap p .

Misalnya, jika $t = f(x, g(g(y)))$, maka posisi legal t adalah $\epsilon, 1, 2, 2.1$, dan $2.1.1$, dan subistilah yang bersesuaian adalah $t, x, g(g(y)), g(y)$, dan y , masing-masing.

Sekarang, dengan menambahkan operator logika biasa dari logika proposisional ditambah dua kuantifikasi, kita memiliki hampir semua simbol yang dibutuhkan untuk bahasa orde pertama.

Definisi 5.1.5 Diberikan bahasa orde pertama $L = (P, F, X, Op)$, rumus-rumus L adalah himpunan string yang dibangun oleh istilah, simbol predikat, dan operator logika. Misalkan op adalah operator biner yang digunakan dalam aplikasi saat ini, maka rumus-rumus untuk aplikasi ini dapat didefinisikan oleh tata bahasa BNF berikut:

$$\begin{aligned}
 op &::= \wedge \mid \vee \mid \rightarrow \mid \oplus \mid \leftrightarrow \\
 Atoms &::= p(\langle Terms \rangle_1, \langle Terms \rangle_2, \dots, \langle Terms \rangle_k) \text{ if } p \in P, \text{arity}(p) = k \\
 Literals &::= \langle Atoms \rangle \mid \neg \langle Atoms \rangle \\
 Formulas &::= \top \mid \perp \mid \langle Atoms \rangle \mid \neg \langle Formulas \rangle \mid \\
 &\quad (\langle Formulas \rangle \langle op \rangle \langle Formulas \rangle) \mid \\
 &\quad (\forall \langle Variables \rangle \langle Formulas \rangle) \mid (\exists \langle Variables \rangle \langle Formulas \rangle)
 \end{aligned}$$

Suatu rumus (klausa, literal, atom) disebut dasar jika tidak mengandung variabel apa pun.

Atom adalah singkatan dari rumus atomik. Kita dapat memiliki pohon rumus untuk setiap rumus orde pertama.

Beberapa tanda kurung tidak diperlukan jika kita menggunakan prioritas di antara operator logika. Saat menampilkan rumus, biasanya tanda kurung terluar atau tanda kurung yang mengikuti simbol predikat dengan aritas nol dihilangkan, misalnya, kita menulis p alih-alih $p()$. Kita juga dapat mendefinisikan subrumus dari suatu rumus seperti yang kita lakukan untuk rumus proposisional dan istilah.

Contoh 5.1.6 Diberikan tanda tangan (P, F) , di mana $P = \{child/2, love/2\}$ dan $F = \{a_i/0 \mid 1 \leq i \leq 100\}$, maka $T(F) = F$ dan $T(F, X) = F \cup X$, karena F tidak memiliki simbol fungsi non-konstan. Jika $X = \{x, y, z\}$, berikut adalah beberapa contoh rumus:

- **Atom dasar:** $child(a_2, a_1), child(a_3, a_1), love(a_1, a_2), \dots$
- **Atom non-dasar:** $child(x, y), love(y, x), love(z, x), \dots$

- **Rumus dasar non-atom:** $\neg child(a_3, a_4), child(a_2, a_1) \vee child(a_3, a_1), child(a_4, a_1) \rightarrow love(a_1, a_4), \dots$
- **Rumus umum:** $\neg child(a_1, y), \neg child(x, a_2), \neg child(x, a_2), \neg child(x, y) \vee love(y, x), love(x, y) \vee \neg love(y, x), \dots$

Dalam contoh-contoh ini, jika kita mengganti setiap atom dengan variabel proposisional, maka rumus tersebut menjadi rumus proposisional. Hal ini berlaku untuk semua rumus tanpa kuantifikasi. Contoh rumus dengan kuantifikasi diberikan pada subbagian berikutnya.

Kuantifikasi

Yang membuat logika orde pertama kuat adalah kemampuannya untuk membuat pernyataan umum menggunakan kuantifikasi: Kuantifikasi universal $\forall$ dan kuantifikasi eksistensial $\exists$. Dalam definisi Rumus, setiap kemunculan kuantifikasi diikuti oleh variabel, misalnya x , dan kemudian diikuti oleh rumus, misalnya A . Artinya, kuantifikasi muncul dalam rumus seperti $\forall x A$ atau $\exists x A$; yang pertama mengatakan " A benar untuk setiap nilai x ," dan yang kedua mengatakan " A benar untuk beberapa nilai x ."

Dalam aljabar sekolah menengah, ketika kita mengatakan komutatif $+$, kita menggunakan rumus $x + y = y + x$, di sini x dan y dimaksudkan untuk semua nilai. Ketika menyelesaikan persamaan seperti $x^2 + 3x + 2 = 0$, kita mencoba menemukan beberapa nilai x sehingga persamaan tersebut berlaku. Dalam logika orde pertama, kedua kasus tersebut dinyatakan sebagai $\forall x \forall y (x + y = y + x)$ dan $\exists x (x^2 - 3x + 2 = 0)$, masing-masing.

Dalam bahasa alami, kalimat yang menggunakan kata-kata seperti "setiap," "selalu," atau "tidak pernah," biasanya diterjemahkan ke dalam rumus logika orde pertama dengan $\forall$:

- "Semua yang dilakukan Ada sempurna.": $x (did(Ada, x) \rightarrow perfect(x))$, di mana $did(x, y)$ benar jika dan hanya jika x melakukan y . Jika kita dapat menemukan sesuatu yang tidak dilakukan Ada dengan sempurna, maka kalimat tersebut salah.
- "Bob selalu terlambat untuk janji temu!": $\forall x (appointment(x) \rightarrow late(Bob, x))$, di mana $late(x, y)$ berarti x terlambat untuk acara y . Kalimat ini salah jika Bob tepat waktu sekali.
- "Cathy tidak pernah menyukai makanan yang dimasak oleh Tom.": $\forall x (cook(Tom, x) \rightarrow like(Cathy, x))$, di mana $cook(x, y)$ benar jika dan hanya jika x memasak y .

Ketiga kalimat di atas kemungkinan merupakan contoh dari kekeliruan generalisasi tergesa-gesa, yang kadang-kadang disebut kekeliruan generalisasi berlebihan. Pada dasarnya, ini adalah membuat klaim berdasarkan bukti yang terlalu sedikit. Secara umum, orang berasumsi bahwa "setiap aturan memiliki pengecualian." Namun, dalam logika orde pertama, jika pengecualian tidak dinyatakan secara eksplisit dalam suatu aturan, maka aturan tersebut tidak mungkin benar dan karenanya tidak dapat digunakan dalam suatu argumen.

Contoh 5.1.7 Dari Contoh 5.1.6, jika $child(x, y)$ berarti " x adalah anak dari y ," $parent(x, y)$ berarti " x adalah orang tua dari y ," $descen(x, y)$ berarti " x adalah keturunan dari y ," dan $love(x, y)$ berarti " x mencintai y ," dengan

menggunakan predikat-predikat ini, kita dapat menulis banyak pernyataan dalam logika orde pertama.

- x adalah anak dari y jika dan hanya jika y adalah orang tua dari x : $\forall x \forall y (child(x, y) \leftrightarrow parent(y, x))$.
- Jika x adalah anak dari y , maka x adalah keturunan dari y : $\forall x \forall y (child(x, y) \rightarrow descen(x, y))$.
- Setiap orang tua mencintai anaknya: $\forall x \forall y (parent(x, y) \rightarrow love(x, y))$.
- Setiap orang dicintai oleh seseorang: $\forall x \exists y love(y, x)$.
- Hubungan keturunan bersifat transitif: $\forall x \forall y \forall z (descen(x, y) \wedge descen(y, z) \rightarrow descen(x, z))$.

Kita dapat mendefinisikan banyak hubungan yang lebih kompleks menggunakan predikat-predikat ini.

Saat menggunakan $\forall$ untuk menggambarkan suatu pernyataan dalam berbagai kondisi, kondisi-kondisi ini biasanya merupakan argumen pertama dari $\rightarrow$. Saat menggunakan $\exists$ untuk menggambarkan suatu pernyataan dalam berbagai kondisi, kita tidak dapat menggunakan $\rightarrow$. Misalnya, misalkan $own(x, y)$ berarti " x owns y ," $red(x)$ berarti " x berwarna merah," $toy(x)$ berarti " x adalah mainan," dan $car(x)$ berarti " x adalah mobil." Maka pernyataan "Tom mencintai setiap mainan merah yang dimilikinya" dapat diungkapkan sebagai

$$\forall x (own(Tom, x) \wedge red(x) \wedge toy(x) \rightarrow love(Tom, x)).$$

Wajar untuk membacanya sebagai kuantifikasi terbatas: "Untuk setiap x , jika Tom memiliki x , x berwarna merah dan x adalah mainan, maka Tom mencintai x ." Tentu saja, jika kita menghilangkan $\rightarrow$ dalam rumus di atas, kita memiliki

$$\forall x (\neg own(Tom, x) \vee \neg red(x) \vee \neg toy(x) \vee love(Tom, x)),$$

yang dapat dibaca sebagai "untuk setiap x , Tom tidak memiliki x , x tidak berwarna merah, x bukan mainan, atau Tom mencintai x ." Perhatikan bahwa jika $\rightarrow$ diganti $\wedge$ dengan dalam rumus di atas, kita mendapatkan

$$\forall x (own(Tom, x) \wedge red(x) \wedge toy(x) \wedge love(Tom, x))$$

yang menyatakan bahwa "Tom memiliki segalanya," "segalanya berwarna merah," "segalanya adalah mainan," dan "Tom mencintai segalanya."

Di sisi lain, untuk menyatakan "Tom mencintai salah satu mobil mainan merahnya," rumusnya adalah

$$\exists x (own(Tom, x) \wedge toy(x) \wedge red(x) \wedge car(x) \wedge love(Tom, x)).$$

Jika kita mengganti yang terakhir dengan dalam rumus di atas, maka rumus yang dihasilkan dapat diubah menjadi rumus ekuivalen berikut:

$$\exists x (\neg \text{own}(\text{Tom}, x) \vee \neg \text{toy}(x) \vee \neg \text{red}(x) \vee \neg \text{car}(x) \vee \text{love}(\text{Tom}, x)),$$

yang menyatakan bahwa ada sesuatu, yang dilambangkan dengan "itu," dan satu (atau lebih) dari hal berikut ini benar: "Tom tidak memilikinya," "itu bukan mainan," "itu bukan merah," "itu bukan mobil," atau "Tom menyukainya." Karena ada banyak hal yang "Tom tidak memilikinya," "itu bukan mainan," "itu bukan merah," atau "itu bukan mobil," rumus di atas tidak menangkap makna bahwa "Tom menyukai salah satu mobil mainan merahnya."

Definisi 5.1.8 Dalam $\forall x A$ atau $\exists x A$, *cakupan* variabel x adalah A , dan setiap kemunculan x dalam A dikatakan *terbatas*. Variabel dalam A dikatakan *bebas* jika tidak terbatas. Suatu rumus A dikatakan *tertutup* jika tidak memiliki variabel bebas. Suatu *kalimat* adalah rumus tertutup.

Dalam $\text{love}(x, y)$, baik x maupun y adalah bebas; dalam $\exists y p(x, y)$, dan x adalah bebas, y terbatas (kita tidak dapat mengatakan " y dalam $p(x, y)$ terbatas" sebagai " y bebas dalam $p(x, y)$ "); dalam $\forall x \exists y p(x, y)$, baik x maupun y bukan bebas; dengan demikian, rumus tersebut tertutup; dalam $(\forall x q(x)) \vee r(x)$, kemunculan pertama x terbatas dan kemunculan kedua x bebas.

Ketika A mewakili rumus apa pun, kita menulis $A(x_1, x_2, \dots, x_n)$ untuk menunjukkan bahwa variabel bebas dari A adalah di antara $x_1, x_2, \dots, x_n$.

Seperti halnya rumus proposisional, kita dapat menggunakan prioritas pada operator logika untuk menghindari penulisan semua tanda kurung dalam sebuah rumus. Hubungan prioritas yang digunakan dalam Bab 2 adalah

$$\neg, \quad \wedge, \quad \vee, \quad \rightarrow, \quad \{\oplus, \leftrightarrow\}$$

Di mana menempatkan $\forall$ dan $\exists$? Sebagai praktik umum, kita akan menempatkannya tepat setelah $\neg$.

$$\neg, \quad \{\forall, \exists\}, \quad \wedge, \quad \vee, \quad \rightarrow, \quad \{\oplus, \leftrightarrow\}.$$

Artinya, dalam rumus, $\forall$ atau $\exists$ diperlakukan sebagai $\neg$ ketika operator Boolean biner hadir. Misalnya, $\forall x A \vee B$ diinterpretasikan sebagai $(\forall x A) \vee B$ (cakupan x adalah A), sama seperti $\neg A \vee B$ bukan $\neg(A \vee B)$. Kita akan menulis $\forall x (A \vee B)$ untuk menunjukkan bahwa cakupan x adalah $A \vee B$. Seperti biasa, tanda kurung yang dihilangkan dapat ditambahkan kembali untuk menghindari kemungkinan kebingungan.

Sekarang, dengan menggunakan predikat dan simbol relasi, kita dapat membuat rumus tentang pernyataan yang disebutkan di awal bagian ini:

- $\forall x (odd(x) \leftrightarrow \neg even(x))$: Setiap bilangan asli adalah genap atau ganjil, tetapi tidak keduanya.
- $\forall x (even(s(x)) \leftrightarrow odd(x))$: Suatu bilangan asli x adalah ganjil jika dan hanya jika $s(x)$, yang menunjukkan $x + 1$, adalah genap.
- $\forall x (x < s(x) \wedge \neg(x < x))$: Untuk setiap bilangan asli x , $x < x + 1$ dan $(x < x)$ adalah salah.
- $\forall x \exists y x < y$: Untuk setiap bilangan asli x , terdapat bilangan asli y , sedemikian sehingga $x < y$.
- $\forall x \forall y \forall z ((x < y) \wedge (y < z) \rightarrow (x < z))$: Untuk setiap tiga bilangan asli x, y , dan z , jika $x < y$ dan $y < z$, maka $x < z$, yaitu, $<$ bersifat transitif.

Logika Tak Terurut dan Banyak Terurut

Bahasa orde pertama umumnya dianggap sebagai bahasa tak terurut. Jika kita melihat sintaksnya dengan saksama, dua himpunan objek didefinisikan: (*Terms*) dan (*Formulas*). Jelas, (*Formulas*) berjenis Boolean, dilambangkan dengan $Bool = \{0, 1\}$. (*Terms*) mewakili objek yang diminati, dan himpunan objek tersebut umumnya disebut semesta (juga disebut semesta wacana atau domain wacana) dari bahasa tersebut. Misalnya, kita dapat menggunakan $T(F)$, himpunan istilah dasar, untuk mewakili semesta.

Jika kita menggunakan U untuk menunjukkan jenis *Terms*, untuk setiap tanda tangan (P, F) , simbol predikat $p \in P$ mewakili fungsi $p: U^{arity(p)} \rightarrow Bool$ dan simbol fungsi $f \in F$ mewakili fungsi $f: U^{arity(f)} \rightarrow U$. Kuantifikasi universal dan eksistensial mencakup semesta U . Misalnya, bahasa orde pertama dalam Contoh 5.1.6 berbicara tentang 100 orang yang tinggal di kota tertentu, dengan relasi $love(x, y)$ untuk menyatakan bahwa x mencintai y . Dalam bahasa seperti itu, kita dapat menyatakan pernyataan bahwa “setiap orang mencintai seseorang” dengan menulis $\forall x \exists y love(x, y)$, dengan asumsi U adalah himpunan 100 orang.

Untuk banyak aplikasi, $Bool$ dan U saling lepas, seperti pada Contoh 5.1.2 dan 5.1.6. Untuk beberapa aplikasi, $Bool$ dapat dianggap sebagai himpunan bagian dari U . Misalnya, untuk memodelkan perintah *if – then – else* dalam bahasa pemrograman, kita dapat menggunakan simbol *ite*: $Bool \times U \times U \rightarrow U$. Contoh lain adalah simbol fungsi *says*: $U \times Bool \rightarrow Bool$ dalam teka-teki Knight and Knave (Contoh 2.5.4).

Tidak sulit untuk memperkenalkan sort (atau tipe) dalam logika orde pertama dengan menggunakan simbol predikat unary (atau monadik) ($arity = 1$). Dalam Contoh 5.1.6, kita menggunakan a_i untuk mewakili orang i . Jika kita membutuhkan fungsi $age(x)$ untuk memberi tahu usia orang x , kita membutuhkan simbol fungsi untuk bilangan asli. Jika kita menggunakan simbol-simbol tersebut dalam Contoh 5.1.2, maka $F = \{0, s, a_i \mid 1 \leq i \leq 100\}$. Dalam aplikasi ini, kita perlu memberi tahu mana yang merupakan orang dan mana yang merupakan bilangan dengan menggunakan simbol predikat berikut:

- $person : U \rightarrow Bool$: $person(x)$ bernilai benar jika dan hanya jika x adalah orang.
- $nat : U \rightarrow Bool$: $nat(x)$ bernilai benar jika dan hanya jika x adalah bilangan asli.

$person(x)$ dapat dengan mudah didefinisikan dengan menyatakan $person(a_i)$ untuk $1 \leq i \leq 100$. $nat(x)$ juga dapat didefinisikan dengan menyatakan $nat(0)$ dan $nat(s(x)) \equiv nat(x)$. Dengan menggunakan nat , kita dapat menghindari istilah yang tidak bermakna seperti $s(a_i)$ karena $nat(s(a_i))$ bernilai salah. Sekarang, untuk menyatakan sifat umum seperti pada Contoh 5.1.2 dan 5.1.6, kita harus memberikan batasan pada variabel. Misalnya, untuk menyatakan “setiap orang dicintai oleh seseorang,” rumusnya sekarang adalah

$$\forall x \exists y (person(x) \wedge person(y) \rightarrow love(y, x)).$$

Untuk mengatakan “<” bersifat transitif pada bilangan asli, kita menggunakan

$$\forall x, y, z (nat(x) \wedge nat(y) \wedge nat(z) \wedge (x < y) \wedge (y < z) \rightarrow (x < z))$$

Menangani predikat unary seperti itu sangat melelahkan. Untuk menghilangkan kerumitan tersebut, orang memperkenalkan “*sort*”. Misalkan

$$\begin{aligned} Person &= \{x \mid person(x) \wedge x \in U\} \\ Nat &= \{x \mid nat(x) \wedge x \in U\} \end{aligned}$$

Kemudian definisikan $love: Person^2 \rightarrow Bool$ dan $<: Nat^2 \rightarrow Bool$ dan rumusnya menjadi rapi kembali: Kita tidak membutuhkan $person(x)$, $nat(x)$, dll., dalam rumus tersebut. Untuk fungsi usia (*age*), kita memiliki $age: Person \rightarrow Nat$. Itulah awal mula *many – sorted – logic*, yang merupakan perluasan ringan dari logika orde pertama.

Gagasan banyak jenis sangat berguna dalam bahasa pemrograman di mana setiap jenis adalah himpunan objek yang serupa. Himpunan biasanya didefinisikan oleh predikat unary. Dari sudut pandang seorang ahli logika, himpunan $S = \{x \mid p(x)\}$ dan predikat $p(x)$ memiliki arti yang sama. Dalam logika banyak jenis, seseorang dapat memiliki berbagai jenis objek—seperti orang dan bilangan asli—dan himpunan variabel terpisah yang mencakup masing-masing jenis. Selain itu, spesifikasi simbol fungsi dan simbol predikat menunjukkan jenis argumen apa yang mereka harapkan (domain), dan, dalam kasus simbol fungsi, jenis nilai apa yang mereka kembalikan (rentang). Konsep aritas dan tanda tangan diperluas untuk memuat informasi domain dan rentang dari setiap simbol fungsi dan predikat.

Logika multisort dapat secara formal mencerminkan niat kita untuk tidak menangani semesta sebagai kumpulan objek, tetapi untuk mempartisi semesta dengan cara yang mirip dengan tipe dalam bahasa pemrograman. Partisi ini dapat dilakukan pada tingkat sintaksis: Substitusi variabel hanya dapat dilakukan sesuai dengan “*sort*”. Kita akan kembali ke topik ini di Bagian 5.2 setelah memperkenalkan semantik logika orde pertama.

5.2 SEMANTIK

Kita harus menyadari bahwa, pada tahap ini, semua simbol fungsi dan predikat hanyalah simbol. Dalam logika proposisional, simbol p dapat mewakili “Tom berlari cepat,”

atau "Mary tinggi." Hal yang sama berlaku dalam logika orde pertama karena kita dapat memberikan arti yang berbeda pada suatu simbol.

Dalam Contoh 5.1.7, kita telah melihat dua rumus berikut:

- $\forall x \forall y (child(x, y) \rightarrow descen(x, y)).$
- $\forall x \forall y (parent(x, y) \rightarrow love(x, y)).$

Rumus pertama mengatakan "jika x adalah anak dari y , maka x adalah keturunan dari y "; rumus kedua mengatakan "setiap orang tua mencintai anaknya." Apa perbedaan di antara keduanya? Perbedaannya hanya terletak pada simbol predikat. Jika kita memberikan makna yang berbeda pada simbol predikat, satu dapat menggantikan yang lain. Artinya, kita telah merancang bahasa dengan interpretasi tertentu dalam pikiran, tetapi seseorang juga dapat menginterpretasikan rumus yang sama secara berbeda.

Pada bagian ini, kita akan menjabarkan konsep interpretasi, model, dan rumus yang dapat dipenuhi atau valid, sebagai perluasan dari konsep-konsep logika proposisional ini.

Interpretasi

Untuk logika proposisional, interpretasi adalah penugasan variabel proposisional ke nilai kebenaran. Untuk bahasa orde pertama $L = (P, F, X, Op)$, kita membutuhkan interpretasi untuk setiap simbol L .

Definisi 5.2.1 Diberikan formula tertutup A dari $L = (P, F, X, Op)$, interpretasi dari A adalah rangkap tiga $I = (D, R, G)$, di mana:

- D adalah himpunan tak kosong yang disebut semesta I , sering dilambangkan dengan D^I , jika kita ingin menekankan bahwa D digunakan oleh I .
- Untuk setiap simbol predikat $p \in P$ dengan aritas k , terdapat relasi k -ary $r_p \in R$ atas D , $r_p \subseteq D^k$, yang dilambangkan dengan $r_p = p^I$. Jika $k = 0$, maka p adalah variabel proposisional dan p^I adalah 1 atau 0, penugasan kebenaran untuk p .
- Untuk setiap simbol fungsi $f \in F$ dengan aritas k , terdapat fungsi $g : D^k \rightarrow D \in G$, yang dilambangkan dengan $g = f^I$. Jika $k = 0$, maka $f^I \in D$.

Contoh 5.2.2 Perhatikan $L = (P, F, X, Op)$, di mana $P = \{p/2\}$, $F = \{a/0\}$, dan $X = \{x\}$. Maka untuk rumus $A = \forall x p(a, x)$, kita dapat memiliki interpretasi berikut:

1. $I_1 = (N, \{\leq\}, 0)$ dan arti dari A adalah "untuk setiap bilangan asli x , $0 \leq x$."
2. $I_2 = (N, \{|\}, \{1\})$, $x | y$ bernilai benar jika dan hanya jika x membagi y , dan arti dari A adalah "untuk setiap bilangan asli x , $1 | x$."
3. $I_3 = (\{0,1\}^3 \cup \{\epsilon\}, prefix, \{\epsilon\})$, $prefix(x, y)$ mengembalikan true jika dan hanya jika x adalah prefiks dari y , arti dari A adalah "untuk setiap string biner x dengan panjang 3 atau x kosong, string kosong ϵ adalah prefiks dari x ."

4. $I_4 = (P(N), \{C\}, \{\emptyset\})$, $P(N)$ adalah himpunan kuasa dari N , dan arti dari A adalah “untuk setiap himpunan bagian x dari bilangan asli, $\emptyset \subseteq x$.”
5. $I_5 = (V, \{E\}, \{a\})$, di mana $V = \{a, b, c\}$, $E = \{(a, a), (a, b), (a, c), (c, c)\}$, $G = (V, E)$ adalah sebuah graf, dan arti dari A adalah “untuk setiap simpul $x \in V$, terdapat sisi $(a, x) \in E$.”

Dalam logika proposisional, setelah kita memiliki interpretasi, kita dapat menggunakan prosedur seperti *eval* (Algoritma 2.2.4) untuk mendapatkan nilai kebenaran dari suatu formula. Kita dapat melakukan hal yang sama dalam logika orde pertama, asalkan kita memiliki cara untuk menangani variabel bebas dan kuantifikasi dalam formula tersebut.

Misalkan $free(A)$ adalah variabel bebas dari formula A . Diberikan semesta D , sebuah penugasan adalah pemetaan $\theta: free(A) \rightarrow D$, yang memetakan setiap variabel bebas dari A ke elemen D . Tujuan dari θ adalah bahwa, selama evaluasi suatu formula di bawah interpretasi, jika kita melihat variabel bebas x , kita akan menggunakan nilai $\theta(x)$ untuk x . Awalnya, setiap variabel bebas dari A terpengaruh dalam θ . Jika A tertutup, maka θ kosong. Saat kuantifikasi dihilangkan dan variabel menjadi bebas selama evaluasi, kita memperbarui θ secara dinamis.

Prosedur 5.2.3 Prosedur *eval* menerima input berupa rumus atau istilah A , dengan atau tanpa variabel bebas, dalam $L = (P, F, X, Op)$, interpretasi I untuk L , dan penugasan $\theta: free(A) \rightarrow D^I$, dan mengembalikan nilai Boolean untuk rumus dan elemen DI untuk suatu istilah. Jika $A = f(t_1, t_2, \dots, t_k)$ dan $f \in P$, maka A adalah sebuah atom.

```

proc eval( $A, I, \theta$ )
  if  $A = \top$  return 1 // 1 means true.
  if  $A = \perp$  return 0 // 0 means false.
  if  $A \in X$  return  $\theta(A)$  //  $A$  is a free variable.
  if  $A = f(t_1, t_2, \dots, t_k)$  return  $f^I(t'_1, t'_2, \dots, t'_k)$ , where  $t'_i = eval(t_i, I, \theta)$ 
  if  $A = \neg B$  return  $\neg eval(B, I, \theta)$ 
  if  $A = (B \text{ op } C)$  return  $eval(B, I, \theta) \text{ op } eval(C, I, \theta)$ , where  $op \in Op$ 
  if  $A = (\forall x B)$  return  $allInD(B, I, \theta, x, D^I)$ 
  if  $A = (\exists x B)$  return  $someInD(B, I, \theta, x, D^I)$ 
  else return “unknown”

```

Prosedur $allInD(B, I, \theta, x, S)$ mengambil rumus B , interpretasi I , penugasan θ , variabel bebas x dalam B , dan $S \subseteq D$ dan mengevaluasi nilai B di bawah I dan $\theta \cup \{x \sim d\}$ untuk setiap nilai $d \in S$. Jika $eval(B, I, \theta \cup \{x \sim d\})$ mengembalikan 0 untuk satu $d \in D^I$, kembalikan 0; jika tidak, kembalikan 1.

```

proc allInD( $B, I, \theta, x, S$ )
  if  $S = \emptyset$  return 1
  pick  $d \in S$ 
  if ( $eval(B, I, \theta \cup \{x \sim d\}) = 0$ ) return 0
  return allInD( $B, I, \theta, x, S - \{d\}$ )

```

Prosedur *someInD*(B, I, θ, x, S) bekerja dengan cara yang sama seperti *allInD*, kecuali jika salah satu evaluasi, $eval(B, I, \theta \cup \{x \sim d\})$, mengembalikan 1, kembalikan 1; Jika tidak, kembalikan 0.

```

proc someInD( $B, I, \theta, x, S$ )
  if  $S = \emptyset$  return 0
  pick  $d \in S$ 
  if ( $eval(B, I, \theta \cup \{x \sim d\}) = 1$ ) return 1
  return someInD( $B, I, \theta, x, S - \{d\}$ )

```

Jelas, ketika domain I, D^I , tak terbatas, baik *allInD*(B, I, θ, x, S) maupun *someInD*(B, I, θ, x, S) tidak akan berhenti, dan eval tidak dapat menjadi algoritma. Dalam hal ini, kita memandangnya sebagai definisi matematika rekursif. Meskipun nilai-nilai prosedur ini tidak dapat ditentukan oleh algoritma, kita dapat menalar tentangnya menggunakan induksi matematika. Di bagian terakhir bab ini, kita akan melihat bahwa D^I dapat direpresentasikan oleh basis Herbrand, yang merupakan himpunan semua rumus atom dasar. Karena setiap basis Herbrand dapat dihitung, urutan yang ditemukan dengan baik ada untuk D^I dan kita dapat menerapkan induksi pada D^I . Kami menyajikan eval sebagai prosedur karena kami berasumsi bahwa pembaca sudah familiar dengan algoritma. Ketika D^I terbatas, penghentian eval mudah dibuktikan.

Proposisi 5.2.4 Diberikan formula A dari $L = (P, F, X, Op)$, interpretasi I , dan penugasan θ : $free(A) \subseteq D^I$, $eval(A, I, \theta)$ akan mengembalikan nilai Boolean; jika t adalah istilah yang muncul dalam A , maka $eval(t, I, \theta)$ akan mengembalikan nilai D^I .

Bukti Sketsa Misalkan D adalah D_I . Kita melakukan induksi pada struktur pohon A . Kasus dasar ketika A atau t adalah simbol dengan aritas 0 adalah trivial. Jika $t = f(t_1, t_2, \dots, t_k)$ adalah istilah dalam A , nilai yang dikembalikan adalah $f^I(t'_1, t'_2, \dots, t'_k) \in D^I$ karena $f^I : D^k \rightarrow D$ adalah sebuah fungsi. Jika A adalah atom $p(t_1, t_2, \dots, t_k)$, di mana $p \in P$, maka p^I adalah relasi dan $p^I(t'_1, t'_2, \dots, t'_k) = 1$ jika dan hanya jika $(t'_1, t'_2, \dots, t'_k) \in p^I$ berdasarkan asumsi kita. Artinya, $p^I(t'_1, t'_2, \dots, t'_k)$ adalah nilai Boolean. Untuk rumus non-atom, nilai yang dikembalikan selalu berupa nilai Boolean seperti yang ditunjukkan oleh proposisi berikutnya.

Mulai sekarang, kita akan menggunakan notasi $eval(A, I, \theta)$ dengan $I(A, \theta)$ dan $eval(A, I,)$ dengan $I(A)$ untuk mempersingkat penulisan.

Proposisi 5.2.5 Untuk rumus A dan B , interpretasi I , dan penugasan θ ,

- $I(\neg A, \theta) = \neg I(A, \theta)$
- $I(A \vee B, \theta) = I(A, \theta) \vee I(B, \theta)$
- $I(A \wedge B, \theta) = I(A, \theta) \wedge I(B, \theta)$
- $I(A \oplus B, \theta) = I(A, \theta) \oplus I(B, \theta)$
- $I(A \leftrightarrow B, \theta) = I(A, \theta) \leftrightarrow I(B, \theta)$
- $I(\forall x B(x), \theta) = \text{allInD}(B(x), I, \theta, x, D^I)$
- $I(\exists x B(x), \theta) = \text{someInD}(B(x), I, \theta, x, D^I)$

Dari definisi *eval*, mudah untuk memeriksa bahwa proposisi di atas berlaku.

Contoh 5.2.6 Untuk I_5 dalam Contoh 5.2.2, $I_5(\forall x p(a, x))$ akan memanggil *allInD* dengan parameter $(p(a, x), I_5, \emptyset, x, \{a, b, c\})$, yang akan memanggil *eval* tiga kali, dengan $\theta = \{x \sim a\}$, $\{x \sim b\}$, dan $\{x \sim c\}$, masing-masing. Karena E berisi (a, x) untuk $x \in \{a, b, c\}$, ketiga panggilan *eval* mengembalikan 1. Hasil akhirnya dapat dinyatakan sebagai:

$$I_5(\forall x p(a, x)) = \bigwedge_{d \in \{a, b, c\}} I_5(p(a, x), \{x \sim d\})$$

Kita dapat melakukan hal yang sama untuk semua interpretasi dalam Contoh 5.2.2. Dengan menggeneralisasi dari contoh di atas,

$$I(\forall x B(x), \theta) = \bigwedge_{d \in D^I} I(B(x), \theta \cup \{x \sim d\})$$

yang menangkap makna bahwa " $I(\forall x B(x), \theta)$ mengembalikan 1 jika dan hanya jika $B(x)$ dievaluasi menjadi 1 di bawah I dan $\theta \cup \{x \sim d\}$ untuk setiap nilai $d \in D$."

Demikian pula, kita memiliki notasi untuk $I(\exists x B(x), \theta)$, atau setara dengan *someInD*($B(x), I, \theta, x, D$):

$$I(\exists x B(x), \theta) = \bigwedge_{d \in D^I} I(B(x), \theta \cup \{x \sim d\})$$

yang menangkap makna bahwa " $I(\exists x B(x), \theta)$ mengembalikan 1 jika dan hanya jika $B(x)$ dievaluasi menjadi benar di bawah I dan $\theta \cup \{x \sim d\}$ untuk beberapa nilai $d \in D^I$." Notasi ini akan membantu kita memahami rumus terkuantifikasi dan menetapkan sifat-sifatnya. Perhatikan bahwa jika domain D^I kosong, maka $I(\forall x B(x)) = 1$ dan $I(\exists x B(x)) = 0$

Model, Keterpuasan, dan Validitas

Berikut ini hanyalah perluasan langsung dari logika proposisional ke logika orde pertama. Ingat bahwa kalimat adalah rumus tertutup.

Definisi 5.2.7 Diberikan kalimat A dan interpretasi I , I dikatakan sebagai model dari A jika $I(A) = 1$. Jika A memiliki model, A dikatakan dapat dipuaskan.

Masalah keterpuasan logika orde pertama adalah untuk memutuskan apakah suatu himpunan rumus orde pertama dapat dipuaskan atau tidak. Masalah ini juga disebut pemenuhan kendala di mana kendalanya adalah himpunan formula orde pertama. Masalah SAT dan masalah pemrograman linier adalah kasus khusus dari pemenuhan kendala.

Untuk formula proposisional, jumlah modelnya terbatas; untuk formula orde pertama, jumlah modelnya umumnya tak terbatas, karena kita memiliki jumlah pilihan yang tak terbatas untuk domain, relasi, dan fungsi dalam suatu interpretasi. Namun, kita masih dapat meminjam notasi $M(A)$ dari logika proposisional.

Definisi 5.2.8 Diberikan formula tertutup A , misalkan $M(A)$ adalah himpunan semua model dari A . Jika $M(A) \neq \emptyset$, maka A tidak dapat dipenuhi; jika $M(A)$ mengandung setiap interpretasi, yaitu, setiap interpretasi adalah model dari A , maka A valid.

Dengan definisi yang identik dari logika proposisional, hasil berikut diharapkan dalam logika orde pertama.

Proposisi 5.2.9 Setiap formula proposisional yang valid adalah formula yang valid dalam logika orde pertama, dan setiap formula proposisional yang tidak dapat dipenuhi adalah tidak dapat dipenuhi dalam logika orde pertama.

Tentu saja, ada lebih banyak formula yang valid dalam logika orde pertama dan perhatian kita tertuju pada formula dengan kuantifikasi.

Contoh 5.2.10 Mari kita periksa beberapa contoh.

1. A_1 is $\forall x (p(a, x) \rightarrow p(a, a))$.

Consider $I = (\{a, b\}, \{p\}, \{a\})$, where $p^I = \{\langle a, a \rangle\}$, $a^I = a$, $I(A_1) = 1$ because $p^I(a, a) = 1$ and

$$\begin{aligned} I(A_1) &= \bigwedge_{d \in \{a, b\}} I(p(a, x) \rightarrow p(a, a), \{x \sim d\}) \\ &= I(p(a, a) \rightarrow p(a, a)) \wedge I(p(a, b) \rightarrow p(a, a)) \\ &= 1 \wedge (p^I(a, b) \rightarrow p^I(a, a)) \\ &= 1 \end{aligned}$$

Misalkan $I' = (\{a, b\}, \{\{a, b\}\}, \{a\})$ maka $I'(A_1) = 0$ karena $p^{I'}(a, a) = 0$ dan $p^{I'}(a, b) = 1$ menyiratkan bahwa $I'(p(a, b) \rightarrow p(a, a)) = 0$. $I'(A_1) = I'(p(a, a) \rightarrow p(a, a)) \wedge I'(p(a, b) \rightarrow p(a, a)) = 1 \wedge 0 = 0$. Dengan demikian, A_1 dapat dipenuhi tetapi tidak valid.

2. A_2 adalah $(\forall x p(a, x)) \rightarrow p(a, a)$

A_2 berbeda dari A_1 karena cakupan x berbeda di A_1 dan A_2 . Untuk setiap interpretasi I , kita mempertimbangkan nilai kebenaran dari $p^I(a^I, a^I)$. Jika $p^I(a^I, a^I) = 1$, maka $I(A_2, \emptyset) = 1$. Jika $p^I(a^I, a^I) = 0$, maka $I(\forall x p(a, x)) = 0$ karena x akan mengambil nilai a^I . Jadi, $I(A_2) = 1$ dalam setiap kasus. Karena I bersifat sembarang, A_2 harus valid.

3. A_3 adalah $\exists x \exists y (p(x) \wedge \neg p(y))$.

Perhatikan $I = (\{a, b\}, \{p\}, \emptyset)$, di mana $p(a) = 1$ dan $p(b) = 0$, maka $p(a) \wedge \neg p(b) = 1$. Dari $I(p(x) \wedge \neg p(y), x \sim a, y \sim b) = 1$, kita simpulkan bahwa I adalah model dari A_3 . A_3 salah dalam interpretasi apa pun yang domainnya kosong atau hanya berisi satu elemen. Dengan demikian, A_3 tidak valid.

Definisi 5.2.11 Misalkan $A(x)$ adalah sebuah rumus di mana x adalah variabel bebas dari A . Maka $A(t)$ menyatakan rumus $A(x)$ $x \sim t$, di mana setiap kemunculan x digantikan oleh suku t . Rumus $A(t)$ disebut sebagai contoh dari $A(x)$ dengan mengganti x dengan t (lihat Definisi 2.1.3).

Kita akan menggunakan substitusi secara ekstensif di bab berikutnya. Substitusi $\theta = X \rightarrow D$ yang digunakan dalam prosedur *eval* bertepatan dengan definisi di atas jika $D = T(F)$ (himpunan suku dasar yang dibangun di atas F).

Artinya, untuk setiap interpretasi $I = (D, R, G)$ dari $L = (P, F, X, Op)$, jika $D = T(F)$, maka $I(A(x), \{x \sim d\}) = I(A(d))$, yaitu, $eval(A(x), I, x \sim d) = eval(A(d), I, \emptyset)$, di mana $x \in X, d \in T(F)$, dan $A(d) = A(x)[x \sim d]$.

Kita akan mempertahankan konvensi bahwa himpunan S dari rumus menunjukkan konjungsi dari rumus-rumus yang muncul dalam himpunan tersebut. Jika setiap rumus benar dalam interpretasi yang sama, maka himpunan S dikatakan dapat dipenuhi, dan interpretasinya adalah modelnya.

Contoh 5.2.12 Misalkan $S = \{\forall x \exists y p(x, y), \forall x \neg p(x, x), \forall x \forall y \forall z (p(x, y) \wedge p(y, z) \rightarrow p(x, z))\}$. Rumus kedua menyatakan bahwa " p bersifat irefleksif"; rumus ketiga mengatakan " p bersifat transitif."

Perhatikan interpretasi $I(\{N\}, \{<\}, \emptyset)$; mudah untuk memeriksa bahwa I adalah model dari S , karena $<$ bersifat transitif dan irefleksif, yaitu $\neg(x < x)$, dan untuk setiap $x \in N$, terdapat $x + 1 \in N$ sedemikian sehingga $x < x + 1$.

Apakah S memiliki model terbatas, yaitu model di mana semesta terbatas? Jika S memiliki model terbatas, katakanlah $I = (D, \{r_p\}, \emptyset)$, di mana $|D| = n$, perhatikan graf berarah $G = (D, r_p)$. Jika ada jalur dari a ke b di G , maka harus ada sisi dari a ke b , karena r_p bersifat transitif. Kita tidak dapat memiliki sisi dari sebuah simpul ke dirinya sendiri karena r_p bersifat irefleksif.

Jadi, jika G memiliki siklus, maka ada jalur dari simpul v dalam siklus ke dirinya sendiri, yang mengarah ke sisi dari v ke dirinya sendiri berdasarkan transitivitas, sebuah kontradiksi.

Tanpa siklus, setiap jalur di $G = (D, r_p)$ memiliki ujung karena D terbatas. Untuk sebuah simpul x di ujung sebuah jalur, kita tidak dapat menemukan simpul y lain yang memenuhi $\forall x \exists y p(x, y)$. Oleh karena itu, S tidak dapat memiliki model terbatas.

Dalam Bab 2, diberikan himpunan P variabel proposisional, kita menggunakan All_P untuk menunjukkan semua interpretasi proposisional yang melibatkan P . Demikian pula, dalam logika orde pertama, diberikan himpunan P simbol predikat dan himpunan F simbol fungsi, misalkan All_P, F menunjukkan semua interpretasi yang melibatkan P dan F . Maka kita memiliki salinan Teorema 2.2.11 untuk logika orde pertama, yang menunjukkan hubungan erat antara logika dan teori himpunan.

Teorema 5.2.13 Untuk setiap rumus tertutup A dan B dari $L = (P, F, X, Op)$,

- (a) $M(A \vee B) = M(A) \cup M(B)$
- (b) $M(A \wedge B) = M(A) \cap M(B)$
- (c) $M(\neg A) = All_{P,F} - M(A)$

Pembuktian teorema di atas mengikuti pendekatan yang sama dengan Teorema 2.2.11.

Kesetaraan dan Keterkaitan

Dua definisi berikut disalin dari Definisi 2.2.13 dan 2.2.17.

Definisi 5.2.14 Diberikan dua rumus A dan B , A dan B secara logis setara jika $M(A) = M(B)$, yang dilambangkan dengan $A \equiv B$.

Definisi 5.2.15 Diberikan dua rumus A dan B , kita katakan A mengimplikasikan B , atau B adalah konsekuensi logis dari A , yang dilambangkan dengan $A \models B$, jika $M(A) \subseteq M(B)$.

Dengan asumsi A adalah , kita dapat dengan mudah menulis B untuk menunjukkan bahwa “ B valid.” Dengan definisi yang sama, kita memiliki hasil yang sama untuk logika orde pertama.

Teorema 5.2.16 Untuk setiap dua rumus A dan B , (a) $A \models B$ jika dan hanya jika $A \rightarrow B$ valid; dan (b) $A \equiv B$ jika dan hanya jika $A \leftrightarrow B$ valid.

Teorema substitusi dari logika proposisional masih berlaku dalam logika orde pertama.

Teorema 5.2.17 Untuk setiap rumus A, B , dan C , di mana B adalah subrumus dari A , dan $B \equiv C$, maka $A \equiv A[B \sim C]$.

Proposisi 5.2.18 Untuk setiap rumus $A(x)$, di mana x adalah variabel bebas dari A , variabel y tidak muncul dalam $A(x)$, dan $A(y)$ menunjukkan $A(x)[x \sim y]$, maka $\forall x A(x) \equiv \forall y A(y)$ dan $\exists x A(x) \equiv \exists y A(y)$.

Bukti Untuk setiap interpretasi I ,

$$I(\forall x A(x)) = \bigwedge_{d \in D} I(A(x), \{x \sim d\}) = \bigwedge_{d \in D} I(A(d))$$

$$I(\forall y A(y)) = \bigwedge_{d \in D} I(A(y), \{y \sim d\}) = \bigwedge_{d \in D} I(A(d))$$

Variabel x berfungsi sebagai placeholder dalam $A(x)$, sama seperti y berfungsi sebagai placeholder yang sama dalam $A(y)$, untuk memberi tahu *eval* kapan harus mengganti x atau y dengan d . Nama yang berbeda dapat digunakan untuk placeholder ini, sehingga $I(\forall x A(x), \theta) = I(\forall y A(y), \theta)$. Karena I bersifat sembarang, maka harus berlaku bahwa $\forall x A(x) \equiv \forall y A(y)$. Bukti untuk $\exists x A(x) \equiv \exists y A(y)$ identik.

Proposisi di atas memungkinkan kita untuk mengubah nama variabel terbatas dengan aman, sehingga variabel yang berbeda digunakan untuk setiap kemunculan kuantifikasi. Misalnya, lebih baik mengganti $\forall x (p(x) \vee \exists x q(x, y))$ dengan $\forall x (p(x) \vee \exists z q(z, y))$, untuk meningkatkan keterbacaan.

Proposisi 5.2.19 Misalkan $A(x)$, $B(x)$, dan $C(x, y)$ adalah rumus dengan variabel bebas x dan y .

1. $\neg \forall x A(x) \equiv \exists x \neg A(x)$
2. $\neg \exists x A(x) \equiv \forall x \neg A(x)$
3. $\forall x (A(x) \wedge B(x)) \equiv (\forall x A(x)) \wedge (\forall x B(x))$
4. $\exists x (A(x) \vee B(x)) \equiv (\exists x A(x)) \vee (\exists x B(x))$
5. $(\forall x A(x)) \vee B \equiv \forall x (A(x) \vee B)$ if x is not free in B
6. $(\exists x A(x)) \wedge B \equiv \exists x (A(x) \wedge B)$ if x is not free in B
7. $\forall x \forall y C(x, y) \equiv \forall y \forall x C(x, y)$
8. $\exists x \exists y C(x, y) \equiv \exists y \exists x C(x, y)$

Bukti Kami memberikan bukti untuk 1, 3, 5, dan 7, dan sisanya kami serahkan sebagai latihan.

1. Pertimbangkan interpretasi apa pun I , menggunakan hukum de Morgan, $\neg(A \wedge B) \equiv \neg A \vee \neg B$,

$$\begin{aligned} & I(\neg \forall x A(x)) \\ &= \neg I(\forall x A(x)) \\ &= \neg \bigwedge_{d \in D^I} I(A(x), \{x \sim d\}) \\ &= \bigvee_{d \in D^I} \neg I(A(x), \{x \sim d\}) \\ &= \bigvee_{d \in D^I} I(\neg A(x), \{x \sim d\}) \\ &= I(\exists x \neg A(x)). \end{aligned}$$

Karena I bersifat sembarang, kesetaraan berlaku.

3. Pertimbangkan interpretasi apa pun $I = (D, R, G)$, menggunakan sifat komutatif dan asosiatif dari $\wedge$,

$$\begin{aligned} & I(\forall x (A(x) \wedge B(x))) \\ &= \bigwedge_{d \in D} I(A(x) \wedge B(x), \{x \sim d\}) \\ &= \bigwedge_{d \in D} (I(A(x), \{x \sim d\}) \wedge I(B(x), \{x \sim d\})) \end{aligned}$$

$$\begin{aligned}
&= (\wedge_{d \in D} I(A(x), \{x \sim d\})) \wedge (\wedge_{d \in D} I(B(x), \{x \sim d\})) \\
&= I(\forall x A(x)) \wedge I(\forall x B(x)) \\
&= I((\forall x A(x)) \wedge (\forall x B(x)))
\end{aligned}$$

Karena I bersifat sembarang, kesetaraan berlaku.

5. Dalam bukti berikut, kita mengasumsikan bahwa x tidak muncul di B . Pertimbangkan interpretasi apa pun $I = (D, R, G)$, menggunakan hukum distribusi $\vee$ atas $\wedge$,

$$\begin{aligned}
&I((\forall x A(x)) \vee B) \\
&= I(\forall x A(x)) \vee I(B) \\
&= (\wedge_{d \in D} I(A(x), \{x \sim d\})) \vee I(B) \\
&= \wedge_{d \in D} (I(A(x), \{x \sim d\}) \vee I(B)) \\
&= \wedge_{d \in D} (I(A(x), \{x \sim d\}) \vee I(B, \{x \sim d\})) \\
&= \wedge_{d \in D} I(A(x) \vee B, \{x \sim d\}) \\
&= I(\forall x (A(x) \vee B))
\end{aligned}$$

Karena I bersifat sembarang, kesetaraan berlaku.

7. Pertimbangkan interpretasi apa pun $I = (D, R, G)$, menggunakan sifat komutatif dan asosiatif dari $\wedge$,

$$\begin{aligned}
&I(\forall x \forall y C(x, y)) \\
&= \wedge_{d \in D} I(\forall y C(x, y), \{x \sim d\}) \\
&= \wedge_{d \in D} (\wedge_{e \in D} I(C(x, y), \{x \sim d, y \sim e\})) \\
&= \wedge_{d \in D} (\wedge_{e \in D} I(C(x, y), \{x \sim d, y \sim e\})) \\
&= \wedge_{d \in D} I(\forall y C(x, y), \{x \sim d\}) \\
&= I(\forall y \forall x C(x, y))
\end{aligned}$$

Karena I bersifat sembarang, kesetaraan berlaku.

Contoh 5.2.20 Mari kita buktikan bahwa $(\exists x p(x)) \rightarrow \forall y q(y)$ dan $\forall x (p(x) \rightarrow q(x))$ tidak setara. Misalkan $I = (\{a, b\}, \{r_p, r_q\}, \emptyset)$, di mana $r_p(a) = r_q(a) = 1$ dan $r_p(b) = r_q(b) = 0$. Maka $I(\exists x p(x)) = 1$ dan $I(\forall y q(y)) = 0$. Jadi, $I((\exists x p(x)) \rightarrow \forall y q(y)) = 0$. Di sisi lain, $I(\forall x (p(x) \rightarrow q(x))) = 1$ karena $r_p = r_q$, sehingga $I(p(x) \rightarrow q(x), \{x \sim d\}) = 1$. Oleh karena itu, $(\exists x p(x)) \rightarrow (\forall y q(y))$ dan $\forall x (p(x) \rightarrow q(x))$ tidak ekuivalen.

Contoh di atas adalah bukti tipikal bahwa kedua rumus tersebut tidak ekuivalen: Kami memberikan interpretasi di mana salah satunya benar dan yang lainnya salah.

Contoh 5.2.21 Perhatikan $\exists x \forall y p(x, y)$ dan $\forall y \exists x p(x, y)$. Jika $p(x, y)$ berarti “ x mencintai y ,” maka yang pertama berarti “ada seseorang yang mencintai semua orang,”

tetapi yang kedua berarti “semua orang dicintai oleh seseorang.” Jelas, keduanya tidak setara. Untuk menunjukkan

$$\exists x \forall y p(x, y) \models \forall y \exists x p(x, y)$$

Misalkan $I = (D, R, G)$ adalah model apa pun dari $\exists x \forall y p(x, y)$. Terdapat $d \in D$ sedemikian sehingga $I(\forall y p(x, y), \{x \sim d\}) = 1$. Maka $\forall y \exists x p(x, y)$ harus benar di I :

$$\begin{aligned} & I(\forall y \exists x p(x, y), \emptyset) \\ &= \bigwedge_{d \in D} I(\exists x p(x, y), \{y \sim d\}) \\ &= \bigwedge_{d \in D} I(p(x, y), \{y \sim d, x \sim d\}) \\ &= I(\forall y p(x, y), \{x \sim d\}) \\ &= 1 \end{aligned}$$

Untuk menunjukkan bahwa $\forall y \exists x p(x, y)$ tidak menyiratkan $\exists x \forall y p(x, y)$, kita perlu menemukan interpretasi di mana $\forall y \exists x p(x, y)$ benar tetapi $\exists x \forall y p(x, y)$ salah. Detailnya diserahkan sebagai latihan.

Contoh 5.2.22 Kita dapat menggunakan relasi ekuivalensi yang diketahui untuk menunjukkan $\exists x (A(x) \rightarrow B(x)) \equiv (\forall x A(x)) \rightarrow (\exists x B(x))$.

$$\begin{aligned} & \exists x (A(x) \rightarrow B(x)) \\ \equiv & \exists x (\neg A(x) \vee B(x)) && // A \rightarrow B \equiv \neg A \vee B \\ \equiv & (\exists x \neg A(x)) \vee (\exists x B(x)) && // \text{Proposition 5.2.19(4)} \\ \equiv & \neg(\forall x A(x)) \vee (\exists x B(x)) && // \text{Proposition 5.2.19(1)} \\ \equiv & (\forall x A(x)) \rightarrow (\exists x B(x)) && // A \rightarrow B \equiv \neg A \vee B \end{aligned}$$

Semakin banyak kesetaraan yang ditetapkan, semakin banyak kesempatan kita dapat menerapkan metode ini.

Contoh 5.2.23 Paradoks peminum dapat dinyatakan sebagai “Ada seseorang di pub sedemikian rupa sehingga, jika dia minum, maka semua orang di pub juga minum.” Paradoks ini dipopulerkan oleh ahli logika Raymond Smullyan, yang menyebutnya sebagai “prinsip minum” dalam bukunya tahun 1978, *What Is the Name of This Book*.

Misalkan $p(x)$ menyatakan “ x sedang minum di pub”; paradoks peminum dapat diwakili oleh rumus

$$\exists x (p(x) \rightarrow \forall y p(y))$$

yang dapat dikonversi menjadi rumus setara berikut:

$$A : \exists x \neg p(x) \vee \forall y p(y)$$

Rumus ini valid karena untuk interpretasi I apa pun, $\forall y p(y)$ bernilai benar atau salah di I . Jika bernilai benar, maka A bernilai benar di I ; jika bernilai salah, maka terdapat elemen a dari I sedemikian sehingga $p(a)$ bernilai salah, sehingga $\exists x \neg p(x)$ bernilai benar di I . Perhatikan bahwa jika domain I kosong, $\forall y p(y)$ bernilai benar secara trivial di I .

Misalkan $p(x)$ menyatakan “ x kaya”; dari A , kita memperoleh pernyataan valid berikut dalam logika orde pertama: “Ada seseorang di dunia sedemikian sehingga, jika dia kaya, maka semua orang di dunia kaya.” Tampaknya jika kita telah menemukan orang ini, dunia akan terbebas dari kemiskinan, meskipun dalam logika, setiap orang miskin dapat menjadi orang ini. Sifat paradoks dari pernyataan ini berasal dari fakta bahwa pernyataan “jika maka” sering kali mewakili sebab akibat dalam bahasa alami. Namun, dalam logika orde pertama, $A \rightarrow B$ benar jika A salah dan tidak ada hubungan sebab-akibat antara A dan B .

Pernyataan paradoks peminum mungkin salah ditentukan oleh rumus tersebut.

$$(\exists x p(x)) \rightarrow (\forall y p(y))$$

yang setara dengan

$$B : (\forall x \neg p(x)) \vee (\forall y p(y))$$

B hanya benar ketika “tidak ada yang minum” atau “semua orang minum.” Pencampuran A dan B mungkin merupakan alasan lain mengapa paradoks ini membingungkan.

Konstruksi Himpunan dan Logika Banyak-Sorotan

Dalam matematika dan ilmu komputer, umum untuk mendefinisikan himpunan dengan predikat unari (atau monadik). Misalnya, himpunan S_1 bilangan asli antara 5 dan 10 dapat didefinisikan sebagai

$$S_1 = \{x \mid x \in N \wedge x \geq 5 \wedge x \leq 10\}$$

di mana N menunjukkan himpunan semua bilangan asli. Misalkan $A(x)$ adalah “ $x \in N \wedge x \geq 5 \wedge x \leq 10$ ”. Jelas, $A(x)$ adalah predikat monadik (dengan variabel bebas x). Misalkan $B(x)$ adalah “ $x \in N \wedge \exists y (y \in N \wedge x = y * y)$ ”. Maka $S_2 = \{x \mid B(x)\}$ adalah himpunan semua bilangan kuadrat.

Penggunaan predikat monadik dalam konstruksi suatu himpunan sangat menarik karena (1) rumusnya harus memiliki satu variabel bebas dan (2) konstruksinya menggunakan interpretasi yang dimaksudkan dari rumus tersebut. Untuk dua contoh di atas, interpretasi untuk $A(x)$ dan $B(x)$ adalah sama: Domainnya adalah N dengan relasi yang diketahui (misalnya, $\geq$) dan fungsi (misalnya, $*$). S_1 dan S_2 adalah himpunan $x \in N$ sedemikian sehingga $A(x)$ dan $B(x)$ diinterpretasikan sebagai benar, masing-masing. Secara umum, jika N dipahami dari konteksnya, “ $x \in N$ ” dapat dihilangkan dari $A(x)$ dan $B(x)$.

Definisi Ekstensional dan Intensional

Pembentukan himpunan menggunakan predikat dapat membantu kita memahami definisi ekstensional dan intensional dalam logika filosofis. Definisi intensional memberikan makna pada suatu istilah dengan menentukan kondisi yang diperlukan dan cukup untuk kapan istilah tersebut harus digunakan. Dalam kasus kata benda, ini setara dengan menentukan kondisi yang harus dimiliki suatu objek agar dapat dianggap sebagai referen dari istilah tersebut. Definisi ekstensional memberikan makna pada suatu istilah dengan menentukan ekstensinya (yaitu, domainnya). Artinya, setiap objek yang termasuk dalam definisi istilah yang dimaksud termasuk dalam domain ini. Misalnya, misalkan $p(x)$ menyatakan bahwa “ x adalah bilangan asli yang lebih besar dari 1 dan satu-satunya pembaginya adalah 1 dan dirinya sendiri.” Maka $p(x)$ adalah definisi intensional dari bilangan prima dan $P = \{2, 3, 5, 7, 11, \dots\}$ adalah definisi ekstensionalnya.

Definisi intensional dan ekstensional adalah dua cara penting di mana objek atau konsep didefinisikan secara formal. Dalam konstruksi himpunan dengan predikat, kita melihat kedua definisi dengan jelas: Diberikan $S = \{x \mid A(x), A(x)\}$ adalah definisi intensional dan menentukan kondisi yang diperlukan dan cukup dari himpunan tersebut; S adalah definisi ekstensial yang merupakan kumpulan semua objek yang memenuhi kondisi yang dimaksud.

Contoh 5.2.24 Misalkan $r(x)$ adalah “ x adalah reptil,” $s(x)$ adalah “ x memiliki tubuh yang terbungkus cangkang,” dan $t(x)$ adalah “ x adalah kura-kura.” Maka himpunan T dari semua kura-kura dapat didefinisikan sebagai $T = x \mid r(x) \wedge s(x)$. Misalkan $t(x)$ adalah $r(x) \wedge s(x)$, maka $t(x)$ adalah definisi intensional dari kura-kura dan T adalah definisi ekstensial dari kura-kura. Demikian pula, kita dapat mendefinisikan himpunan R dari semua reptil tanpa menggunakan $r(x)$ untuk menghindari definisi diri. Jelas, $T \subseteq R$, dan $\forall x(t(x) \rightarrow r(x))$ valid.

Secara umum, kita memiliki hasil berikut, yang pembuktiannya diserahkan sebagai latihan.

Teorema 5.2.25 Diberikan dua rumus $A(x)$ dan $B(x)$, di mana x adalah satu-satunya variabel bebas, misalkan $S_A = \{x \mid A(x)\}$ dan $S_B = \{x \mid B(x)\}$.

- (a) $(S_A \cup S_B) = \{x \mid A(x) \vee B(x)\}$
- (b) $(S_A \cap S_B) = \{x \mid A(x) \wedge B(x)\}$
- (c) $(S_A \subseteq S_B) \equiv \forall x (A(x) \rightarrow B(x))$

Kita katakan $A(x)$ lebih kuat daripada $B(x)$ jika $\forall x(A(x) \rightarrow B(x))$ benar. Jelas, rumus terkuat adalah dan yang terlemah adalah T . Dari (c) teorema di atas, jelas bahwa semakin kuat kondisi dalam definisi intensional, semakin kecil himpunan ekstensi dan sebaliknya. Hubungan implikasi di antara predikat uner ini menginduksi suatu urutan, yaitu, hubungan himpunan bagian, atas himpunan yang didefinisikan oleh predikat-predikat ini. Logika yang disebut logika terurut memperhatikan urutan ini atas himpunan-himpunan ini.

Kami mengomentari bahwa Teorema 5.2.13 menunjukkan hubungan erat antara logika dan teori himpunan. Teorema di atas memperkuat hubungan ini. Memahami hubungan ini membantu kita untuk menghindari kesalahan logika. Misalnya, Contoh 5.2.24, jika kita tahu c adalah kura-kura, kita menyimpulkan bahwa c adalah reptil. Ini dapat dibuktikan dengan dua cara: (1) $c \in T$ dan $T \subseteq R$ menyiratkan $c \in R$; dan (2) $t(c)$ dan $\forall x (t(x) \rightarrow r(x))$ menyiratkan $r(c)$.

Logika Banyak-Solusi dan Pengetikan Kuat

Logika banyak-solusi adalah perluasan logika orde pertama dengan memperkenalkan tipe (atau sorti) ke dalam logika, di mana setiap tipe dipandang sebagai himpunan bagian dari semesta. Bahasa pemrograman saat ini menggunakan logika banyak-solusi untuk memodelkan tipe data. Arti dari suatu tipe hanyalah himpunan objek yang memenuhi predikat unari atau monodik. Misalnya, arti dari tipe *int* bilangan bulat adalah himpunan

$$int = \{x \mid P_{int}(x)\}$$

di mana $P_{int}(x)$ bernilai benar jika dan hanya jika x adalah bilangan bulat. Logika banyak-solusi mengasumsikan himpunan universal U dan setiap tipe adalah himpunan bagian dari U . Mungkin ada relasi antar tipe. Misalnya, tipe *nat* bilangan asli adalah subtype dari *int*, yang artinya adalah

$$nat = \{x \mid x \in int, x \geq 0\}$$

Dengan kata lain, $nat \subseteq int$. Logika multisort berkaitan dengan hierarki tipe dan merupakan salah satu pendekatan yang diciptakan untuk menghindari paradoks dalam teori himpunan naif (lihat Bagian 1.3).

Beberapa bahasa pemrograman modern menggunakan sistem tipe yang kuat, di mana tipe digunakan oleh kompiler sebagai "nilai" untuk pemeriksaan tipe, sebuah ide yang berasal dari logika tingkat tinggi. Kami mengilustrasikan ide pengetikan yang kuat dengan sebuah contoh. Pada Bab 1, kami memperkenalkan bentuk *Backus – Naur* dengan definisi bilangan asli berikut:

$$(Num) ::= 0 \mid suc((Num))$$

di mana *suc* menunjukkan fungsi penerus. Bahasa pemrograman dengan sistem tipe yang kuat dapat menerima definisi di atas untuk mendefinisikan tipe baru *Num*. Selain itu, ia dapat membuat banyak tipe baru secara dinamis. Misalnya, jika *Num* menunjukkan himpunan bilangan asli, maka *suc(Num)* adalah tipe bilangan bulat positif, *suc(suc(Num))* adalah tipe bilangan asli lebih besar dari 1, dan seterusnya. Jika suatu fungsi membutuhkan bilangan bulat positif x (sebagai pembagi), kita dapat mendeklarasikan tipe x sebagai *suc(Num)*, tanpa secara eksplisit menambahkan $x > 0$ dalam kode. Kompilator akan menambahkan kondisi

$x > 0$ secara otomatis sebelum pemanggilan fungsi. Cara penggunaan tipe ini dapat menghasilkan kode yang rapi dan menghindari kesalahan manusia.

Sistem tipe yang kuat adalah fitur umum dari bahasa pemrograman fungsional berdasarkan kalkulus lambda bertipe (akan dibahas selanjutnya). Ketika kita menulis kode kita dalam bahasa pemrograman fungsional seperti Haskell, kita dapat memanfaatkan sistem tipe yang kuat yang membantu kita menyaring beberapa kesalahan logika terlebih dahulu.

Logika Orde Pertama Versus Logika Orde Tinggi

Tampaknya logika orde pertama merupakan objek studi yang alami dan bermakna. Dari bab ini hingga akhir buku ini, tujuh dari delapan bab sepenuhnya dikhususkan untuk studi logika orde pertama. Bahkan untuk satu-satunya bab yang tidak membahasnya, logika orde pertama sangat diperlukan. Artinya, logika orde pertama menempati dua pertiga ruang buku ini, karena logika orde pertama telah lama dianggap sebagai logika yang "tepat" untuk mempelajari matematika dan ilmu komputer.

Logika Orde Tinggi

Orang mungkin bertanya-tanya arti "orde pertama" dan bertanya apa itu "orde kedua," "orde ketiga," dan seterusnya. Variabel logika orde pertama mengambil individu dari himpunan yang disebut semesta. Jika kita mengizinkan variabel untuk mengambil himpunan individu, itu disebut logika orde kedua. Logika orde ketiga akan memungkinkan variabel untuk mengambil himpunan dari himpunan dan seterusnya. Logika orde tinggi sering disebut sebagai logika orde ke- n untuk $n \geq 1$ dan memungkinkan kuantifikasi variabel atas himpunan yang bersarang sedalam mungkin.

Logika tingkat tinggi lebih ekspresif daripada logika tingkat pertama, tetapi sifat-sifat model-teoretisnya kurang baik dibandingkan dengan logika tingkat pertama. Pada Bab 11, kita akan menunjukkan bahwa beberapa masalah logika tingkat pertama tidak dapat dihitung. Lebih banyak masalah menjadi tidak dapat dihitung dalam logika tingkat tinggi.

Definisi lain dari "logika tingkat tinggi" adalah bahwa variabel dapat mengambil fungsi sembarang di seluruh semesta, bukan hanya individu-individu di semesta tersebut. Contoh terkenal dari sistem logika formal semacam ini adalah kalkulus lambda, yang diperkenalkan oleh Alonzo Church pada tahun 1930-an. Kalkulus lambda menggunakan abstraksi fungsi, pengikatan variabel, dan substitusi untuk komputasi. Ini adalah dasar dari teori tipe dan pemrograman fungsional. Informasi lebih lanjut terkait kalkulus lambda dapat ditemukan di Bab 9 dan 11. Pengenalan rinci tentang kalkulus lambda berada di luar cakupan buku ini, meskipun memiliki dampak besar pada banyak bidang ilmu komputer.

Terkadang orang menggunakan logika tingkat tinggi untuk menunjukkan gabungan logika tingkat kedua, tingkat ketiga, ..., tingkat ke- n , ..., sebagai kontras dengan logika tingkat pertama. Karena logika tingkat ketiga atau tingkat keempat jarang dibutuhkan, logika tingkat tinggi mungkin hanyalah logika tingkat kedua karena keuntungan dan kerugian yang muncul seiring dengan melampaui logika tingkat pertama telah muncul dalam logika tingkat kedua.

Status "Istimewa" Logika Tingkat Pertama

Menurut William Ewald, sejarah logika tingkat pertama sama sekali tidak sederhana, dan tentu saja bukan masalah penemuan tiba-tiba oleh seorang peneliti tunggal. Logika tingkat

pertama dipelajari oleh para ahli logika seperti Gottlob Frege dan secara eksplisit diidentifikasi oleh Charles Peirce pada tahun 1885, tetapi kemudian dilupakan. Logika orde pertama ditemukan kembali secara independen oleh David Hilbert pada tahun 1917. Artinya, Peirce adalah orang pertama yang mengidentifikasinya, tetapi Hilbert-lah yang mempopulerkannya, seperti halnya "Viking menemukan Amerika, tetapi mereka melupakannya, karena mereka belum membutuhkannya."

Hilbert mengisolasi logika orde pertama dengan mengajukan pertanyaan tentang kelengkapan, konsistensi, dan keterputusan untuk sistem logika. Meskipun pertanyaan-pertanyaan ini mewakili lompatan konseptual yang sangat besar dalam logika, Hilbert tidak memperlakukan logika orde pertama sebagai lebih signifikan daripada logika orde tinggi. Gödel dan yang lainnya yang mengakui logika orde pertama sebagai sesuatu yang sangat berbeda dari logika orde tinggi pada awal tahun 1930-an. Bahkan setelah hasil Gödel dipahami secara luas, para ahli logika terus bekerja dalam logika orde tinggi, dan butuh bertahun-tahun sebelum logika orde pertama mencapai status "istimewa".

Kemunculan logika orde pertama terkait erat dengan penemuan-penemuan teknis, dengan konsepsi yang berbeda tentang apa yang membentuk logika, dengan program penelitian matematika yang berbeda, dan dengan refleksi filosofis dan konseptual. Ada alasan mengapa logika orde pertama dianggap sebagai sistem logika yang "istimewa"—yaitu, sebagai logika yang "tepat" untuk penyelidikan dalam dasar-dasar matematika.

Secara umum, logika orde pertama lebih ekspresif daripada logika proposisional dan kurang rumit daripada logika orde tinggi. Tampaknya pemahaman kita tentang kuantifikasi atas objek lebih kuat daripada pemahaman kita tentang kuantifikasi atas properti, bahkan ketika semesta terbatas. Misalnya, memecahkan teka-teki Sudoku adalah masalah kuantifikasi orde pertama (lihat Bagian 4.3).

Pertanyaan tentang konfigurasi awal Sudoku, seperti "dapatkah ia menerima satu dan hanya satu solusi?", "berapa tingkat kesulitannya?", "berapa batas jumlah sel yang diisi pada awalnya?", dll., adalah masalah kuantifikasi orde kedua. Ini adalah masalah yang sulit bahkan bagi para ahli teka-teki Sudoku.

Setiap semesta logika orde pertama diasumsikan dapat dihitung. Namun, kuantifikasi atas semua himpunan bagian dari himpunan tak hingga terhitung memerlukan kuantifikasi atas himpunan tak terhitung (Proposisi 1.3.7).

Dalam Bab 11, kita akan menunjukkan bahwa himpunan tak terhitung tidak dapat dihitung (Proposisi 11.4.8). Dengan kata lain, kuantifikasi dalam logika tingkat tinggi pada umumnya tidak dapat dihitung ketika alam semesta tak terbatas.

Tampaknya status "istimewa" logika tingkat pertama juga berasal dari pertimbangan filosofis: kebutuhan untuk menghindari paradoks teori himpunan, pencarian landasan yang aman untuk matematika, dan perasaan bahwa logika tingkat tinggi secara metodologis diragukan dan dapat dihindari. Semua hal ini menunjukkan pengaruh berkelanjutan dari "Krisis Landasan," *Grundlagenkrise* dalam bahasa Jerman, pada tahun 1920-an, yang sangat berperan dalam menetapkan syarat-syarat pemahaman filosofis selanjutnya tentang landasan matematika.

Teori Tipe Versus Logika Tingkat Tinggi

Teori tipe diciptakan oleh Alonzo Church pada tahun 1940, disebut teori tipe sederhana, dan terdapat banyak generalisasi dari teori tipe sederhana. Teori tipe Church didasarkan pada kalkulus lambda dan notasi lambda adalah satu-satunya mekanisme pengikatan yang digunakan dalam teori tipe sederhana. Karena properti dan relasi dapat dianggap sebagai fungsi dari semesta ke nilai kebenaran, konsep fungsi dianggap sebagai primitif dalam teori tipe Church, dan notasi lambda dimasukkan ke dalam bahasa formal teori tipe. Selain itu, kuantifikasi dan operator deskripsi diperkenalkan sedemikian rupa sehingga mekanisme pengikatan tambahan dapat dihindari, dan notasi lambda digunakan kembali sebagai gantinya. Saat ini teori tipe biasanya berarti studi tentang sistem berdasarkan kalkulus lambda bertipe.

Dalam logika matematika, teori tipe merupakan sinonim dari logika tingkat tinggi, karena memungkinkan kuantifikasi tidak hanya pada elemen semesta (seperti pada logika tingkat pertama) tetapi juga pada fungsi, predikat, dan bahkan variabel tingkat tinggi. Teori tipe menetapkan tipe pada entitas, membedakan, misalnya, antara bilangan, himpunan bilangan, fungsi dari bilangan ke himpunan bilangan, dan himpunan fungsi tersebut. Perbedaan ini memungkinkan seseorang untuk membahas dunia himpunan dan fungsi yang kaya secara konseptual tanpa menghadapi paradoks teori himpunan naif. Oleh karena itu, teori tipe dan logika tingkat tinggi setara dan keduanya dapat dilihat sebagai gabungan dari keluarga logika tingkat terbatas.

Teori tipe adalah bahasa logika formal termasuk orde pertama, tetapi lebih ekspresif dalam arti praktis. Teori tipe merupakan bahasa formal yang sangat baik untuk merepresentasikan pengetahuan dalam sistem informasi otomatis, sistem penalaran otomatis yang canggih, sistem untuk memverifikasi kebenaran bukti matematika, dan berbagai proyek yang melibatkan logika dan kecerdasan buatan. Teori tipe juga memainkan peran penting dalam studi semantik formal bahasa alami. Terlepas dari kompleksitas algoritma yang terkait dengan teori tipe yang tinggi, terdapat implementasi yang sukses dan berpengaruh berdasarkan teori tipe, termasuk ML, Haskell, Coq, keluarga HOL, dan Twelf.

5.3 METODE PEMBUKTIAN

Metode pembuktian logika orde pertama memiliki tiga tugas yang sama seperti pada logika proposisional: membuktikan bahwa suatu formula A adalah (a) valid, (b) tidak dapat dipenuhi, atau (c) dapat dipenuhi. (a) dan (b) setara karena A valid jika dan hanya jika $\neg A$ tidak dapat dipenuhi. (c) berbeda karena baik A maupun $\neg A$ dapat dipenuhi secara bersamaan. Seperti yang ditunjukkan pada Contoh 5.2.22, kita dapat menggunakan relasi kesetaraan untuk mengubah A menjadi T atau $\perp$, sehingga menunjukkan bahwa A valid atau tidak dapat dipenuhi. Namun, pendekatan ini hanya memiliki peluang keberhasilan yang sangat terbatas.

Beberapa metode pembuktian seperti yang didasarkan pada tabel kebenaran atau ROBDD tidak berfungsi untuk logika orde pertama. Namun, banyak metode pembuktian dapat dengan mudah diperluas dari logika proposisional ke logika orde pertama, meskipun yang terakhir jauh lebih ekspresif daripada yang pertama. Itu berarti logika orde pertama memiliki

metode pembuktian yang mudah dipahami secara teoritis atau diimplementasikan dengan benar pada komputer. Sebagai contoh, aturan modus ponens yang terkenal dapat ditentukan sebagai berikut:

Definisi 5.3.1 Aturan modus ponens dari logika orde pertama adalah aturan inferensi berikut:

$$(modus\ ponens) \quad \frac{\forall x (A(x) \rightarrow B(x)) \quad A(t)}{B(t)} \quad t \text{ is any term}$$

Misalnya, misalkan $A(x)$ adalah “ x adalah manusia,” $B(x)$ adalah “ x adalah fana,” t adalah konstanta “Adam,” maka “Adam adalah fana” dapat disimpulkan dari “Setiap manusia adalah fana” dan “Adam adalah manusia.”

Pada Bab 3, kita telah memperkenalkan berbagai sistem pembuktian untuk logika proposisional: tabel semantik, deduksi alamiah, resolusi, dll. Semua sistem ini dapat diperluas ke logika orde pertama. Pada bagian ini, kita akan secara singkat memperkenalkan perluasan tabel semantik dan deduksi alamiah ke logika orde pertama dan meninggalkan resolusi ke bab berikutnya.

Tabel Semantik

Tabel semantik seperti yang diberikan pada Bab 3 mengubah formula proposisional menjadi DNF dengan serangkaian aturan α (yang menangani konjungsi dan menghasilkan satu anak dalam tabel) dan aturan β (yang menangani disjungsi dan menghasilkan dua anak dalam tabel). Aturan-aturan ini akan diwarisi oleh logika orde pertama. Perlu diingat bahwa aturan tersebut berlaku untuk operator logika teratas dalam sebuah rumus di simpul daun dari tabel.

α	$\alpha 1, \alpha 2$	β	$\beta 1$	$\beta 2$
$A \wedge B$	A, B	$\neg(A \wedge B)$	$\neg A$	$\neg B$
$\neg(A \vee B)$	$\neg A, \neg B$	$A \vee B$	A	B
$\neg(A \rightarrow B)$	$A, \neg B$	$A \rightarrow B$	$\neg A$	
$\neg(A \oplus B)$	$(A \vee \neg B), (\neg A \vee B)$	$A \oplus B$	$A, \neg B$	$\neg A, B$
$A \leftrightarrow B$	$(A \vee \neg B), (\neg A \vee B)$	$\neg(A \leftrightarrow B)$	$A, \neg B$	$\neg A, B$
$\neg(A \uparrow B)$	A, B	$A \uparrow B$	$\neg A$	$\neg B$
$A \downarrow B$	$\neg A, \neg B$	$\neg(A \downarrow B)$	A	B
$\neg\neg A$	A			

Untuk menangani kuantifikasi, kita memperkenalkan dua aturan: satu untuk kuantifikasi universal, yang disebut aturan- $\forall$, dan satu untuk kuantifikasi eksistensial, yang disebut aturan- $\exists$.

Definisi 5.3.2 Aturan-6 adalah aturan inferensi:

$$(\forall) = \frac{\forall x A(x)}{A(t)} \quad t \text{ is a ground term}$$

Aturan- $\forall$ bekerja sebagai berikut: Jika $\forall x A(x)$ muncul di simpul daun n , kita membuat simpul anak n' dari n , menyalin setiap rumus n ke n' , dan menambahkan $A(t)$ ke n' , di mana t adalah istilah dasar (tanpa variabel) yang muncul di n . Jika n' tidak memiliki istilah dasar, kita dapat memperkenalkan konstanta baru sebagai t .

Kami menekankan bahwa aturan- $\forall$ adalah aturan inferensi, bukan aturan penyederhanaan, karena aturan ini menciptakan dan menambahkan sesuatu, bukan mengganti $\forall x A(x)$ dengan sesuatu. Karena aturan- $\forall$, kita mungkin tidak dapat menunjukkan penghentian metode tabel semantik.

Definisi 5.3.3 Aturan- $\exists$ adalah aturan penyederhanaan:

$$(\exists) = \frac{\exists x A(x)}{A(c)} \quad c \text{ adalah Konstanta Baru}$$

Aturan- $\exists$ bekerja sebagai berikut: Jika $\exists x A(x)$ muncul di simpul daun n , kita membuat simpul anak nr dari n , menyalin setiap rumus n ke nr , dan mengganti $\exists x A(x)$ di nr dengan $A(c)$. Nanti setelah selesai membaca bab ini, Anda akan tahu bahwa konstanta baru ini disebut konstanta Skolem, untuk mengingat kontribusi Thoralf Skolem terhadap logika.

Jika rumus-rumus tersebut tidak dalam NNF (bentuk normal negasi), kita juga membutuhkan aturan- $\neg\exists$, yang juga merupakan aturan inferensi:

$$\neg(\exists) = \frac{\neg\exists x A(x)}{\neg A(t)} \quad t \text{ adalah Istilah Baru}$$

dan aturan $\neg\forall$, yang merupakan aturan penyederhanaan:

$$\neg(\forall) = \frac{\neg\forall x A(x)}{\neg A(c)} \quad c \text{ adalah Konstanta Baru}$$

Contoh 5.3.4 Untuk menunjukkan bahwa $(\exists x \forall y p(x, y)) \rightarrow (\forall y \exists x p(x, y))$ valid menurut tabel semantik, kita bekerja pada negasi dari rumus tersebut.

	:	$\neg((\exists x \forall y p(x, y)) \rightarrow (\forall y \exists x p(x, y)))$	$\alpha \neg \rightarrow$
1 :		$\exists x \forall y p(x, y), \neg \forall y \exists x p(x, y)$	$\exists$
11 :		$\forall y p(c_1, y), \neg \forall y \exists x p(x, y)$	$\neg \forall$
111 :		$\forall y p(c_1, y), \neg \exists x p(x, c_2)$	$\forall$
1111 :		$\forall y p(c_1, y), \neg \exists x p(x, c_2), p(c_1, c_2)$	$\neg \exists$
11111 :		$\forall y p(c_1, y), \neg \exists x p(x, c_2), p(c_1, c_2), \neg p(c_1, c_2)$	<i>closed</i>

Karena negasi dari rumus tersebut memiliki tabel tertutup, maka rumus tersebut valid.

Contoh 5.3.5 Misalkan A adalah $B \wedge C \wedge D$, di mana $B = (\forall x p(x, s(x)))$, $C = (\forall x \forall y (q(x) \vee q(y)))$, dan $D = (\neg \exists x q(x))$. Jika kita memilih untuk mengerjakan q terlebih dahulu, kita akan menemukan tabel tertutup untuk A ; dengan demikian, A tidak dapat dipenuhi. Selanjutnya, kita akan mulai dengan $\{B, C, D\}$ dengan mengabaikan penggunaan aturan $\alpha^{\wedge}$.

$\epsilon :$	$B, C, (\neg \exists x q(x))$	$\forall$
1 :	$B, C, (\neg \exists x q(x)), \forall y (q(c) \vee q(y)),$	$\forall$
11 :	$B, C, (\neg \exists x q(x)), \forall y (q(c) \vee q(y)), q(c) \vee q(c),$	$\neg \exists$
111 :	$B, C, (\neg \exists x q(x)), \forall y (q(c) \vee q(y)), q(c) \vee q(c), \neg q(c),$	$\beta \vee$
1111 :	$B, C, (\neg \exists x q(x)), \forall y (q(c) \vee q(y)), q(c), \neg q(c),$	<i>closed</i>
1112 :	$B, C, (\neg \exists x q(x)), \forall y (q(c) \vee q(y)), q(c), \neg q(c),$	<i>closed</i>

Jika kita memilih untuk mengerjakan p terlebih dahulu, maka tabel tersebut tidak akan pernah berakhir:

$\epsilon :$	$(\forall x p(x, s(x))), (\forall x \forall y (q(x) \vee q(y))), (\neg \exists x q(x))$	$\forall$
1 :	$(\forall x p(x, s(x))), \dots, p(c, s(c)),$	$\forall$
11 :	$(\forall x p(x, s(x))), \dots, p(c, s(c)), p(s(c), s(s(c)))$	$\forall$
111 :	$(\forall x p(x, s(x))), \dots, p(c, s(c)), p(s(c), s(s(c))), p(s(s(c)), s(s(s(c))))$	$\forall$
1 ... 1 :	...	

Satu pilihan mengarah pada pembuktian dan pilihan lainnya mengarah pada perulangan.

Contoh ini menunjukkan dengan jelas bahwa nondeterminisme dalam penerapan aturan (yaitu, memilih simpul daun, memilih aturan, atau memilih istilah dalam aturan) memang penting untuk mendapatkan tabel tertutup. Jika kita menggunakan strategi yang adil untuk penerapan aturan, maka tabel tertutup dijamin akan ditemukan ketika rumus masukan tidak dapat dipenuhi, seperti yang ditunjukkan oleh hasil berikut (tanpa bukti).

Teorema 5.3.6 Suatu rumus A tidak dapat dipenuhi jika dan hanya jika A memiliki tabel tertutup, yang dapat ditemukan dengan strategi yang adil.

Deduksi Alami

Dalam deduksi alami untuk logika proposisional, kita memiliki aturan pengenalan dan aturan eliminasi untuk setiap operator logika. Aturan-aturan ini akan diwariskan, ditambah aturan-aturan baru untuk kuantifikasi atas sekuen:

<i>op</i>	<i>Introduction</i>	<i>Elimination</i>
$\forall$	$(A(x) \mid \alpha) \vdash (\forall x A(x) \mid \alpha)$	$(\forall x A(x) \mid \alpha) \vdash (A(t) \mid \alpha)$
$\exists$	$(A(t) \mid \alpha) \vdash (\exists x A(x) \mid \alpha)$	$(\exists x A(x) \mid \alpha) \vdash (A(c) \mid \alpha)$

di mana x adalah variabel dan tidak muncul dalam α , t adalah suku, dan c adalah konstanta baru. Dari aturan-aturan ini, kita dapat melihat bahwa variabel bebas memiliki fungsi yang sama dengan variabel yang dikuantifikasi secara universal. Kita menggunakan dua contoh untuk mengilustrasikan aturan-aturan ini.

Contoh 5.3.7 Misalkan $A = \{\forall x p(x), \forall x q(x)\}$. Bukti $A \models \forall x (p(x) \wedge q(x))$ dalam deduksi alami diberikan di bawah ini

1.	$(\forall x p(x))$	assumed
2.	$(\forall x q(x))$	assumed
3.	$(p(x))$	$\forall_E$ from 1
4.	$(q(x))$	$\forall_E$ from 2
5.	$(p(x) \wedge q(x))$	$\wedge_I$ from 3, 4
6.	$(\forall x (p(x) \wedge q(x)))$	$\forall_I$ from 5

Berdasarkan Teorema 5.2.16, $(\forall x p(x)) \wedge (\forall x q(x)) \rightarrow \forall x (p(x) \wedge q(x))$ adalah valid.

Contoh 5.3.8 Tunjukkan bahwa $(\exists x (p(x) \vee q(x))) \models (\exists x p(x)) \vee (\exists x q(x))$ dengan deduksi alami.

1.	$(\exists x (p(x) \vee q(x)))$	assumed
2.	$(p(c) \vee q(c))$	$\exists_E$ from 1
3.	$(p(c) \mid q(c))$	$\vee_E$ from 2
4.	$((\exists x p(x)) \mid q(c))$	$\exists_I$ from 3
5.	$((\exists x p(x)) \mid (\exists x q(x)))$	$\exists_I$ from 4
6.	$((\exists x p(x)) \vee (\exists x q(x)))$	$\vee_I$ from 5

Berdasarkan Teorema 5.2.16, $(\exists x (p(x) \vee q(x))) \rightarrow ((\exists x p(x)) \vee (\exists x q(x)))$ adalah valid.

5.4 BENTUK NORMAL KONJUNGtif (CNF)

Kita baru saja memperkenalkan perluasan tabel semantik dan deduksi alami ke logika orde pertama dan mengilustrasikan penggunaan kedua metode ini untuk membuktikan teorema dalam logika orde pertama. Dalam praktiknya, kedua metode ini tidak seefektif metode resolusi, yang menjadi fokus bab berikutnya.

Ingatlah bahwa dalam logika proposisional, resolusi adalah metode refutasi dan rumus masukan harus dalam bentuk klausa. Hal ini masih berlaku untuk logika orde pertama. Pada bagian ini, kita akan membahas cara mengubah rumus orde pertama menjadi bentuk klausa.

Definisi 5.4.1 Dalam logika orde pertama, literal adalah atom (literal positif) atau negasi dari atom (literal negatif); klausa adalah disjungsi literal; Suatu formula

berada dalam bentuk normal konjugasi (CNF) jika merupakan konjugsi klausa.

Dari definisi tersebut, jika kita mengganti setiap atom dalam suatu klausa dengan variabel proposisional, kita memperoleh klausa dalam logika proposisional. Karena klausa hanya menggunakan tiga operator logika, yaitu $\neg$, $\wedge$, dan $\vee$, kita perlu menghilangkan $\rightarrow$, $\leftrightarrow$, $\oplus$, dan seterusnya, seperti yang kita lakukan dalam logika proposisional. Kita juga perlu menghilangkan kuantifikasi dan kita akan membahasnya sekarang.

Bentuk Normal Prenex

Definisi 5.4.2 Suatu formula A berada dalam *bentuk normal prenex (PNF)* jika

$$A = Q_1x_1Q_2x_2 \cdots Q_kx_kB(x_1, x_2, \dots, x_k),$$

di mana $Q_i \in \{\forall, \exists\}$ dan B tidak mengandung kuantifikasi apa pun.

Ide dari bentuk normal prenex adalah memindahkan semua kuantifikasi ke bagian atas rumus. Misalnya, $\forall x \forall y (p(x, y) \vee q(x))$ adalah PNF tetapi $\forall x ((\forall y p(x, y)) \vee q(x))$ bukan. Untuk mempermudah pembahasan, kita mengasumsikan bahwa rumus hanya berisi $\neg$, $\wedge$, $\vee$ ditambah dua kuantifikasi, sebagai operator. Kuantifikasi yang tidak berada di bagian atas harus memiliki operator lain tepat di atasnya: Dalam hal ini operator akan menjadi salah satu dari tiga operator: $\neg$, $\wedge$, atau $\vee$. Karena kuantifikasi dapat berada di posisi parameter $\wedge$ atau $\vee$ dan ada dua kuantifikasi, ada 10 kasus yang perlu dipertimbangkan. Jika kita menganggap $\wedge$ dan $\vee$ bersifat komutatif, kita membutuhkan enam aturan, yang diberikan dalam Proposisi 5.2.19:

Aturan Kesetaraan untuk PNF

1. $\neg \forall x A(x) \equiv \exists x \neg A(x)$
2. $\neg \exists x A(x) \equiv \forall x \neg A(x)$
3. $(\forall x A(x)) \wedge B \equiv \forall x (A(x) \wedge B)$ if x is not free in B
4. $(\forall x A(x)) \vee B \equiv \forall x (A(x) \vee B)$ if x is not free in B
5. $(\exists x A(x)) \wedge B \equiv \exists x (A(x) \wedge B)$ if x is not free in B
6. $(\exists x A(x)) \vee B \equiv \exists x (A(x) \vee B)$ if x is not free in B

di mana argumen dan dapat dipertukarkan.

Untuk memenuhi syarat bahwa “ x tidak bebas di B ,” pertama-tama perlu untuk mengubah nama semua variabel terkuantifikasi sehingga tidak ada variabel yang muncul lebih dari sekali setelah kuantifikasi.

Pada Bab 2, kita memperkenalkan konsep “bentuk normal negasi,” di mana negasi hanya muncul dalam literal. Dengan menggunakan dua aturan pertama di atas ditambah hukum de Morgan dan hukum negasi ganda, kita dapat mendorong negasi ke bawah, hingga

semua negasi muncul dalam literal. Rumus yang dihasilkan dikatakan berada dalam bentuk normal negasi (NNF), sebuah perluasan langsung dari logika proposisional.

Proposisi 5.4.3 Setiap rumus orde pertama dapat diubah menjadi rumus yang setara dalam NNF.

Contoh 5.4.4 Diberikan rumus berikut:

$$(\forall x \exists y p(x, y)) \wedge \neg(\forall x \forall y (q(x, y) \wedge q(y, x)))$$

kita dapat memilih untuk terlebih dahulu mengganti nama variabel sehingga setiap variabel terkuantifikasi memiliki nama yang berbeda:

$$(\forall x \exists y p(x, y)) \wedge \neg(\forall z \forall w (q(z, w) \wedge q(w, z)))$$

Dengan menerapkan aturan kesetaraan 1 dua kali,

$$(\forall x \exists y p(x, y)) \wedge (\exists z \exists w \neg(q(z, w) \wedge q(w, z)))$$

Sekarang kita dapat menerapkan hukum de Morgan untuk mendapatkan

$$(\forall x \exists y p(x, y)) \wedge (\exists z \exists w (\neg q(z, w) \vee \neg q(w, z)))$$

yang berada dalam NNF; langkah ini tidak diperlukan untuk mendapatkan PNF. Kita sekarang memiliki dua pilihan: Pilih aturan kesetaraan 3 atau aturan 5. Untuk menggunakan aturan 3, kita memiliki

$$\forall x ((\exists y p(x, y)) \wedge (\exists z \exists w (\neg q(z, w) \vee \neg q(w, z))))$$

Untuk menerapkan aturan 5 tiga kali, kita memiliki

$$\forall x \exists y \exists z \exists w (p(x, y) \wedge (\neg q(z, w) \vee \neg q(w, z))),$$

yang merupakan PNF.

Perhatikan bahwa urutan variabel terkuantifikasi dalam rumus ini adalah $xyzw$. Ada lima kemungkinan hasil lain di mana urutannya adalah $xzyw, xzwy, zxyw, zxwy$, atau $zwxy$, masing-masing. Jika kita diizinkan untuk menukar kuantifikasi dari tipe yang sama (dua yang terakhir dalam Proposisi 5.2.19), kita memperoleh lebih banyak rumus ekuivalen di mana urutannya adalah $xywz, xwyz, xwzy, wxyz, wxzy$, atau $wzxy$. Rumus-rumus ini semuanya ekuivalen karena kita menggunakan relasi ekuivalen dalam memperoleh PNF.

Proposisi 5.4.5 Setiap rumus orde pertama dapat ditransformasikan menjadi PNF yang ekuivalen.

Bukti Dengan Proposisi 5.2.18, penggantian nama variabel kuantifikasi mempertahankan ekuivalen. Aturan ekuivalen untuk PNF bersifat terminasi (soal latihan) dan mempertahankan ekuivalen. Dengan menerapkan aturan-aturan ini berulang kali, kita akan memperoleh hasil yang diinginkan.

Berikut ini, kita akan menggunakan $PrenexNF(A)$ untuk menunjukkan prosedur transformasi rumus A menjadi bentuk normal prenex.

Skolemisasi

Skolemisasi adalah proses di mana kita menghilangkan semua kuantifikasi, baik universal maupun eksistensial, sehingga hanya menyisakan rumus dengan variabel bebas. Proses ini dipermudah dengan memiliki rumus asli dalam bentuk normal prenex, dan kita mengasumsikan hal ini di bagian ini.

Menghilangkan kuantifikasi universal sambil mempertahankan makna itu mudah karena kita mengasumsikan bahwa makna variabel bebas sama dengan variabel yang dikuantifikasi secara universal: Rumus akan benar untuk setiap nilai variabel bebas. Dengan demikian, kita cukup membuang semua kuantifikasi universal teratas dari PNF.

Melakukan hal yang sama untuk variabel yang dikuantifikasi secara eksistensial tidak mungkin karena variabel bebas hanya dapat memiliki satu makna default. Ide kuncinya adalah memperkenalkan fungsi baru untuk menggantikan variabel tersebut. Fungsi-fungsi ini disebut fungsi Skolem. Ketika fungsi-fungsi tersebut nullary (yaitu, aritasnya 0), fungsi-fungsi tersebut disebut konstanta Skolem.

Contoh 5.4.6 Perhatikan rumus $\forall x \exists y (y^2 \leq x \wedge \neg((y+1)^2 \leq x))$. Rumus ini menyatakan keberadaan bilangan maksimal y yang kuadratnya tidak lebih dari x untuk setiap nilai x . Kita dapat menghilangkan kuantifikasi sehingga x menjadi bebas dan mempertahankan makna rumus tersebut. Efek yang sama dapat dicapai dengan menyatakan keberadaan fungsi f , yang bergantung pada x , yang memenuhi

$$f(x)^2 \leq x \wedge \neg((f(x)+1)^2 \leq x)$$

Fungsi f memang merupakan fungsi Skolem. Kita kemudian dapat menggunakan pengetahuan kita tentang aljabar untuk membuktikan bahwa $\lfloor \sqrt{x} \rfloor$ adalah solusi untuk $y = f(x)$, tetapi ini di luar cakupan perhatian kita saat ini.

Secara umum, parameter fungsi Skolem akan berupa variabel bebas, yang memiliki arti yang sama dengan variabel yang dikuantifikasi secara universal, sehingga memungkinkan nilai yang berbeda untuk setiap nilai variabel bebas yang berbeda.

Algoritma 5.4.7 $Sko(A)$ menerima input berupa rumus A dan mengembalikan rumus tanpa kuantifikasi. Prosedur $PrenexNF(A)$ mengembalikan formula dalam PNF dan setara dengan A . $free(B)$ mengembalikan daftar variabel bebas dalam B .

```

proc  $Sko(A)$ 
1   $A := PrenexNF(A)$ 
2  while  $A$  has a quantifier do
3    if  $(A = \exists x B)$   $A := B[x \sim f(free(B))]$  //  $f$  is a new Skolem function
4    else if  $(A = \forall x B)$   $A := B$ 
5  return  $A$ 

```

Algoritma $Sko(A)$ pertama-tama mengubah formula A menjadi bentuk normal prenex dengan $PrenexNF(A)$ (dibahas di bagian sebelumnya), kemudian mengupas setiap kuantifikasi dari kiri ke kanan, sampai semuanya dihilangkan.

Contoh 5.4.8 Misalkan A adalah $\forall x \exists y \exists z \exists w (p(x, y) \wedge (\neg q(z, w) \vee \neg q(w, z)))$ dari Contoh 5.4.4. A adalah PNF, jadi $PrenexNF(A) = A$. Baris 2–4 dari $Sko(A)$ pertama-tama akan menghilangkan $\forall x$, sehingga x menjadi bebas, dan kemudian mengganti y dengan $f_1(x)$, z dengan $f_2(x)$, dan w dengan $f_3(x)$, di mana f_1, f_2 , dan f_3 adalah fungsi Skolem yang diperkenalkan saat menghilangkan y, z , dan w . Jadi, $Sko(A)$ mengembalikan $p(x, f_1(x)) \wedge (\neg q(f_2(x), f_3(x)) \vee \neg q(f_3(x), f_2(x)))$. Untuk

$$B = \exists z \exists w \forall x \exists y (p(x, y) \wedge (\neg q(z, w) \vee \neg q(w, z))),$$

$Sko(B) = p(x, f_3(x)) \wedge (\neg q(c_1, c_2) \vee \neg q(c_2, c_1))$. Kita tahu bahwa $A \equiv B$ dari Contoh 5.4.4. Namun, $Sko(A) \not\equiv Sko(B)$, karena keduanya mengandung simbol fungsi yang berbeda.

Contoh di atas mengilustrasikan bahwa Skolemisasi tidak mempertahankan kesetaraan. Proses Skolemisasi, yang diterapkan pada formula dalam bentuk normal prenex, berakhir dengan formula yang setara lemah. Untuk melihat bahwa proses tersebut berakhir itu mudah, karena proses mengubah formula menjadi PNF berakhir dan jumlah kuantifikasi berkurang di dalam badan loop while dari $Sko(A)$.

Loop berlanjut selama kuantifikasi tetap berada di bagian atas formula, sehingga formula akhir tidak akan memiliki kuantifikasi. Rumus masukan dan keluaran tidak selalu setara, tetapi maknanya terkait erat, dan kita akan menangkap hubungan ini dengan gagasan "sama-sama dapat dipenuhi" atau "ekuisatisfiable".

Ingatlah bahwa dalam logika proposisional (Definisi 3.3.7), kita mengatakan dua rumus, A dan B , adalah ekuisatisfiable, yang dilambangkan dengan $A \approx B$, jika $M(A)$ setiap kali $M(B)$ dan sebaliknya. Hubungan ini lebih lemah daripada kesetaraan logis, yang membutuhkan $M(A) \approx M(B)$. Kita akan menggunakan kembali definisi $\approx$ yang sama untuk rumus orde

pertama. Mudah untuk melihat bahwa penghapusan kuantifikasi universal mempertahankan kesetaraan. Kesulitan muncul dengan pengenalan fungsi/konstanta Skolem.

Lemma 5.4.9 Misalkan A adalah $\forall x \exists y B(x, y)$. Maka terdapat fungsi $f(x)$ sedemikian sehingga (a) $\forall x B(x, f(x)) \models A$ dan (b) $\forall x B(x, f(x)) \approx A$.

Bukti

- (a) Untuk setiap model I dari $\forall x B(x, f(x))$, dan untuk setiap $d \in D^I$, harus berlaku bahwa $I(B(x, f(x)), x \sim d) = 1$. Misalkan $f^I(d) = e \in D^I$, di mana $f^I: D^I \rightarrow D^I$ adalah interpretasi f dalam I , maka $I(B(x, y), \{x \sim d, y \sim e\}) = 1$. Jadi, kita memiliki

$$I(\exists y B(x, y), \{x \sim d\}) = 1$$

Karena d sembarang di D^I , $I(\forall x \exists y B(x, y)) = 1$. Dengan kata lain, I adalah model dari A .

- (b) Karena (a) mengatakan “jika $\forall x B(x, f(x))$ dapat dipenuhi, maka A juga dapat dipenuhi,” kita hanya perlu menunjukkan bahwa jika A dapat dipenuhi, maka $\forall x B(x, f(x))$ juga dapat dipenuhi. Misalkan $I = (D, R, G)$ adalah model dari A . Maka untuk setiap $d \in D$, terdapat $e \in D$ sedemikian sehingga:

$$I(B(x, y), \{x \sim d, y \sim e\}) = 1$$

Misalkan $g: D \rightarrow D$ sedemikian sehingga $g(d) = e$, dan $I' = (D, R, G \cup \{g\})$ dengan $g = f^{I'} \text{ Ir}$. Maka I' adalah model dari $\forall x B(x, f(x))$.

Akibat 5.4.10 Misalkan $A = \forall x_1 \forall x_2 \cdots \forall x_k \exists y B(x_1, x_2, \dots, x_k, y)$. Maka terdapat fungsi $f(x_1, x_2, \dots, x_k)$ sedemikian sehingga

- (a) $\forall x_1 \forall x_2 \cdots \forall x_k B(x_1, x_2, \dots, x_k, f(x_1, x_2, \dots, x_k)) \models A$
 (b) $\forall x_1 \forall x_2 \cdots \forall x_k B(x_1, x_2, \dots, x_k, f(x_1, x_2, \dots, x_k)) \approx A$

Bukti Ganti x dengan $(x_1, x_2, \dots, x_k)$ dalam bukti Lemma 5.4.9.

Teorema 5.4.11 (a) $Sko(A) \models A$; dan (b) $Sko(A) \approx A$.

Bukti Karena variabel bebas dan variabel yang dikuantifikasi secara universal memiliki interpretasi yang sama, menghilangkan kuantifikasi universal tidak mengubah semantik suatu formula. Ketika kita menghilangkan kuantifikasi eksistensial, kita memperkenalkan fungsi Skolem. Berdasarkan Corollari 5.4.10, formula yang dihasilkan sama-sama dapat dipenuhi dengan formula asli. Dengan menerapkan Corollari 5.4.10 pada setiap kuantifikasi eksistensial, kita mendapatkan hasil yang diinginkan, karena baik $\models$ maupun $\approx$ bersifat transitif:

- (a): $Sko(A) \models A$ berdasarkan Corollari 5.4.10(a).

- (b): $A \approx Sko(A)$ berdasarkan Corollari 5.4.10(b).

Relasi $\approx$ ini disebut "ekuivalen lemah," dan kita akan melihat bahwa ini cukup untuk tujuan kita.

Skolemisasi Rumus Non-Prenex

Meskipun Skolemisasi mudah dijelaskan dan dibenarkan pada rumus dalam PNF, sebenarnya dapat diterapkan langsung pada rumus apa pun. Proses yang terlibat tidak mudah dijelaskan seperti pada kasus PNF. Sebagai gantinya, kami akan memberikan algoritma rekursif untuk Skolemisasi yang mengambil dua parameter: Parameter pertama adalah rumus yang akan di-Skolemisasi, yang harus dalam NNF (bentuk normal negasi), dan parameter kedua adalah daftar variabel bebas yang telah dikumpulkan sejauh ini.

Algoritma 5.4.12 $Skol(A, V)$ mengambil sebagai input rumus A yang berada dalam NNF (tidak mengandung operator Boolean selain $\neg$, $\wedge$, $\vee$ dan setiap variabel terkuantifikasi berbeda) dan daftar V variabel yang bebas dalam A , dan mengembalikan rumus tanpa kuantifikasi.

proc $Skol(A, V)$

- 1 **if** A is a literal or $A = \top$ or $A = \perp$ **return** A // base case
- 2 **if** $(A = B \text{ op } C)$ **return** $Skol(B, V) \text{ op } Skol(C, V)$ // op is either $\wedge$ or $\vee$
- 3 **if** $(A = \exists x B)$ **return** $Skol(B[x \sim f(V)], V)$ // f is a new Skolem function.
- 4 **if** $(A = \forall x B)$ **return** $Skol(B, V \cup \{x\})$

Ketika A adalah literal, yaitu, atom, atau negasi dari atom, A dikembalikan. Ketika operator teratas adalah $\wedge$ atau $\vee$, situasinya sederhana: $Skol$ hanya dipanggil secara rekursif pada setiap argumen operator. Jika simbol teratas adalah $\exists$, $Skol$ memperkenalkan fungsi Skolem baru f (konstanta Skolem jika V kosong) dan mengganti setiap kemunculan x dengan $f(V)$. Jika simbol teratas adalah $\forall$, variabel x dikumpulkan dalam parameter kedua untuk panggilan rekursif pada B . Dengan asumsi menghitung $B[x \sim f(V)]$ membutuhkan waktu linier, $Skol$ akan membutuhkan waktu kuadrat dalam hal ukuran rumus masukan.

Contoh 5.4.13 Misalkan A adalah NNF pertama dalam Contoh 5.4.4

$$A = (\forall x \exists y p(x, y)) \wedge (\exists z \exists w (\neg q(z, w) \vee \neg q(w, z)))$$

Kemudian $Skol(A, \emptyset)$ memanggil dirinya sendiri pada dua argumen dari $\wedge$:

- $Skol(\forall x \exists y p(x, y), \emptyset)$ memanggil $Skol(\exists y p(x, y), \{x\})$ dan mengembalikan $p(x, f(x))$.
- $Skol(\exists z \exists w \neg q(z, w) \vee \neg q(w, z), \emptyset)$ mengembalikan $\neg q(c_1, c_2) \vee \neg q(c_2, c_1)$.

Oleh karena itu, $Skol(A, \emptyset)$ mengembalikan $p(x, f(x)) \wedge (\neg q(c_1, c_2) \vee \neg q(c_2, c_1))$.

Teorema 5.4.14 (a) $Skol(A) \models A$, dan (b) $Skol(A) \approx A$.

Pembuktian teorema di atas mirip dengan Teorema 5.4.11 dan dihilangkan di sini. Seperti yang ditunjukkan dalam Contoh 5.4.4, kita memiliki banyak PNF yang berbeda untuk rumus yang sama dan setiap PNF dapat memberi kita hasil yang berbeda setelah Skolemisasi. Semua PNF dari rumus yang sama secara logis setara dan semua rumus Skolemisasi dari rumus yang sama dapat dipenuhi secara setara.

Dalam praktiknya, kita ingin menghasilkan rumus Skolemisasi di mana fungsi Skolem memiliki aritas minimal, sehingga istilah dasar dari rumus-rumus ini akan lebih mudah ditangani. Perhatikan bahwa aturan kesetaraan untuk memperoleh PNF mencoba untuk memperluas cakupan setiap variabel terkuantifikasi. Untuk memperoleh fungsi Skolem dengan jumlah argumen minimal, kita mungkin perlu menggunakan aturan kesetaraan ini secara terbalik.

Bentuk Klausula

Yang dimaksud dengan CNF adalah rumus merupakan konjungsi klausa, dan klausa merupakan disjungsi literal. Untuk memperoleh CNF, rumus A melalui tahapan-tahapan berikut:

- Tahap 1: Ubah A menjadi NNF A' .
- Tahap 2: Skolemisasi A' menjadi B yang bebas kuantifikasi.
- Tahap 3: Ubah B menjadi CNF C .

Tahap 3 mengubah NNF B yang telah di-Skolemisasi menjadi klausa, seperti yang kita lakukan dalam logika proposisional untuk mengubah NNF menjadi klausa. Artinya, gunakan hukum distribusi $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ berulang kali, hingga tidak lagi berlaku.

Tahap 1 dan 3 mempertahankan kesetaraan logis, $\equiv$, dan tahap 2 hanya mempertahankan kesetaraan lemah, $\approx$ (kesetaraan kepuasan). Dengan demikian, kita memiliki $A \equiv A'$, $A' \approx B$, dan $B \equiv C$. Akibatnya, $A \approx C$. Jadi, A dapat dipenuhi jika dan hanya jika C dapat dipenuhi.

Contoh 5.4.15 Perhatikan rumus-rumus berikut dan makna yang dimaksudkan:

- $A = \forall x (human(x) \rightarrow mortal(x))$: Setiap manusia fana.
- $B = \forall x (iowan(x) \rightarrow human(x))$: Setiap penduduk Iowa adalah manusia.
- $C = \forall x (iowan(x) \rightarrow mortal(x))$: Setiap penduduk Iowa fana.

Untuk menunjukkan $A \wedge B \models C$, kita mengubah $A \wedge B \wedge \neg C$ menjadi himpunan S yang terdiri dari empat klausa, di mana c_1 berasal dari A , c_2 dari B , c_3 dan c_4 dari $\neg C$:

$$\begin{aligned} c_1 &: (\neg human(x) \mid mortal(x)) \\ c_2 &: (\neg iowan(y) \mid human(y)) \\ c_3 &: (iowan(c)) \\ c_4 &: (\neg mortal(c)) \end{aligned}$$

Contoh dari c_1 adalah $c'_1 = (\neg human(c) \mid mortal(c))$, dan contoh dari c_2 adalah $c'_2 = (\neg iowan(c) \mid human(c))$. Resolusi dari c_3 dan c'_2 , kita peroleh $c_5 : (human(c))$. Dari c_5 dan c'_1 , kita peroleh $c_6 : (mortal(c))$. Dari c_6 dan c_4 , kita peroleh klausa *kosong* ().

Sekarang, kita tahu cara mengubah formula menjadi bentuk klausa, yang diperlukan oleh aturan resolusi. Ingatlah bahwa variabel dalam klausa diasumsikan bebas dan memiliki semantik yang sama dengan variabel yang dikuantifikasi secara universal. Variabel-variabel ini berbeda dari satu klausa ke klausa lainnya, meskipun menggunakan nama yang sama. Praktik umum adalah memberikan nama yang berbeda untuk variabel dalam klausa yang berbeda. Misalnya, dalam klausa c_i , kita menggunakan x_i, y_i, z_i , dll. Dengan cara ini, kita memastikan bahwa tidak ada variabel yang muncul di lebih dari satu klausa.

Tahap 3 konversi formula ke CNF dapat memperbesar ukuran formula keluaran, sementara tahap 2 menghasilkan formula dengan ukuran linier dari formula masukan. Dalam logika proposisional, untuk mendapatkan himpunan klausa dengan ukuran linier dari formula asli, kita dapat memperkenalkan variabel proposisional baru untuk setiap kemunculan operator biner dan klausa yang dihasilkan adalah 3CNF, yang tidak ekuivalen tetapi ekuisatisfiable dengan formula asli. Kita dapat melakukan hal yang sama pada tahap 3 sehingga klausa akhir memiliki ukuran linier dari formula asli.

Bentuk klausa mungkin tampak sebagai representasi yang sangat terbatas dan tidak alami untuk pengetahuan matematika, dan telah dikritik atas dasar tersebut. Kita akan melihat bahwa hal itu secara mengejutkan alami dalam banyak kasus. Bahkan, suatu masalah dapat diekspresikan sebagai himpunan kendala atas beberapa variabel, seperti masalah pemenuhan kendala (CSP). Suatu kendala dapat diekspresikan dalam CNF dan sekumpulan kendala memberi kita sekumpulan klausa.

Kita telah melihat bahwa setiap rumus kalkulus predikat dapat diubah menjadi rumus yang ekuisatisfiable dalam CNF, yang berguna untuk membuktikan hasil teoritis dengan resolusi.

Model Herbrand untuk CNF

Ketika kita mempertimbangkan interpretasi untuk sekumpulan rumus, ada terlalu banyak cara untuk memilih himpunan sebagai semesta, terlalu banyak cara untuk memilih relasi untuk simbol predikat, dan terlalu banyak cara untuk memilih fungsi untuk simbol fungsi. Kita dapat meningkatkan hal ini secara signifikan karena ada kelas interpretasi, yang diciptakan oleh matematikawan Prancis Jacques Herbrand (1908–1931), yang disebut interpretasi Herbrand, yang dapat mewakili semua interpretasi lainnya. Interpretasi Herbrand memiliki semesta yang sama, yang disebut semesta Herbrand. Apa yang terkandung dalam semesta ini? Untuk setiap simbol fungsi n -ary f dalam rumus (termasuk konstanta) dan n elemen, misalnya $t_1, t_2, \dots, t_n$, dalam semesta, simbol tersebut harus mengandung hasil penerapan f pada n elemen tersebut. Hal ini dapat dicapai dengan cara sintaksis sederhana; kita menempatkan semua istilah dasar dalam semesta dan membiarkan setiap istilah menunjukkan hasil penerapan fungsinya pada parameternya. Dengan cara ini, simbol fungsi terikat pada interpretasi unik di seluruh semesta.

Definisi 5.4.16 Diberikan formula CNF A dalam bahasa orde pertama $L = (P, F, X, Op)$, semesta Herbrand dari A terdiri dari semua istilah dasar, yaitu himpunan $T(F)$; basis Herbrand adalah himpunan semua atom dasar, yang dilambangkan dengan $B(P, F)$. Interpretasi Herbrand adalah pemetaan $I: B(P, F) \rightarrow \{1, 0\}$.

Perhatikan bahwa $T(F)$ adalah himpunan istilah bebas variabel yang dapat kita buat dari konstanta dan fungsi dalam formula. Jika tidak ada konstanta dalam F , $T(F)$ akan kosong. Untuk menghindari kasus ini, kita mengasumsikan bahwa setidaknya harus ada satu konstanta dalam F , misalnya a . Mulai sekarang akan berguna untuk membedakan konstanta dari fungsi non-konstan, jadi dengan "fungsi" kita maksudkan fungsi non-nullary. Jika F tidak mengandung simbol fungsi, maka $T(F)$ akan menjadi himpunan konstanta yang terbatas.

Basis Herbrand $B(P, F)$ terdiri dari semua atom seperti $p(t_1, t_2, \dots, t_n)$, di mana p adalah simbol predikat n -ary dalam P dan $t_i \in T(F)$. Jika $n = 0$ untuk semua predikat, maka $B(P, F) = P$, direduksi menjadi logika proposisional. Jika tidak, $B(P, F)$ terbatas jika dan hanya jika $T(F)$ terbatas. Untuk kenyamanan, kita dapat merepresentasikan interpretasi I dengan himpunan H literal dasar sedemikian sehingga untuk setiap $g \in B(P, F)$, $g \in H$ jika dan hanya jika $I(g) = 1$ dan $\bar{g} \in H$ jika dan hanya jika $I(g) = 0$, di mana $\bar{g}$ mewakili $\neg g$.

Contoh 5.4.17 Dalam Contoh 5.2.12, kita menunjukkan bahwa himpunan S tidak memiliki model terbatas:

$$S = \{\forall x \exists y p(x, y), \forall x \neg p(x, x), \forall x \forall y \forall z (p(x, y) \wedge p(y, z) \rightarrow p(x, z))\}$$

Perhatikan bahwa S tidak memiliki simbol fungsi. Dengan mengubah S ke CNF, kita memperoleh himpunan klausa S' berikut:

$$S' = \{c_1 : (p(x, f(x))), c_2 : (\overline{p(x, x)}), c_3 : (\overline{p(x, y)} \mid \overline{p(y, z)} \mid p(x, z))\}$$

Dengan menambahkan konstanta a , $F = \{f, a\}$ dan $P = \{p\}$, maka semesta Herbrand dari S' adalah

$$T(F) = \{a, f(a), f(f(a)), \dots, f^i(a), \dots\},$$

di mana $f^0(a) = a$ dan $f^{i+1}(a) = f(f^i(a))$. Basis Herbrand dari S' adalah

$$B(P, F) = \{p(a, a), p(a, f(a)), p(f(a), a), p(a, f(f(a))), \dots, p(f^i(a), f^j(a)), \dots\}$$

Interpretasi Herbrand terhadap S' adalah

$$H = \{p(f^i(a), f^j(a)) \mid 0 \leq i < j\} \cup \{\overline{p(f^i(a), f^j(a))} \mid 0 \leq j \leq i\}$$

Jika $f^i(a)$ menyatakan bilangan asli i , maka H identik dengan $R_< = \{(i, j) \mid i, j \in N, i < j\}$.

Proposisi 5.4.18 Interpretasi Herbrand mendefinisikan interpretasi unik untuk suatu formula dalam CNF.

Bukti Misalkan A adalah formula dalam CNF dan H adalah interpretasi Herbrand. Definisikan interpretasi $I = (D, R, G)$, di mana $D = T(F)$ (semesta Herbrand),

$$R = \{r_p \mid p \in P, r_p = \{\langle t_1, \dots, t_n \rangle \mid H(p(t_1, \dots, t_n)) = 1\}\}$$

dan $G = \{f^H \mid f \in F, f^H(t_1, \dots, t_n) = f(t_1, \dots, t_n)\}$, maka I adalah interpretasi sebagaimana didefinisikan dalam Definisi 5.2.1 dan didefinisikan secara unik oleh H .

Karena D dan G tetap dalam I , H adalah satu-satunya bagian yang memengaruhi I . Artinya, satu-satunya cara interpretasi Herbrand berbeda satu sama lain adalah pada makna simbol predikat. Proposisi di atas memberi kita pembenaran untuk menyebut H sebagai "interpretasi."

Setelah kita memiliki interpretasi, kita dapat menghitung nilai kebenaran dari formula A dalam interpretasi ini menggunakan prosedur seperti *eval* (Prosedur 5.2.3). Jika A benar di bawah interpretasi Herbrand H , kita katakan H adalah model Herbrand dari A . Karena CNF A dapat direpresentasikan oleh himpunan klausa S , dan variabel dalam setiap klausa memiliki arti yang sama dengan variabel yang dikuantifikasi secara universal, untuk setiap interpretasi I , $I(A) = I(S) = \bigwedge_{C \in S} I(C)$, dan untuk setiap klausa C , $I(C) = I(\forall x_1 \dots \forall x_k C)$, di mana $x_1, \dots, x_k$ adalah variabel bebas dari C .

Definisi interpretasi Herbrand memberi kita cara yang mudah untuk mengekspresikan arti-arti ini dengan menetapkan nilai kebenaran untuk setiap atom dasar dalam basis Herbrand. Tentu saja, nilai kebenaran beberapa atom tidak relevan jika atom-atom ini tidak dapat menjadi contoh dari predikat apa pun dalam rumus, karena kebenaran rumus tidak bergantung pada atom-atom ini. Mulai sekarang, ketika kita mengatakan "sekumpulan klausa memiliki model", kita maksudkan "ia memiliki model Herbrand."

Bab ini memperkenalkan konsep-konsep dasar logika orde pertama, yang juga disebut kalkulus predikat karena merupakan perluasan dari logika proposisional dengan predikat beserta variabel, fungsi, dan kuantifikasi. Dalam tiga bab berikutnya, kita akan memperkenalkan alat-alat penalaran dalam logika orde pertama.

5.5 LATIHAN SOAL

1. Dengan menggunakan bahasa orde pertama dengan variabel yang mencakup orang dan predikat $trusts(x, y)$, $politician(x)$, $crazy(x)$, $know(x, y)$, dan $related(x, y)$, dan $rich(x)$, tuliskan kalimat orde pertama yang menyatakan hal berikut:
 - (a) Tidak ada seorang pun yang mempercayai seorang politisi.
 - (b) Siapa pun yang mempercayai seorang politisi adalah gila.
 - (c) Semua orang mengenal seseorang yang berhubungan dengan seorang politisi.

- (d) Semua orang yang kaya adalah seorang politisi atau mengenal seorang politisi.
2. Misalkan arti dari $taken(x, y)$ adalah “mahasiswa x telah mengambil mata kuliah $CS\ y$.” Silakan tuliskan arti dari rumus-rumus berikut:
- $\exists x \exists y\ taken(x, y)$
 - $\exists x \forall y\ taken(x, y)$
 - $\forall x \exists y\ taken(x, y)$
 - $\exists y \forall x\ taken(x, y)$
 - $\forall y \exists x\ taken(x, y)$
3. Misalkan $c(x)$ berarti “ x adalah seorang kriminal” dan $s(x)$ berarti “ x waras.” Gunakan keduanya untuk menulis rumus orde pertama untuk setiap kalimat berikut.
- Semua kriminal waras.
 - Setiap kriminal waras.
 - Hanya kriminal yang waras.
 - Beberapa kriminal waras.
 - Ada seorang kriminal yang waras.
 - Tidak ada kriminal yang waras.
 - Tidak semua kriminal waras.
4. Diberikan predikat $male(x)$, $female(x)$, dan $parent(x, y)$ (x adalah orang tua dari y), definisikan predikat-predikat berikut dalam bentuk $male$, $female$, $parent$, atau predikat lain yang didefinisikan dari ketiga predikat ini, dalam logika orde pertama:
- $father(x, y)$: x adalah ayah dari y .
 - $mother(x, y)$: x adalah ibu dari y .
 - $ancestor(x, y)$: x adalah leluhur dari y .
 - $descendant(x, y)$: x adalah keturunan dari y .
 - $son(x, y)$: x adalah putra dari y .
 - $daughter(x, y)$: x adalah putri dari y .
 - $sibling(x, y)$: x adalah saudara kandung dari y .
5. Misalkan $+$, $-$, $*$, $\%$, $>$, $<$, $\geq$, $\leq$, $=$ adalah operasi biasa pada bilangan asli N , di mana $(x \% y)$ mengembalikan sisa pembagian x oleh y untuk $y > 0$. Harap definisikan himpunan-himpunan berikut menggunakan rumus orde pertama:
- Himpunan bilangan asli yang habis dibagi 5, tetapi tidak habis dibagi 3.
 - Himpunan bilangan asli yang kuadratnya merupakan jumlah dari dua bilangan kuadrat.
 - Himpunan bilangan prima.
6. Berikan bukti Teorema 5.2.25.
7. Misalkan Himpunan didefinisikan oleh BNF berikut:
- $$\langle Set \rangle ::= \emptyset \mid \text{adjoin}(\langle Any \rangle, \langle Set \rangle)$$
- di mana $\text{adjoin}(e, s)$ menambahkan elemen e ke dalam himpunan s . Harap definisikan predikat berikut dalam istilah $\emptyset$ dan adjoin dalam logika orde pertama:
- $isSet(x)$: x adalah objek dari Himpunan .
 - $member(x, s)$: x adalah anggota dari himpunan s .

- (c) $subset(s, t)$: s adalah himpunan bagian dari himpunan t .
- (d) $equal(s, t)$: s dan t mengandung elemen yang sama.
8. Tentukan apakah rumus-rumus berikut valid, tidak valid tetapi dapat dipenuhi, atau tidak dapat dipenuhi, menggunakan definisi interpretasi:
- $p(a, a) \vee \neg(\exists x p(x, a))$
 - $\neg p(a, a) \vee (\exists x p(x, a))$
 - $\neg q(a) \vee (\forall x q(x))$
 - $\neg(\exists x q(x)) \vee (\forall x q(x))$
 - $\exists x (q(x) \vee \neg \forall x q(x))$
9. Buktikan relasi ekuivalensi berikut:
- $\neg \exists x p(x) \equiv \forall x \neg p(x)$
 - $\exists x (p(x) \vee q(x)) \equiv (\exists x p(x)) \vee (\exists x q(x))$
 - $(\exists x (p(x)) \wedge q) \equiv (\exists x (p(x) \wedge q))$
 - $\exists x \exists y r(x, y) \equiv \exists y \exists x r(x, y)$
 - $\exists x (p(x) \rightarrow q(x)) \equiv (\forall x p(x)) \rightarrow (\exists x q(x))$
10. Buktikan bahwa relasi implikasi berikut ini benar, tetapi kebalikannya tidak benar.
- $(\forall x p(x)) \vee (\forall x q(x)) \models \forall x (p(x) \vee q(x))$
 - $\exists x (p(x) \wedge q(x)) \models (\exists x p(x)) \wedge (\exists x q(x))$
 - $(\forall x p(x)) \rightarrow (\forall x q(x)) \models \forall x (p(x) \rightarrow q(x))$
 - $(\exists x p(x)) \rightarrow (\exists x q(x)) \models \exists x (p(x) \rightarrow q(x))$
 - $(\exists x p(x)) \rightarrow (\forall x q(x)) \models \forall x (p(x) \rightarrow q(x))$
 - $\forall x (p(x) \rightarrow q(x)) \models (\exists x p(x)) \rightarrow (\exists x q(x))$
 - $\forall x (p(x) \rightarrow q(x)) \models (\forall x p(x)) \rightarrow (\exists x q(x))$
 - $\forall x (p(x) \leftrightarrow q(x)) \models (\forall x p(x)) \leftrightarrow (\forall x q(x))$
 - $\forall x (p(x) \leftrightarrow q(x)) \models (\exists x p(x)) \leftrightarrow (\exists x q(x))$
11. Gunakan metode tableau semantik untuk menunjukkan bahwa relasi implikasi benar pada masalah sebelumnya.
12. Berikan pseudo-kode untuk algoritma $PrenexNF(A)$ dan tunjukkan bahwa algoritma tersebut berjalan dalam waktu $O(n)$ ketika A berada dalam NNF, di mana n adalah ukuran A .
13. Tunjukkan bahwa algoritma $Sko(A)$ berjalan dalam waktu $O(n^2)$, di mana A berada dalam NNF, dan n adalah ukuran A .
14. Tunjukkan bahwa algoritma $Skol(A)$ berjalan dalam waktu $O(n^2)$, di mana A berada dalam NNF, dan n adalah ukuran A .
15. Tunjukkan bahwa rumus $\forall x \exists y ((p(x) \wedge p(y)) \vee (q(x) \wedge q(y)))$ dapat diubah menjadi bentuk normal prenex yang setara sedemikian sehingga berada sebelum dalam bentuk normal prenex tersebut.
16. Buktikan bahwa jika setiap simbol predikat bersifat monodik (yaitu, uner) dan setiap simbol fungsi bersifat monodik atau konstanta, maka setiap rumus orde pertama dapat diubah menjadi bentuk normal prenex di mana $\exists$ berada sebelum $\forall$.
17. Ubahlah setiap rumus berikut menjadi serangkaian klausa:

- (a) $\neg((\forall x p(x)) \vee (\forall x q(x)) \rightarrow \forall x (p(x) \vee q(x)))$
 (b) $\neg((\exists x (p(x) \wedge q(x))) \rightarrow ((\exists x p(x)) \wedge (\exists x q(x))))$
 (c) $\neg(((\forall x p(x)) \rightarrow (\forall x q(x))) \rightarrow (\forall x (p(x) \rightarrow q(x))))$
 (d) $\neg(((\exists x (p(x) \rightarrow q(x))) \rightarrow (\exists x p(x))) \rightarrow (\exists x q(x)))$
 (e) $\neg((\exists x p(x)) \rightarrow ((\forall x q(x)) \rightarrow (\forall x (p(x) \rightarrow q(x)))))$
 (f) $\neg(((\forall x (p(x) \rightarrow q(x))) \rightarrow (\exists x p(x))) \rightarrow (\exists x q(x)))$
 (g) $\neg((\forall x (p(x) \rightarrow q(x))) \rightarrow ((\forall x p(x)) \rightarrow (\exists x q(x))))$
 (h) $\neg(((\forall x (p(x) \leftrightarrow q(x))) \rightarrow (\forall x p(x))) \leftrightarrow (\forall x q(x)))$
 (i) $\neg(((\forall x (p(x) \leftrightarrow q(x))) \rightarrow (\exists x p(x))) \leftrightarrow (\exists x q(x)))$
18. Diberikan himpunan klausa $S, S = \{(p(f(x), c)), (\neg p(f(w), z) \mid p(f(f(w))), \text{mohon jawab pertanyaan berikut: (a) Apa semesta Herbrand dari } S? \text{ (b) Apa basis Herbrand dari } S? \text{ (c) Apa model Herbrand minimum dari } S \text{ (tidak ada subset dari model ini yang merupakan model)? (d) Apa model Herbrand maksimum dari } S \text{ (tidak ada superset dari model ini yang merupakan model)?}$
19. Misalkan c adalah:
 $(\overline{p(x, y)} \mid \overline{p(y, z)} \mid q(z) \mid p(x, z))$ dan
 $S = \{(\overline{p(x, y)} \mid \overline{p(y, z)} \mid r(x, z)), (\overline{} \mid q(x, z) \mid p(x, z))\}.$
 Tunjukkan bahwa $S \models c$ dan $S \approx c$. Secara umum, tunjukkan bahwa klausa c yang terdiri dari n literal dapat diubah menjadi himpunan S yang terdiri dari $n - 2$ klausa yang masing-masing terdiri dari 3 literal (3CNF) sedemikian sehingga $S \models c$ dan $S \approx c$.

BAB 6

UNIFIKASI DAN RESOLUSI

Resolusi dikenal sebagai metode pembuktian sanggahan, yang dapat menunjukkan bahwa rumus masukan tidak dapat dipenuhi. Jika kita ingin menunjukkan bahwa rumus A valid, kita perlu mengubah $\neg A$ menjadi klausa dan menunjukkan bahwa klausa tersebut tidak dapat dipenuhi. Untuk menunjukkan bahwa $A \models B$, kita perlu mengubah $A \wedge \neg B$ menjadi CNF dan menunjukkan bahwa klausa tersebut tidak dapat dipenuhi. Perhatikan bahwa A dan B harus berupa rumus tertutup.

Misalnya, jika kita ingin menyatakan bahwa $p(x, y)$ adalah relasi komutatif, kita dapat menggunakan rumus $p(x, y) \rightarrow p(y, x)$, di mana x dan y diasumsikan sebagai nilai sembarang. Negasi dari rumus ini adalah $\neg(\forall x \forall y (p(x, y) \rightarrow p(y, x)))$, bukan $\neg(p(x, y) \rightarrow p(y, x))$. Bentuk normal konjugasi (CNF) dari yang pertama adalah $p(a, b) \wedge \neg p(b, a)$, di mana a dan b adalah konstanta Skolem; CNF dari yang kedua adalah $p(x, y) \wedge \neg p(y, x)$, yang setara dengan himpunan klausa $\{(p(x, y)), (\neg p(y, x))\}$. Kedua klausa unit ini akan menghasilkan klausa kosong melalui resolusi ketika x digantikan oleh y . Proses menemukan substitusi untuk membuat dua rumus atomik (yaitu, atom) identik disebut "unifikasi," yang dibutuhkan dalam definisi aturan resolusi untuk rumus orde pertama.

6.1 UNIFIKASI

Dalam Contoh 5.4.15, ketika kita menerapkan aturan resolusi

$$(A \mid \alpha), (\neg A \mid \beta) \vdash (\alpha \mid \beta)$$

pada dua klausa, kita perlu menemukan contoh dari kedua klausa tersebut sedemikian sehingga satu contoh berisi A dan contoh lainnya berisi $\neg A$, di mana A adalah sebuah atom. Sebuah klausa memiliki contoh yang tak terhingga banyaknya, dan proses unifikasi membantu kita menemukan contoh yang tepat untuk resolusi.

Substitusi dan Penyatuan

Diberikan bahasa orde pertama $L = (P, F, X, Op)$, substitusi adalah pemetaan $\sigma : X \rightarrow T(F, X)$ yang memetakan setiap variabel X ke suatu term $T(F, X)$. Jika $\sigma(x) \neq x$, kita katakan x dipengaruhi oleh σ . Jika $\sigma(x)$ adalah variabel yang berbeda untuk setiap x yang dipengaruhi, σ dikatakan sebagai penamaan ulang.

Saat menampilkan σ , kita menggunakan $[x \sim \sigma(x)]$, di mana x biasanya merupakan variabel yang dipengaruhi. Misalnya, jika $X = \{x, y\}$, maka x dipengaruhi oleh $\sigma_1 = [x \sim f(y)]$ dan $\sigma_2 = [x \sim f(x)]$; baik x maupun y dipengaruhi oleh $\sigma_3 = [x \sim f(y), y \sim a]$.

Substitusi $\sigma : X \rightarrow T(F, X)$ dapat diperluas menjadi pemetaan σ dari suku ke suku sedemikian rupa sehingga jika $t = x$, maka $\sigma(t) = \sigma(x)$; jika $t = f(t_1, t_2, \dots, t_k)$, maka $\sigma(t) = f(\sigma(t_1), \sigma(t_2), \dots, \sigma(t_k))$. Dengan cara yang serupa, σ dapat diperluas menjadi pemetaan dari rumus ke rumus.

Kita menggunakan $p(t)$ untuk menunjukkan sebuah contoh dari $p(x)$, yaitu, $p(t) = p(x)[x \sim t]$, di mana setiap kemunculan x digantikan oleh suku t . Di sini, $\sigma = [x \sim t]$ adalah substitusi dan $p(t) = \sigma(p(x)) = p(x)\sigma$, karena kita bebas untuk menulis $p(x)\sigma$ atau $\sigma(p(x))$.

Definisi 6.1.1 (Substitusi Idempoten) Substitusi σ dikatakan idempoten jika untuk setiap suku t , $t\sigma = (t\sigma)\sigma$.

Misalnya, $\sigma_1 = [x \sim f(y)]$ adalah idempoten; $\sigma_2 = [x \sim f(y), y \sim a]$ bukan idempoten, karena $x\sigma_2 = f(y)$ dan $(x\sigma_2)\sigma_2 = f(a)$. Beberapa substitusi non-idempoten dapat dibuat menjadi substitusi idempoten. Misalnya, σ_2 dapat digantikan oleh $\sigma_3 = [x \sim f(a), y \sim a]$, yang bersifat idempoten. Beberapa substitusi seperti $\sigma_4 = [x \sim f(x)]$ tidak pernah dapat dibuat menjadi substitusi idempoten.

Untuk substitusi idempoten, variabel yang terpengaruh tidak pernah dikembalikan oleh variabel lain yang terpengaruh.

Proposisi 6.1.2 Suatu substitusi σ bersifat idempoten jika dan hanya jika untuk setiap $x \in X$, baik $\sigma(x) = x$ atau x tidak muncul dalam $\sigma(y)$ untuk setiap $y \in X$.

Bukti Diberikan sembarang σ , jika x yang terpengaruh muncul dalam $\sigma(y) = t(x)$, misalkan $\sigma(x) = s$, maka $y\sigma = t(x)$ dan $(y\sigma)\sigma = t(s)$, sehingga σ tidak idempoten. Jika tidak, untuk setiap istilah s , $s\sigma$ tidak mengandung variabel apa pun yang dipengaruhi oleh σ , sehingga $(s\sigma)\sigma = s\sigma$ dan σ bersifat idempoten.

Definisi 6.1.3 (Penyatu) Misalkan s, t adalah dua istilah dari $T(F, X)$ atau dua atom. Kita katakan s dan t dapat disatukan, jika ada substitusi idempoten $\sigma : X \rightarrow T(F, X)$ sedemikian sehingga $s\sigma = t\sigma$; σ dikatakan sebagai penyatu s dan t .

Suatu himpunan S yang terdiri dari pasangan istilah atau atom dapat disatukan jika terdapat substitusi σ sedemikian sehingga untuk setiap pasangan $(s_i, t_i) \in S$, $s_i\sigma = t_i\sigma$, dan σ dikatakan sebagai penyatu S .

Contoh 6.1.4 Misalkan $s = f(x, g(y))$ dan $t = f(g(z), x)$, maka s dan t dapat disatukan dengan penyatu $\sigma = [x \sim g(z), y \sim z]$, karena

$$\begin{aligned} s\sigma &= f(x, g(y))[x \sim g(z), y \sim z] = f(g(z), g(z)) \\ &= f(g(z), x)[x \sim g(z), y \sim z] = t\sigma \end{aligned}$$

Faktanya, s dan t memiliki banyak sekali penyatu seperti

$$[x \sim g(w), y \sim w, z \sim w]$$

untuk setiap $w \in T(F, X)$ dan $w \neq x$.

Dalam contoh di atas, kita melihat bahwa sepasang istilah dapat memiliki beberapa penyatu. Bagaimana Anda membandingkan penyatu-penyatu ini? Kita membutuhkan beberapa relasi atas penyatu dan relasi ini didasarkan pada komposisi substitusi.

Menggabungkan Substitusi

Definisi 6.1.5 (Komposisi Substitusi) Diberikan dua substitusi σ dan θ , komposisi σ dan θ adalah substitusi γ , ditulis $\gamma = \sigma \cdot \theta$, atau disingkat $\sigma\theta$, di mana

$$\gamma = [x \sim t\theta \mid (x \sim t) \in \sigma]$$

Definisi di atas mengasumsikan bahwa σ mengandung setiap variabel y , termasuk $y = y\sigma$.

Jika tidak, kita perlu menambahkan secara eksplisit $[y \sim s \mid y = y\sigma, (y \sim s) \in \theta]$ ke dalam γ . Kita akan mengasumsikan bahwa $t\sigma\theta = (t\sigma)\theta = \theta(\sigma(t))$. Untuk menerapkan komposisi $\sigma\theta$ ke t , kita tulis $t(\sigma\theta)$.

Contoh 6.1.6 Diberikan $\sigma = [x \sim f(y), z \sim f(u), v \sim u]$ dan $\theta = [y \sim g(a), u \sim z, v \sim f(f(a))]$ maka

$$\begin{aligned} \sigma\theta &= [x \sim f(g(a)), y \sim g(a), z \sim f(z), u \sim z, v \sim z] \\ p(x, y, z, u, v)\sigma\theta &= p(f(y), y, f(u), u, u)\theta \\ &= p(f(g(a)), g(a), f(z), z, z) \\ &= p(x, y, z, u, v)(\sigma\theta) \end{aligned}$$

Perhatikan bahwa komposisi tidak selalu mempertahankan sifat idempoten karena baik σ maupun θ bersifat idempoten tetapi $z \sim f(z)$ muncul dalam $\sigma\theta$.

Proposisi 6.1.7 Untuk setiap substitusi σ dan θ , dan setiap suku t , $t\sigma\theta = t(\sigma\theta)$.

Bukti Untuk setiap $x \in X$, jika x dipengaruhi oleh σ , misalnya $\sigma(x) = s$, maka $x\sigma\theta = s\theta = x(\sigma\theta)$. Jika x tidak dipengaruhi oleh σ , maka $x\sigma\theta = x\theta = x(\sigma\theta)$.

Proposisi di atas menyatakan sifat bahwa menerapkan dua substitusi pada suatu suku secara berurutan menghasilkan hasil yang sama dengan menerapkan komposisi dua substitusi pada suku tersebut.

Operasi komposisi pada umumnya tidak komutatif. Misalnya, jika $\sigma = [x \sim a]$ dan $\theta = [x \sim b]$, maka $\sigma\theta = \sigma$ dan $\theta\sigma = \theta$. Jadi, $\sigma\theta \neq \theta\sigma$. Menurut definisi, jika x dipengaruhi oleh σ dan θ , yang merupakan idempoten, maka pasangan $x \sim \theta(x)$ diabaikan dalam $\sigma\theta$ dan pasangan $x \sim \sigma(x)$ diabaikan dalam $\theta\sigma$.

Proposisi 6.1.8 Komposisinya bersifat asosiatif, yaitu untuk sembarang substitusi σ, θ , dan γ , $(\sigma\theta)\gamma = \sigma(\theta\gamma)$.

Buktinya dibiarkan sebagai latihan.

Unifikasi Berbasis Aturan

Definisi 6.1.9 (Unifier Paling Umum) Misalkan σ dan θ adalah unifier dari s dan t . Kita katakan σ lebih umum daripada θ , jika ada substitusi γ sedemikian sehingga $\theta = \sigma\gamma$. σ adalah unifier paling umum (mgu) dari s dan t , jika σ lebih umum daripada unifier mana pun dari s dan t .

Perhatikan bahwa mgu bersifat unik setelah penggantian nama variabel. Misalnya, $[x \sim y]$, $[y \sim x]$, dan $[x \sim z, y \sim z]$ adalah mgu dari $f(x)$ dan $f(y)$. Mereka dapat dikonversi satu sama lain dengan penggantian nama substitusi.

Contoh 6.1.10 Dalam Contoh 6.1.4, $\sigma = [x \sim g(z), y \sim z]$ adalah mgu dari $s = f(x, g(y))$ dan $t = f(g(z), x)$. Untuk setiap unifier $\theta = [x \sim g(t'), y \sim t', z \sim t']$, $\theta = \sigma\gamma$, di mana $\gamma = [z \sim t']$. Mgu lain dari s dan t adalah $\sigma' = [x \sim g(y), z \sim y]$, $\sigma = \sigma'[y \sim z]$ dan $\sigma' = \sigma[z \sim y]$.

Seperti yang diilustrasikan oleh contoh di atas, ada lebih dari satu mgu untuk sepasang suku; perbedaannya terletak pada penamaan ulang variabel. Kita dapat mengatakan unifier yang paling umum jika penamaan ulang variabel dipertimbangkan.

Diberikan sepasang istilah s dan t , bagaimana cara menentukan apakah s dan t dapat disatukan? Jika ya, bagaimana cara menghitung mgu-nya? Pertanyaan-pertanyaan ini akan dijawab oleh algoritma penyatuan.

Misalkan $s \doteq t$ menyatakan bahwa “ s dan t dapat disatukan,” dan algoritma penyatuan akan menentukan apakah $s \doteq t$ benar atau salah. Misalkan $s = f(s_1, s_2, \dots, s_m)$ dan $t = g(t_1, t_2, \dots, t_n)$. Jika $f \neq g$ atau $m \neq n$, maka $s \doteq t$ salah, yaitu, s dan t tidak dapat disatukan.

Ini disebut kegagalan benturan (clash failure). Jika $s \doteq t$ benar, maka $s_i \doteq t_i$ juga benar untuk $1 \leq i \leq m = n$. Dengan kata lain, jika $s_i \doteq t_i$ salah untuk beberapa i , maka $s \doteq t$ salah. Jika $s = x$ dan $t = f(x)$ (x muncul dalam t), maka tidak ada substitusi σ sedemikian sehingga $x\sigma = f(x)\sigma$. Oleh karena itu, $x = f(x)$ harus salah. Ini disebut kegagalan pemeriksaan kemunculan, karena x muncul dalam $f(x)$. Jika kita menerima $x \sim f(x)$, maka penyatu tidak dapat bersifat idempoten.

Algoritma penyatuan berbasis aturan menggunakan serangkaian aturan yang mengubah $s \doteq t$ menjadi substitusi σ melalui serangkaian pasangan $\{s_i \doteq t_i\}$ dan setiap aturan mempertahankan ketersatuan dari $s \doteq t$. Aturan-aturan ini berlaku untuk (S, σ) , di mana S adalah himpunan pasangan $s_i \doteq t_i$ dan σ adalah penyatu yang akan dihitung. Awalnya, $S = \{s \doteq t\}$ dan σ kosong.

Definisi 6.1.11 Algoritma penyatuan berbasis aturan akan mencoba mengubah $(\{s \doteq t\}, [])$ menjadi $(\emptyset, \sigma)$, di mana σ adalah mgu dari s dan t , dengan aturan penyatuan berikut:

- **Decompose:** $\langle S \cup \{f(s_1, s_2, \dots, s_m) \doteq f(t_1, t_2, \dots, t_m)\}, \sigma \rangle \rightarrow \langle S \cup \{s_i \doteq t_i \mid 1 \leq i \leq m\}, \sigma \rangle$
- **Clash:** $\langle S \cup \{f(s_1, s_2, \dots, s_m) \doteq g(t_1, t_2, \dots, t_n)\}, \sigma \rangle \rightarrow \langle \emptyset, fail \rangle$
- **Occur – check:** $\langle S \cup \{x \doteq t \mid x \neq t, x \text{ occurs in } t\}, \sigma \rangle \rightarrow \langle \emptyset, fail \rangle$.
- **Redundant:** $\langle S \cup \{t \doteq t\}, \sigma \rangle \rightarrow \langle S, \sigma \rangle$
- **Orient:** $\langle S \cup \{t \doteq x \mid t \text{ is not variable}\}, \sigma \rangle \rightarrow \langle S \cup \{x \doteq t\}, \sigma \rangle$
- **Substitute:** $\langle S \cup \{x \doteq t \mid x \text{ does not occur in } t\}, \sigma \rangle \rightarrow \langle S[x \sim t], \sigma \cdot [x \sim t] \rangle$

Algoritma 6.1.12 Algoritma *unify*(s, t) mengambil dua istilah s dan t dan mengembalikan fail atau σ , a mgu dari s dan t .

```

proc unify( $s, t$ )
1   $S := \{s \doteq t\}; \sigma = []$ 
2  while (true) do
3     $\langle S, \sigma \rangle :=$  apply one unification rule on  $\langle S, \sigma \rangle$ 
4    if ( $\sigma = fail$ ) return fail
5    if ( $S = \emptyset$ ) return  $\sigma$ 

```

Contoh 6.1.13 Diberikan $s = p(f(x, h(x), y), g(y))$ dan $t = p(f(g(z), w, z), x)$, *unify*(s, t) akan mengubah $(\{s \doteq t\}, [])$ menjadi $(\emptyset, \sigma)$ sebagai berikut:

```

            $\{s \doteq t\}, []$ 
decompose  $\rightarrow \{f(x, h(x), y) \doteq f(g(z), w, z), g(y) \doteq x\}, []$ 
orient       $\rightarrow \{f(x, h(x), y) \doteq f(g(z), w, z), x \doteq g(y)\}, []$ 
substitute  $\rightarrow \{f(g(y), h(g(y)), y) \doteq f(g(z), w, z)\}, [x \sim g(y)]$ 
decompose  $\rightarrow \{g(y) \doteq g(z), h(g(y)) \doteq w, y \doteq z\}, [x \sim g(y)]$ 
substitute  $\rightarrow \{g(z) \doteq g(z), h(g(z)) \doteq w\}, [x \sim g(z), y \sim z]$ 
redundant   $\rightarrow \{h(g(z)) \doteq w\}, [x \sim g(z), y \sim z]$ 
orient       $\rightarrow \{w \doteq h(g(z))\}, [x \sim g(z), y \sim z]$ 
substitute  $\rightarrow \emptyset, [x \sim g(z), y \sim z, w \sim h(g(z))]$ 

```

Algoritma *unify*(s, t) tidak menentukan urutan aturan, sehingga kita dapat menggunakan aturan-aturan ini dalam urutan apa pun jika sesuai. Namun, tidak satu pun dari aturan-aturan ini dapat diterapkan selamanya. Oleh karena itu, *unify*(s, t) pada akhirnya akan berakhir.

Lemma 6.1.14 Diberikan pasangan s dan t apa pun, *unify*(s, t) akan berakhir dengan gagal atau σ .

Bukti Pertama, kita menunjukkan bahwa aturan **substitusi** dapat meningkatkan ukuran suku dalam S tetapi memindahkan variabel x dari S ke σ , yaitu, x menghilang dari S , sehingga aturan ini tidak dapat diterapkan lebih dari jumlah variabel.

Untuk aturan yang digunakan di antara dua penerapan aturan substitusi, jika aturan benturan atau aturan pemeriksaan kemunculan berlaku, proses berakhir dengan kegagalan segera. Jika tidak, aturan dekomposisi menghapus dua kemunculan f . Aturan orientasi

mengalihkan $t \doteq x$ menjadi $x \doteq t$, di mana t bukan variabel. Aturan redundansi menghapus $t \doteq t$. Tidak satu pun dari aturan ini dapat digunakan selamanya. Pada akhirnya, akan dikembalikan nilai gagal, atau S menjadi kosong dan σ dikembalikan.

Lemma 6.1.15 Jika (S', σ') adalah hasil penerapan aturan substitusi pada (S, σ) dan θ' adalah mgu dari S' maka $\theta = [x \sim t] \cdot \theta'$ adalah mgu dari S .

Bukti Berdasarkan definisi aturan substitusi, $S' = (S - \{x \doteq t\})[x \sim t]$, di mana x tidak muncul dalam t . Berdasarkan asumsi, S' dapat disatukan dengan mgu θ' . Karena x tidak muncul di S' , $\theta = [x \sim t] \cdot \theta'$ adalah mgu dari $S' \cup [x \doteq t]$. Mudah untuk memeriksa bahwa θ juga merupakan mgu dari S karena $S\theta = S([x \sim t] \cdot \theta') = (S[x \sim t])\theta' = S'\theta' \cup \{t \doteq t\}$.

Teorema 6.1.16 Algoritma $unify(s, t)$ akan mengembalikan mgu dari s dan t jika dan hanya jika s dan t dapat disatukan.

Bukti Lemma 6.1.14 memastikan bahwa $unify(s, t)$ akan berakhir. Di dalam $unify(s, t)$, ketika (S, σ) ditransformasikan menjadi (S', σ') , untuk setiap aturan, kita dapat memeriksa bahwa S dapat diunifikasi (σ tidak memengaruhi unifikasi S) jika dan hanya jika $\sigma' \neq \text{gagal}$ dan S' dapat diunifikasi. Artinya, ketika σ' gagal, S tidak dapat diunifikasi. Ketika $S' = \emptyset$, S' secara trivial dapat diunifikasi, sehingga S harus dapat diunifikasi.

Sebelum aturan substitusi memindahkan $x \doteq t$ dari S , x tidak dipengaruhi oleh σ , $t\sigma = t$ dan t tidak mengandung x . Misalkan $\sigma' = \sigma \cdot [x \sim t] = \sigma[x \sim t] \cup [x \sim t]$. Karena $S[x \sim t]$ dan $\sigma[x \sim t]$ menghilangkan semua kemunculan x dalam S dan σ , x memiliki kemunculan unik dalam σ' dan hal itu juga berlaku untuk semua variabel yang terpengaruh dalam σ . Dengan demikian, σ dan σ' bersifat idempoten menurut Proposisi 6.1.2.

Sekarang anggap s dan t dapat disatukan. Misalkan $S_0 = \{s \doteq t\}$, $\sigma_0 = []$, dan $\sigma_i = \sigma_{i-1} \cdot [x_i \sim t_i]$ mencatat setiap penerapan aturan substitusi untuk $1 \leq i \leq m$. Misalkan S yang bersesuaian adalah $S_1, S_2, \dots, S_m$, masing-masing, di mana $S_m = \emptyset$. Karena mgu dari S_m adalah $\theta_m = []$. Jadi, menurut Lemma 6.1.15,

$$\theta_{m-1} = [x_m \sim t_m]\theta_m = [x_m \sim t_m]$$

adalah mgu dari S_{m-1} . Sekali lagi, berdasarkan Lemma 6.1.15,

$$\theta_{m-2} = [x_{m-1} \sim t_{m-1}]\theta_{m-1} = [x_{m-1} \sim t_{m-1}, x_m \sim t_m]$$

adalah mgu dari S_{m-2} , dan seterusnya. Akhirnya,

$$\theta_0 = [x_1 \sim t_1]\theta_1 = [x_1 \sim t_1, \dots, x_{m-1} \sim t_{m-1}, x_m \sim t_m] = \sigma_m$$

adalah mgu dari $S_0 = \{s \doteq t\}$.

Contoh 6.1.17 Perhatikan pasangan suku berikut:

$$s = f(f(f(f(x_1, a), x_2), x_3)x_4), t = f(x_4, f(x_3, f(x_2, f(x_1, a))))$$

→ $\{s \doteq t\}, []$

decompose → $\{f(f(f(a, x_1), x_2), x_3) \doteq x_4, x_4 \doteq f(x_3, f(x_2, f(x_1, a)))\}, []$

substitute → $\{f(f(f(a, x_1), x_2), x_3) \doteq f(x_3, f(x_2, f(x_1, a)))\}, [x_4 \sim f(x_3, f(x_2, f(x_1, a)))]$

decompose → $\{f(f(a, x_1), x_2) \doteq x_3, x_3 \doteq f(x_2, f(x_1, a))\}, [x_4 \sim f(x_3, f(x_2, f(x_1, a)))]$

substitute → $\{f(f(a, x_1), x_2) \doteq f(x_2, f(x_1, a))\}, [x_4 \sim f(f(x_2, f(x_1, a)), f(x_2, f(x_1, a))), x_3 \sim f(x_2, f(x_1, a))]$

decompose → $\{f(a, x_1) \doteq x_2, x_2 \doteq f(x_1, a)\}, [x_4 \sim \dots, x_3 \sim f(x_2, f(x_1, a))]$

substitute → $\{f(a, x_1) \doteq f(x_1, a)\}, [x_4 \sim \dots, x_3 \sim \dots, x_2 \sim f(x_1, a)]$

decompose → $\{a \doteq x_1, x_1 \doteq a\}, [x_4 \sim \dots, x_3 \sim \dots, x_2 \sim f(x_1, a)]$

substitute → $\{a \doteq a\}, [x_4 \sim \dots, x_3 \sim \dots, x_2 \sim f(a, a), x_1 \sim a]$

redundant → $\emptyset, [x_4 \sim \dots, x_3 \sim \dots, x_2 \sim f(a, a), x_1 \sim a]$

Let $t_1 = a, t_2 = f(t_1, t_1), t_3 = f(t_2, t_2), t_4 = f(t_3, t_3)$, then the mgu of $s \doteq t$ is $[x_i \sim t_i \mid 1 \leq i \leq 4]$

Secara umum, jika s dan t adalah pasangan istilah berikut,

$$s = f(f(f(\dots f(f(a, x_1), x_2), \dots), x_{n-1}), x_n)$$

$$t = f(x_n, f(x_{n-1}, f(\dots, f(x_2, f(x_1, a)) \dots)))$$

Maka mgu dari s dan t adalah $\sigma = [x_i \sim t_i \mid 1 \leq i \leq n]$, di mana $t_1 = a$ dan $t_{i+1} = f(t_i, t_i)$. Kita memiliki $s\sigma = t\sigma = t_{n+1}$. Ukuran s dan t sama, yaitu $2n + 1$. Namun, ukuran t_n adalah $2^n - 1$.

Contoh di atas mengilustrasikan bahwa $unify(s, t)$ akan membutuhkan waktu dan ruang $O(2^n)$ untuk istilah masukan berukuran $O(n)$. Biaya tinggi ini tidak dapat dihindari jika kita menggunakan pohon untuk merepresentasikan istilah. Dengan menggunakan struktur data grafik berarah untuk istilah, subistilah yang identik dapat dibagi, sehingga kita tidak memerlukan ruang berukuran eksponensial.

Algoritma Unifikasi Waktu Hampir Linier

Corbin dan Bidoit (1983) meningkatkan algoritma unifikasi Alan Robinson menjadi algoritma kuadratik dengan menggunakan graf berarah. Ruzicka dan Privara (1988) lebih lanjut meningkatkannya menjadi algoritma yang hampir linier, dengan menggunakan struktur data union/find dan menunda pemeriksaan kemunculan di akhir unifikasi. Struktur data union/find menyimpan kumpulan himpunan yang terpisah, dan setiap himpunan mewakili kelas ekuivalensi.

Dengan demikian, himpunan yang terpisah mewakili relasi ekuivalensi dan merupakan partisi dari elemen-elemen yang terlibat. Operasi Union akan menggabungkan dua himpunan menjadi satu, dan operasi Find akan mengembalikan anggota representatif dari suatu himpunan, sehingga memungkinkan kita untuk mengetahui secara efisien apakah dua elemen berada dalam himpunan yang sama atau tidak.

Berikut ini, kami memperkenalkan algoritma unifikasi waktu hampir linier yang sedikit berbeda dari algoritma Ruzicka dan Privara.

Diberikan dua istilah s dan t , grafik istilah untuk s dan t adalah grafik asiklik terarah $G = (V, E)$ yang diperoleh dengan menggabungkan pohon rumus s dan t dengan operasi berikut: Semua simpul yang diberi label dengan variabel yang sama di s dan t digabungkan menjadi satu simpul tunggal di G .

Grafik G memiliki fitur-fitur berikut:

- Setiap simpul v dari G diberi label dengan simbol di s atau t , yang dilambangkan dengan $label(v)$.
- Akar dari pohon rumus adalah dua simpul di G dengan derajat masuk nol, yang masing-masing mewakili s dan t . Derajat masuk simpul non-variabel lainnya adalah satu. Derajat masuk dari simpul variabel adalah jumlah kemunculan variabel tersebut di s dan t .
- Simpul daun dari pohon rumus adalah simpul dengan derajat keluar nol di G , yang diberi label dengan variabel atau konstanta.
- Jika sebuah node diberi label dengan simbol fungsi f yang aritasnya k , maka node ini memiliki k penerus yang terurut. Kita akan menggunakan $child(v, i)$ untuk merujuk pada penerus ke- i dari v .

Istilah yang terkait dengan setiap node n dari G dapat didefinisikan secara rekursif sebagai berikut:

1. Jika $label(n)$ adalah variabel atau konstanta, maka $term(n) = label(n)$.
2. Jika $label(n)$ adalah fungsi f selain konstanta, maka $term(n) = f(s_1, s_2, \dots, s_k)$, di mana $s_i = term(child(n, i))$ untuk $1 \leq i \leq aritas(f)$.

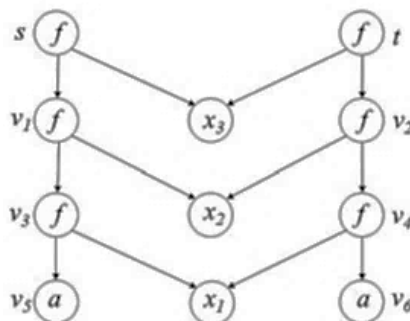
Kita katakan $term(n)$ adalah istilah yang diwakili oleh n dalam G .

Algoritma unifikasi akan bekerja pada $G = (V, E)$ dan dibagi menjadi dua tahap: Pada tahap 1, $unify_1$ memeriksa apakah s dan t tidak memiliki kegagalan benturan; pada tahap 2, $postOccurCheck$ mengembalikan $true$ jika dan hanya jika mereka tidak memiliki kegagalan pemeriksaan kejadian.

Pada tahap 1, algoritma $unify_1$ mengambil dua node yang mewakili s dan t dan mengunjungi G secara bersamaan dengan pencarian mendalam (depth-first search). Selama pencarian, kita mungkin akan menetapkan node lain u ke node v sebagai perwakilan dari v , yang dilambangkan dengan $up(v) = u$. Awalnya, $up(v) = v$ untuk setiap node G . Jika $up(v) = u$ dan $v \neq u$, itu berarti istilah yang diwakili oleh v dan u , masing-masing, kemungkinan dapat disatukan (tidak ada kegagalan benturan) dan kita menggunakan u sebagai perwakilan dari v dalam sisa pencarian.

Pada akhir pencarian, $unify_1$ mengembalikan $true$ jika tidak ditemukan kegagalan benturan. Dalam hal ini, relasi R yang didefinisikan oleh up , yaitu, $R = \{(u, up(u)) \mid u \in V\}$, dapat menghasilkan relasi ekuivalensi E (yaitu, E adalah penutupan refleksif, simetris, dan transitif dari R). Relasi E membagi V menjadi kelas-kelas ekuivalensi dan simpul-simpul dalam kelas ekuivalensi yang sama mewakili istilah-istilah yang harus dapat disatukan jika s dan t dapat disatukan.

Contoh 6.1.18 Diberikan $t = f(f(f(a, x_1), x_2), x_3)$ dan $s = f(x_3, f(x_2, f(x_1, a)))$, grafik istilahnya ditunjukkan pada Gambar 6.1. Relasi up diinisialisasi sebagai $up(v) = v$.



Gambar 6.1 Grafik suku untuk $t = f(f(f(a, x_1), x_2), x_3)$ dan $s = f(x_3, f(x_2, f(x_1, a)))$

Algoritma $unify_1(t, s)$ memulai pencarian mendalam secara tandem dan menuju ke anak pertama dari (t, s) , yaitu (v_1, x_3) . Karena x_3 adalah node variabel, $unify_1(v_1, x_3)$ akan melakukan $up(x_3) := v_1$ dan mundur. Anak kedua dari (t, s) adalah (x_3, v_2) . Karena $up(x_3) = v_1$, kita menggunakan v_1 untuk x_3 dan memanggil $unify_1(v_1, v_2)$ secara rekursif. Karena v_1 dan v_2 memiliki label yang sama, pencarian berlanjut ke anak-anak mereka. Anak pertama dari (v_1, v_2) adalah (v_3, x_2) . Karena x_2 adalah node variabel, kita melakukan $up(x_2) := v_3$ dan mundur. Anak kedua dari (v_1, v_2) adalah (x_2, v_4) . Gunakan v_3 untuk x_2 dan panggil $unify_1(v_3, v_4)$ secara rekursif. Karena v_3 dan v_4 memiliki label yang sama, pencarian berlanjut ke anak-anaknya. Anak pertama dari (v_3, v_4) adalah (v_5, x_1) . Kita melakukan $up(x_1) := v_5$ dan melakukan backtracking. Anak kedua dari (v_3, v_4) adalah (x_1, v_6) . Gunakan v_5 untuk x_1 dan panggil $unify_1(v_5, v_6)$ secara rekursif. Karena (v_5, v_6) memiliki label konstan yang sama, pencarian melakukan backtracking setelah melakukan $up(v_6) := v_5$. Sekarang, semua anak dari (v_3, v_4) telah selesai. Setelah $up(v_4) := v_3$, pencarian melakukan backtracking. Demikian pula, semua anak dari (v_1, v_2) telah selesai, dan kita melakukan backtracking setelah melakukan $up(v_2) := v_1$. Akhirnya, semua anak dari (t, s) telah selesai dan $unify_1(t, s)$ berakhir dengan $up(s) := t$.

Isi dari $up(v)$ setelah penghentian $unify_1(t, s)$ ditunjukkan di bawah ini:

Node n	s	t	v ₁	v ₂	v ₃	v ₄	v ₅	v ₆	x ₁	x ₂	x ₃
up(n)	t	t	v ₁	v ₁	v ₃	v ₃	v ₅	v ₅	v ₅	v ₃	v ₁

Relasi ekuivalensi E yang dihasilkan oleh $(v, up(v))$ dapat direpresentasikan oleh partisi V berikut:

$$\{s, t\}, \{v_1, v_2, x_3\}, \{v_3, v_4, x_2\}, \{v_5, v_6, x_1\}$$

Langkah selanjutnya adalah pemeriksaan kemunculan. Misalnya, apakah x_3 muncul dalam istilah yang diwakili oleh v_1 atau v_2 ? Prosedur *postOccurCheck* pada tahap kedua akan mengkonfirmasi bahwa jawabannya adalah “tidak.”

Jelas dari contoh di atas bahwa setiap node V awalnya berada dalam himpunan tunggal. Ketika istilah yang diwakili oleh kedua node tersebut dapat disatukan, himpunan yang berisi kedua node tersebut digabungkan menjadi satu melalui penugasan $up(v) := u$, yaitu, kita memperoleh “gabungan” dari dua himpunan yang masing-masing berisi u dan v . Untuk mengetahui node mana yang diwakili oleh node mana, kita memerlukan operasi “temukan”. Dengan demikian, kita dapat menggunakan struktur data *union/find* yang terkenal untuk mendukung implementasi *unify*₁.

Mengingat struktur data *union/find*, setiap himpunan terpisah diwakili oleh sebuah pohon: $up(v) = u$ berarti induk dari v adalah u dalam pohon jika $v \neq u$. Jika $up(u) = u$, maka u adalah akar pohon, yang juga berfungsi sebagai nama (atau perwakilan) dari himpunan tersebut. *Find(u)* akan menemukan nama himpunan yang berisi u dan akan mengompres pohon jika perlu. *Union(u, v)* akan menggabungkan dua himpunan yang namanya masing-masing adalah u dan v . Implementasi tipikal dari Union akan membiarkan akar pohon yang lebih pendek/kecil menunjuk ke akar pohon yang lebih tinggi/besar dan dapat dilakukan dalam waktu konstan.

Seperti yang dinyatakan sebelumnya, algoritma *unify*₁ mengambil dua node yang mewakili s dan t , masing-masing, dan mengunjungi G secara bersamaan dengan pencarian mendalam. Algoritma *unify*₁ pertama-tama akan mengganti dua node saat ini dengan nama himpunannya, yang juga merupakan node dalam G . Jika kedua nama himpunan tersebut sama, lakukan backtrack; jika salah satunya adalah node variabel, jadikan yang lain sebagai induknya dalam pohon gabungan/pencarian, dan lakukan penelusuran balik. Jika keduanya adalah node fungsi dan labelnya berbeda, laporkan kegagalan bentrokan; jika tidak, kunjungi anak-anaknya secara rekursif.

Jika tidak ditemukan kegagalan dari anak-anaknya, gabungkan kedua himpunan yang dinamai oleh dua node saat ini dan lakukan penelusuran balik. Selama pencarian mendalam, kami juga menggunakan mekanisme penandaan untuk mendeteksi apakah siklus dibuat karena variabel disatukan dengan suatu istilah; jika ya, terjadi kegagalan pemeriksaan kemunculan. Hanya sebagian dari kegagalan pemeriksaan kemunculan yang dapat ditemukan dengan cara ini. Kode semu algoritma *lunify* diberikan di bawah ini.

Algoritma 6.1.19 Algoritma *lunify(s, t)* mengambil istilah s dan t sebagai input, membuat grafik rumus $G = (V, E)$ dari s dan t , menggabungkan semua node dari variabel yang sama menjadi satu, dan memanggil *unify*₁(v_s, v_t), di mana v_s dan v_t adalah node yang mewakili s dan t , masing-masing. Jika *unify*₁(v_s, v_t) mengembalikan true, *postOccurCheck*(v_s, v_t) dipanggil, yang mengembalikan true jika tidak ada siklus di $G = (V, E)$ berdasarkan pencarian mendalam.

proc *lunify*(*s*, *t*): Boolean

```

1  G := createGraph(s, t) // up(v) = v for each v in G
2  if unify1(vs, vt) // vs, vt represent s, t in G
3      return postOccurCheck(vs, vt)
4  else return false

```

proc *unify*₁(*v*₁, *v*₂): Boolean

```

    // Use the union/find data structure on the term graph G.
1  s1 := Find(v1); s2 := Find(v2)
2  if s1 = s2 return true // already unified.
3  if (label(s1) and label(s2) are variables) Union(s1, s2) return true
4  if (label(s1) is variable) up(s1) := s2; return true
5  if (label(s2) is variable) up(s2) := s1; return true
6  if label(s1) ≠ label(s2) return false // clash failure
    // use mark(s) to detect if G has a cycle
7  for i := 1 to arity(label(s1)) do
8      c1 := child(s1, i); c2 := child(s2, i)
9      if (marked(c1) or marked(c2)) return false // occur – failure
10     mark(c1); mark(c2)
11     if (unify1(c1, c2) = false) return false
12     unmark(c1); unmark(c2)
13 Union(s1, s2)
14 return true

```

proc *Union*(*s*₁, *s*₂)

```

    // Initially, for any node s, h(s) = 0, the height of a singleton tree.
1  if h(s1) < h(s2) up(s1) := s2; return
2  up(s2) := s1 // the parent of s2 is s1
3  if h(s1) = h(s2) h(s1) := h(s1) + 1

```

proc *Find*(*v*)

```

1  r := up(v) // look for the root of the tree containing v
2  while (r ≠ up(r)) r := up(r) // r is root iff r = up(r)
3  if (r ≠ v) // compress the path from v to r
4  while (r ≠ v) up(v) := r; v := up(v)
5  return r

```

Seperti yang telah disebutkan sebelumnya, dalam struktur data union-find, setiap himpunan node direpresentasikan oleh sebuah pohon (didefinisikan oleh $up(v) = u$, di mana induk dari v adalah u) dan akar pohon adalah nama himpunan tersebut. Operasi union mengimplementasikan "union by height," menjadikan akar pohon yang lebih tinggi sebagai induk dari akar pohon yang lebih pendek. $Find(v)$ akan mengembalikan nama himpunan, yaitu akar pohon yang berisi v ; ini dilakukan oleh loop while pertama. Untuk efisiensi panggilan Find di masa mendatang, jalur dari v ke akar dikompresi setelah akar ditemukan; ini diselesaikan oleh loop while kedua.

Ada dua relasi induk-anak yang digunakan dalam algoritma di atas. Prosedur $unify_1$ menggunakan pencarian mendalam berdasarkan relasi induk-anak yang didefinisikan oleh istilah $graph$, bukan relasi induk-anak yang didefinisikan oleh $up(v) = u$. Yang terakhir menginduksi relasi ekuivalensi dan digunakan oleh algoritma $union/find$.

Contoh 6.1.20 Misalkan $s = f(x, x)$ dan $t = f(y, a)$. Sebuah graf $G = (V, E)$ akan dibuat dengan $V = \{v_s, v_t, v_x, v_y, v_a\}$ dan $E = \{(v_s, v_x)_1, (v_s, v_x)_2, (v_t, v_y)_1, (v_t, v_a)_2\}$, di mana indeks bawah adalah urutan anak-anaknya. Algoritma $unify_1(v_s, v_t)$ akan memanggil $unify_1(v_x, v_y)$, yang mengubah $up(v_x) = v_y$, dan $unify_1(v_x, v_a)$, yang mengubah $up(v_y) = v_a$. Setelah dua panggilan rekursif, $unify_1(v_s, v_t)$ akan mengubah $up(v_s) = v_t$ dan mengembalikan $true$. V dipartisi menjadi dua himpunan: $\{v_s, v_t\}$ dan $\{v_x, v_y, v_a\}$.

Dalam algoritma $unify_1$, graf $G = (V, E)$ tidak pernah berubah; hanya $up(v)$ yang berubah. Nilai $up(v)$ mendefinisikan relasi ekuivalensi dalam struktur data $union/find$: v dan $up(v)$ berada dalam himpunan terpisah yang sama. Nilai $up(v)$ diubah secara tidak langsung baik oleh $Find$ (dipanggil pada baris 1 dari $unify_1$; $up(v)$ diubah pada baris 4 dari $Find$), atau oleh $Union$ (dipanggil pada baris 3 dan 13; $up(v)$ dimodifikasi pada baris 1 dan 2 dari $Union$), atau secara langsung pada baris 4 dan 5 dari $unify_1$, yang merupakan kasus khusus dari $Union$: induk dari sebuah node variabel diatur menjadi node fungsi (bukan sebaliknya). Dengan demikian, algoritma hanya menggunakan ruang linier. Algoritma ini menggunakan waktu yang hampir linier karena pengamatan berikut:

- Misalkan $G = (V, E)$ adalah graf term dari s dan t , dan $n = |V|$ dibatasi oleh ukuran total s dan t .
- Jumlah operasi $Find$ yang dilakukan di $unify_1$ adalah dua kali lipat dari jumlah panggilan rekursif $unify_1$.
- Setiap operasi $Union$ mengurangi jumlah himpunan node sebanyak satu; dengan demikian, jumlah operasi $union$ dibatasi oleh n .
- Setelah dua subistilah terbukti dapat disatukan, kedua node yang mewakilinya dimasukkan ke dalam himpunan yang sama; dengan demikian, kita tidak pernah mencoba menyatukan lagi pasangan subistilah yang sama dan jumlah panggilan rekursif $unify_1$ dibatasi oleh n .
- Total biaya $Find$ dan $Union$ yang dilakukan di $unify_1$ adalah $O(n\alpha(n))$, di mana $\alpha(n)$ adalah kebalikan dari fungsi Ackermann dan tumbuh sangat lambat. Faktanya, $\alpha(n)$ praktis merupakan fungsi konstan, misalnya, $\alpha(10^9) \leq 4$.
- Gabungan khusus yang dilakukan pada baris 4 dan 5 dari $unify_1$ tidak memengaruhi kompleksitas hampir linier, karena total biaya gabungan tersebut antara simpul variabel dan non-variabel dibatasi oleh jumlah simpul variabel.

- Prosedur *postOccurCheck* akan memeriksa apakah $G' = (V, E')$, di mana $E' = E \cup \{(v, up(v)) | v \in V, v \neq up(v)\}$, mengandung siklus dan dapat diimplementasikan dalam waktu $O(n)$ dengan pencarian mendalam.

Teorema 6.1.21 Algoritma *lunify*(s, t) membutuhkan waktu $O(n\alpha(n))$, di mana n adalah ukuran total s dan t , dan $\alpha(n)$ adalah invers dari fungsi Ackermann; *lunify*(s, t) mengembalikan nilai true jika dan hanya jika s dan t dapat disatukan.

Algoritma *lunify* memberi tahu kita apakah dua istilah dapat disatukan atau tidak. Namun, ini berbeda dengan mengetahui apa sebenarnya penyatunya. Kami meninggalkan masalah berikut sebagai latihan: menemukan mgu dari s dan t setelah *lunify*(s, t) mengembalikan nilai true.

Karena *lunify* tidak mengubah grafik istilah s dan t , kita mungkin bertanya-tanya bagaimana mendapatkan istilah $t\sigma$ dari G jika σ adalah mgu dari s dan t . Istilah $t\sigma$, yang sama dengan $s\sigma$, dapat diperoleh dengan $term'(v_s)$ (atau $term'(v_t)$), di mana $term'$ dapat didefinisikan secara rekursif sebagai berikut: Untuk setiap node v dari G :

1. Jika $up(v) \neq v$, maka $term'(v) = term'(up(v))$.
2. Jika $label(v)$ adalah konstanta atau variabel dan $up(v) = v$, maka $term'(v) = label(v)$.
3. Jika $label(v)$ adalah fungsi f selain konstanta, maka $term'(v) = f(s_1, s_2, \dots, s_k)$, di mana $s_i = term'(child(v, i))$ untuk $1 \leq i \leq arity(f)$.

Mudah untuk memeriksa bahwa $term'(v) = term(v)$ sebelum eksekusi *lunify* karena pada saat itu, $up(v) = v$ untuk setiap v . Kita dapat melihat $\{(v, up(v)) | up(v) \neq v\}$ sebagai himpunan sisi baru yang ditambahkan ke $G = (V, E)$ selama eksekusi *lunify*. Himpunan sisi ini mendefinisikan tidak hanya relasi kesetaraan yang diwakili oleh struktur data *union/find*, tetapi juga mgu σ ketika s dan t dapat disatukan.

Contoh-contoh yang dibuat-buat dalam Tabel 6.1 mengilustrasikan perilaku *lunify*. Ukuran suatu istilah adalah jumlah total simbol dalam istilah tersebut, tidak termasuk tanda kurung dan koma. Kolom visited memberikan jumlah node yang dikunjungi sebelum hasil dikembalikan.

Tabel 6.1 Kolom terakhir memberikan jumlah node yang dikunjungi oleh *unify1* dalam grafik istilah untuk setiap contoh

	Pasangan Istilah	Ukuran	Kegagalan	Dikunjungi
P1	$f(f(\dots f(f(a, x_1), x_2), \dots), x_n)$ $f(x_n, f(x_{n-1}, f(\dots, f(x_1, a) \dots)))$	$4n + 2$	Tidak ada	$4n + 2$
P2	$f(z, g^n(x))$ $f(g(z), g^n(y))$	$n + 3$	Terjadi	$n + 3$
P3	$h(x, f(x, f(x, \dots, f(x, f(x, x)) \dots), x))$ $h(f(y, \dots, f(y, f(y, y)) \dots), y, y)$	$8n + 3$	Terjadi	8
P4	$f(f(\dots f(f(a, x_1), x_2), \dots), x_n)$	$4n + 2$	Bertentangan	$4n + 2$

	$f(x_n, f(x_{n-1}, f(\dots, f(x_1, b) \dots)))$			
P5	$h(x, g^n(x), x)$ $h(g^n(y), y, y)$	$2n + 8$	Terjadi	$2n + 8$
P6	$f(f(f(\dots f(f(a, x_1), x_2), \dots), x_n), z)$ $f(f(x_n, f(x_{n-1}, f(\dots, f(x_1, a) \dots))), g(z))$	$4n + 7$	Terjadi	$4n + 7$

Pada $P1$, kedua istilah tersebut dapat disatukan, dan pemeriksaan kemunculan dapat sangat mahal jika kita tidak menunda pemeriksaan tersebut. Pada $P2$, pemeriksaan kemunculan pertama (pada $z \sim g(z)$) akan gagal, meskipun *unify* tidak akan mengetahuinya sampai *postOccurCheck* dipanggil. Pada $P3$, pemeriksaan kemunculan terakhir akan gagal.

Menarik untuk diperhatikan bahwa *unify1* akan melaporkan kegagalan ini melalui mekanisme penandaan dalam jumlah langkah yang konstan, setelah hanya mengunjungi delapan node dari grafik istilah. Pada $P4$, kegagalan benturan terjadi di bagian bawah pencarian mendalam. $P5$ mirip dengan $P3$, meskipun kegagalan pemeriksaan kemunculan hanya dapat ditemukan setelah pencarian menyeluruh. $P6$ mirip dengan $P1$, meskipun pemeriksaan kemunculan terakhir akan menemukan kegagalan tersebut.

6.2 RESOLUSI

Resolusi adalah aturan inferensi logika orde pertama yang diusulkan oleh J. Alan Robinson pada tahun 1965.

Definisi Formal

Definisi 6.2.1 (*Resolusi*) Misalkan klausa c_1 adalah $(A \mid \alpha)$ dan c_2 adalah $(B \mid \beta)$, di mana A dan B adalah atom dan α dan β adalah literal sisa dalam c_1 dan c_2 , masing-masing, resolusi adalah aturan inferensi berikut:

$$\frac{(A \mid \alpha) \quad (\bar{B} \mid B)}{(\alpha \mid \beta)\sigma}$$

di mana σ adalah mgu dari A dan B . Klausa $(\alpha \mid \beta)\sigma$ yang dihasilkan oleh aturan resolusi disebut *resolven* dari resolusi; c_1 dan c_2 adalah induk dari resolven. Aturan resolusi ini juga disebut resolusi biner, karena melibatkan dua klausa sebagai premis.

Contoh 6.2.2 Diberikan sepasang klausa $(p(f(x_1)) \mid q(x_1))$ dan $(\overline{p(x_2)} \mid \overline{q(g(x_2))})$. Resolusi dapat menghasilkan dua resolven dari keduanya: Resolve pada p untuk mendapatkan $(q(x_1) \mid \overline{q(g(f(x_1)))})$ dengan mgu $[x_2 \sim f(x_1)]$ dan pada q untuk mendapatkan $(p(f(g(x_2))) \mid \overline{p(x_2)})$ dengan mgu $[x_1 \sim g(x_2)]$.

Contoh di atas menunjukkan bahwa lebih dari satu resolven dapat dihasilkan dari dua klausa. Kita akan menggunakan *resolve*(c_1, c_2) untuk menunjukkan himpunan semua resolven dari klausa c_1 dan c_2 . Dalam logika proposisional, jika dua klausa tidak memiliki literal yang duplikat dan dapat menghasilkan lebih dari satu resolven, maka resolven-resolven ini adalah tautologi.

Proposisi 6.2.3 Aturan resolusi adalah sah, yaitu, $c_1 \wedge c_2 \models \text{resolve}(c_1, c_2)$.

Bukti Misalkan c_1 adalah $(A \mid \alpha)$, c_2 adalah $(\bar{B} \mid \beta)$, dan $\text{resolve}(c_1, c_2)$ mengandung $(\alpha \vee \beta)\sigma$ dan σ adalah mgu dari A dan B . Untuk setiap model I dari $c_1 \wedge c_2$, I juga merupakan model dari $c_1\sigma = A\sigma \vee \alpha\sigma$ dan $c_2\sigma = \bar{B}\sigma \vee \beta\sigma$, karena variabel bebas diperlakukan sebagai variabel yang dikuantifikasi secara universal. Karena σ adalah penyatu A dan B , maka $A\sigma = B\sigma$. Dengan demikian, nilai kebenaran $A\sigma$ dan $\bar{B}\sigma$ berbeda di I . Jika $I(A\sigma) = 0$, maka $I(\alpha\sigma) = 1$; jika tidak, $I(c_1\sigma)$ akan menjadi 0. Jika $I(A\sigma) = 1$, maka $I(\bar{B}\sigma) = 0$, sehingga $I(\beta\sigma) = 1$. Dalam kedua kasus tersebut, baik $\alpha\sigma$ atau $\beta\sigma$ atau keduanya benar di I , sehingga $(\alpha \mid \beta)\sigma$ benar di I .

Berdasarkan proposisi di atas, setiap resolven merupakan konsekuensi logis dari klausa masukan, karena relasi implikasi bersifat transitif. Jika klausa kosong dihasilkan, maka klausa masukan tidak dapat dipenuhi. Dengan kata lain, kita dapat menggunakan aturan resolusi untuk merancang pembuktian sanggahan. Misalkan kita memiliki beberapa aksioma, katakanlah A , dan kita ingin melihat apakah beberapa dugaan, katakanlah B , merupakan konsekuensi logis dari A . Dengan strategi sanggahan, kita ingin menunjukkan bahwa $A \wedge \bar{B}$ tidak dapat dipenuhi, dengan menempatkan $A \wedge \bar{B}$ ke dalam bentuk klausa dan menemukan kontradiksi melalui resolusi. Kita menginginkan kesetaraan logis berikut berlaku.

- B adalah konsekuensi logis dari A , yaitu, $A \models B$, jika dan hanya jika
- $A \wedge \bar{B}$ tidak dapat dipenuhi, jika dan hanya jika
- S , himpunan klausa yang diturunkan dari $A \wedge \bar{B}$, tidak dapat dipenuhi, jika dan hanya jika
- Klausa kosong dapat dihasilkan melalui resolusi dari S .

Sayangnya, "jika dan hanya jika" terakhir tidak berlaku dalam logika orde pertama. Memang, ketika S dapat dipenuhi, resolusi tidak akan pernah menghasilkan klausa kosong. Jika klausa kosong dapat dihasilkan, kita dapat mengklaim bahwa S tidak dapat dipenuhi, karena resolusi itu sah. Namun, ketika S tidak dapat dipenuhi, resolusi mungkin tidak menghasilkan klausa kosong.

Contoh 6.2.4 Misalkan $S = \{(p(x_1) \mid p(y_1)), (\overline{p(x_2)} \mid \overline{p(y_2)})\}$. Salah satu resolven dari dua klausa dalam S adalah $(p(x_3) \mid \overline{p(y_3)})$, dan kita dapat menambahkannya ke S . Resolven lain dari S akan menjadi penamaan ulang dari ketiga klausa tersebut. Dengan kata lain, resolusi saja tidak dapat menghasilkan klausa kosong dari S . Di sisi lain, sebuah contoh dari klausa pertama adalah $(p(a) \mid p(a))$, yang dapat disederhanakan menjadi $(p(a))$. Dengan menambahkan klausa unit ini ke S , kita dapat dengan mudah memperoleh klausa kosong dari S .

Dalam logika proposisional, kita mendefinisikan sistem pembuktian yang menggunakan aturan resolusi sebagai satu-satunya aturan inferensi dan menunjukkan bahwa sistem pembuktian ini

adalah prosedur pengambilan keputusan untuk klausa proposisional. Dalam logika orde pertama, kita tidak seberuntung itu.

Pemfaktoran

Untuk mengatasi masalah yang diilustrasikan pada contoh sebelumnya, kita memerlukan aturan inferensi berikut:

Definisi 6.2.5 (*Pemfaktoran*) Misalkan klausa C adalah $(A \mid B \mid \alpha)$, di mana A dan B adalah literal dan α adalah literal lainnya dalam C . Pemfaktoran adalah aturan inferensi berikut:

$$\frac{(A \mid B \mid \alpha)}{(A \mid \alpha)\sigma}$$

di mana σ adalah mgu dari A dan B .

Untuk klausa $(p(x_1) \mid p(y_1))$ pada contoh sebelumnya, pemfaktoran akan menghasilkan $(p(x_1))$; untuk $(\overline{p(x_2)} \mid \overline{p(y_2)})$, pemfaktoran akan menghasilkan $(\overline{p(x_2)})$. Sekarang resolusi akan menghasilkan klausa kosong dari $(p(x_1))$ dan $(\overline{p(x_2)})$.

Proposisi 6.2.6 Aturan pemfaktoran adalah sah, yaitu, $C \models C'$, di mana C' adalah klausa yang dihasilkan dengan pemfaktoran dari C .

Bukti C' adalah contoh dari C , yaitu, $C' = C\sigma$, di mana σ adalah penyatu yang digunakan dalam pemfaktoran. Setiap contoh dari suatu rumus adalah konsekuensi logis dari rumus tersebut.

Teorema 6.2.7 (*Kelengkapan Resolusi Refutasional*) Suatu himpunan S klausa tidak dapat dipenuhi jika dan hanya jika klausa kosong dapat dihasilkan melalui resolusi dan pemfaktoran.

Meskipun pemfaktoran diperlukan untuk kelengkapan resolusi, dalam praktiknya, pemfaktoran jarang diperlukan.

Prosedur Pembuktian Refutasional

Untuk logika proposisional, kami memperkenalkan tiga strategi penghapusan untuk menghapus klausa tanpa khawatir kehilangan bukti:

- *Penghapusan literal murni*: Klausa yang mengandung literal murni dibuang.
- *Penghapusan tautologi*: Klausa tautologi dibuang.
- *Penghapusan subsumsi*: Klausa yang disubsumsi dibuang.

Penghapusan literal murni dan penghapusan tautologi dapat digunakan tanpa modifikasi. Untuk penghapusan subsumsi, aturannya dimodifikasi sebagai berikut:

Definisi 6.2.8 (*Subsumsi*) Klausa c_1 mensubsumsi klausa c_2 jika ada substitusi σ sedemikian sehingga setiap literal dari $c_1\sigma$ muncul di c_2 .

Misalnya, $(p(x, y) \mid \overline{p(y, x)})$ mensubsumsi $(p(a, b) \mid \overline{p(b, a)} \mid q(a))$ dengan $\sigma = [x \sim a, y \sim b]$. Strategi penghapusan dapat diintegrasikan ke dalam algoritma resolusi sebagai berikut.

Prosedur 6.2.9 Prosedur *resolution*(*C*) mengambil himpunan *C* klausa dan mengembalikan false jika dan hanya jika *C* tidak dapat dipenuhi. Ia menggunakan *preprocessing*(*C*) untuk menyederhanakan klausa masukan *C*. Prosedur *resolve*(α, β) mengembalikan himpunan resolven dari α dan β . Prosedur *tautology*(α) memeriksa apakah klausa α adalah tautologi. Prosedur *subsumedBy*(α, S) memeriksa apakah α tercakup dalam sebuah klausa di *S*. Prosedur *factoring*(α) menerima klausa α sebagai masukan dan menghasilkan himpunan hasil dari aturan pemfaktoran pada α , termasuk α itu sendiri.

```

proc resolution(C)
1  G := preprocessing(C) // G: klausa yang diberikan
2  K :=  $\emptyset$  // K: klausa yang dipertahankan
3  while G  $\neq \emptyset$  do
4       $\alpha$  := pickClause(G) // heuristik untuk memilih klausa
5      G := G -  $\{\alpha\}$ 
6      N :=  $\emptyset$  // klausa baru dari  $\alpha$  dan K melalui resolusi
7      for  $\beta \in K$  if resolvable( $\alpha, \beta$ ) do
8          for  $\gamma \in$  resolve( $\alpha, \beta$ ) do
9              if  $\gamma = ()$  return false // klausa kosong ditemukan
10             if tautology( $\gamma$ )  $\vee$  subsumedBy( $\gamma, G \cup K$ ) continue
11             N := N  $\cup$  factoring( $\gamma$ )
12      K := K  $\cup \{\alpha\}$ 
13      for  $\alpha \in G$  if subsumedBy( $\alpha, N$ ) do G := G -  $\{\alpha\}$ 
14      for  $\beta \in K$  if subsumedBy( $\beta, N$ ) do K := K -  $\{\beta\}$ 
15      G := G  $\cup N$ 
16  return true // K jenuh oleh resolusi.

```

Prosedur di atas hampir identik dengan Algoritma 3.3.27 dengan tiga perbedaan: (a) *resolve*(α, β) mengembalikan himpunan resolven alih-alih resolven tunggal (baris 8); (b) faktorisasi digunakan (baris 11); (c) penghentian prosedur tidak dijamin karena dapat menghasilkan sejumlah klausa baru yang tak terbatas. Dengan demikian, ini bukan algoritma. Meskipun demikian, prosedur ini merupakan dasar bagi banyak pembuktian teorema resolusi, termasuk Otter McCune dan penerusnya Prover9, yang akan diperkenalkan sebentar lagi. Dalam Prover9, pemeriksaan subsumsi pada baris 10 disebut *subsumsi* maju dan pemeriksaan subsumsi pada baris 13 dan 14 disebut *subsumsi* mundur. Prover9 juga menggunakan aturan penyederhanaan lain yang disebut penghapusan unit.

Definisi 6.2.10 (Penghapusan Unit) Penghapusan unit adalah aturan penyederhanaan yang menghapus literal B dari klausa α jika ada klausa unit (A) dan substitusi σ sedemikian sehingga $A\sigma = \bar{B}$ (atau $\bar{A}\sigma = B$).

Penghapusan unit sah karena berdasarkan persyaratan, ada resolusi antara (A) dan α dan resolven dari resolusi ini akan mencakup α . Misalnya, jika kita memiliki klausa unit $(\overline{p(x,b)})$, maka literal $p(a,b)$ dalam $(p(a,b) \mid q(a))$ dapat dihapus untuk mendapatkan klausa baru $(q(a))$, yang mencakup $(p(a,b) \mid q(a))$. Penghapusan unit dapat digunakan pada baris 10, sehingga klausa baru γ disederhanakan oleh klausa unit dalam $G \cup K$. Ini disebut penghapusan unit maju. Penghapusan unit mundur dapat disisipkan pada baris 13 dan 14 ketika klausa unit dalam N digunakan untuk menyederhanakan klausa dalam $G \cup K$.

6.3 URUTAN PENYEDERHANAAN DAN RESOLUSI TERURUT

Pada Bagian 3.3.2, kita telah mencantumkan beberapa strategi resolusi yang terkenal yang menambahkan batasan pada penggunaan aturan resolusi: resolusi unit, resolusi input, resolusi terurut, resolusi positif, resolusi negatif, himpunan pendukung, dan resolusi linier. Strategi-strategi ini dapat diterapkan pada logika orde pertama tanpa modifikasi, kecuali resolusi terurut. Prosedur *resolvable*(A, B) (baris 7 dari *resolusi*) adalah tempat berbagai batasan diimplementasikan sehingga kita dapat menggunakan strategi resolusi terbatas ini.

Untuk resolusi unit dan resolusi input, secara umum masih belum lengkap, tetapi setara dalam arti bahwa jika ada bukti resolusi untuk satu strategi, maka ada bukti resolusi untuk strategi lainnya.

Untuk menggunakan resolusi terurut dalam logika orde pertama, kita membutuhkan konsep urutan penyederhanaan, yang merupakan urutan parsial yang berdasar pada istilah-istilah.

Urutan Parsial yang Terdefinisi dengan Baik

Ingat kembali bahwa urutan parsial adalah relasi biner antisimetris dan transitif atas himpunan S elemen, dan terdefinisi dengan baik jika tidak ada urutan tak hingga elemen berbeda $x_1, x_2, \dots, x_i, \dots$, sedemikian sehingga

$$x_1 \geq x_2 \geq \dots \geq x_i \geq \dots$$

Kita akan menggunakan $>$ dan $\geq$ sesuai kebutuhan, di mana $x > y$ jika dan hanya jika $x \geq y$ dan $x \neq y$. Kita menulis $x < y$ jika $y > x$.

Alasan kita berpegang pada urutan yang terdefinisi dengan baik adalah karena urutan yang terdefinisi dengan baik memungkinkan kita untuk menggunakan induksi matematika. Misalkan S adalah himpunan dengan urutan yang terdefinisi dengan baik $>$. Untuk menunjukkan bahwa suatu sifat $P(x)$ berlaku untuk setiap $x \in S$, kita perlu melakukan dua hal:

- Untuk setiap elemen minimal $m \in S$, $P(m)$ benar.
- Buktikan bahwa $P(x)$ benar, dengan asumsi $P(x')$ benar untuk setiap $x' < x$.

Hal di atas disebut prinsip induksi yang terdefinisi dengan baik.

Ada banyak urutan yang terdefinisi dengan baik pada himpunan yang berbeda. Misalnya, $\geq$ pada himpunan bilangan asli adalah terdefinisi dengan baik; $\geq$ pada himpunan bilangan bulat bukanlah terdefinisi dengan baik. Setiap urutan parsial pada himpunan hingga adalah terdefinisi dengan baik.

Perluasan Multiset

Ada banyak cara untuk memperoleh urutan yang ditemukan dengan baik. *Multiset* atau kantung atas himpunan S elemen adalah modifikasi dari konsep himpunan bagian dari S , yang, tidak seperti himpunan, memungkinkan beberapa contoh untuk setiap elemennya dari S . Operasi gabungan, irisan, dan pengurangan dapat diperluas ke multiset. Jika ada urutan parsial ketat $>$ atas himpunan S , kita dapat memperluas $>$ menjadi $>^{mul}$ untuk membandingkan multiset dari S :

Definisi 6.3.1 Untuk setiap multiset hingga S_1, S_2 atas S , $S_1 >^{mul} S_2$ jika $S_1 - S_2 \neq \emptyset$, dan untuk setiap $y \in S_2 - S_1$, ada $x \in S_1 - S_2$, $x > y$.

Kondisi "hingga" diperlukan. Misalnya, jika S_1 dan S_2 adalah himpunan bilangan asli ganjil dan genap, masing-masing, maka kita memiliki $S_1 >^{mul} S_2$ dan $S_2 >^{mul} S_1$. Dengan kata lain, $>^{mul}$ tidak antisimetris atas multiset tak hingga.

Contoh 6.3.2 Misalkan $S_1 = \{a, b, b, c\}$ dan $S_2 = \{a, a, b, c\}$, maka $S_1 \cup S_2 = \{a, a, a, b, b, b, c, c\}$, $S_1 \cap S_2 = \{a, b, c\}$, $S_1 - S_2 = \{b\}$, dan $S_2 - S_1 = \{a\}$. Jika $c > b > a$, maka $S_1 >^{mul} S_2$ karena $b > a$.

Ingatlah bahwa suatu urutan yang baik bersifat total dan terdefinisi dengan baik.

Proposisi 6.3.3 Jika $>$ adalah suatu urutan yang baik atas suatu himpunan S , maka $>^{mul}$ atas himpunan-himpunan multiset dari S adalah suatu urutan yang baik.

Ekstensi Leksikografis

Salah satu cara yang populer adalah dengan menggunakan kombinasi leksikografis atas hasil perkalian Kartesius (Bagian 1.3) dari urutan yang terdefinisi dengan baik dari himpunan-himpunan individual.

Definisi 6.3.4 Misalkan $>_i$ adalah urutan parsial atas himpunan S_i , $1 \leq i \leq n$. Urutan leksikografis $>^{lex}$ atas $S_1 \times S_2 \times \dots \times S_n$ didefinisikan sebagai berikut: Untuk setiap $(x_1, x_2, \dots, x_n), (y_1, y_2, \dots, y_n) \in S_1 \times S_2 \times \dots \times S_n$, $(x_1, x_2, \dots, x_n) >^{lex} (y_1, y_2, \dots, y_n)$ jika dan hanya jika ada k , $1 \leq k \leq n$, sedemikian sehingga $x_i \geq_i y_i$ untuk $1 \leq i < k$ dan $x_k >_k y_k$.

Proposisi 6.3.5 Jika $>_i$ adalah urutan baik atas S_i , $1 \leq i \leq n$, maka urutan leksikografis $>^{lex}$ adalah urutan baik atas $S_1 \times S_2 \times \cdots \times S_n$.

Kombinasi leksikografis dapat diperluas untuk membandingkan dua urutan terbatas dengan panjang yang berbeda, dengan asumsi urutan yang tidak kosong lebih besar daripada urutan kosong, seperti urutan kamus yang membandingkan dua kata dengan berbagai panjang.

Prioritas

Saat menggunakan resolusi terurut, pengguna seringkali diharuskan untuk memberikan urutan parsial, yang disebut prioritas, atas simbol fungsi atau predikat. Diberikan bahasa orde pertama $L = (P, F, X, Op)$, misalkan $\Sigma = P \cup F \cup Op$. Kita katakan $>$ adalah prioritas jika $>$ adalah urutan baik atas Σ . Perhatikan bahwa jika Σ terbatas, $>$ secara trivial terdefinisi dengan baik.

Untuk beberapa aplikasi, kita mungkin perlu mendefinisikan kesetaraan $\approx$ atas suatu himpunan. Misalnya, kita dapat mengizinkan $\vee \approx \wedge$ dalam suatu prioritas. Menurut definisi, urutan parsial adalah relasi antisimetris dan transitif. Mengizinkan $\approx$ lebih dari $=$ melanggar sifat antisimetri, yang mensyaratkan $s = t$ jika $s \geq t$ dan $t \geq s$. Dalam literatur, jika $s \geq t$ dan $t \geq s$ menyiratkan $s \approx t$, disebut "urutan semu". Urutan semu $\geq'$ dapat diperoleh dari urutan parsial dengan menambahkan $\approx$, yaitu, misalkan $\geq' = \geq \cup \approx$. Dengan kata lain, $a \geq' b$ jika dan hanya jika $a \geq b$ atau $a \approx b$.

Proposisi 6.3.6 Jika urutan parsial terdefinisi dengan baik dan setiap kelas elemen yang setara di bawah $\approx$ adalah terbatas, maka urutan semu $\geq' = \geq \cup \approx$ terdefinisi dengan baik.

Karena quasi-order mewarisi sebagian besar sifat dari partial order, kita akan memperlakukan partial order dan quasi-order dengan kelas ekuivalen terbatas secara berbeda.

Order Penyederhanaan

Terdapat beberapa persyaratan pada order yang digunakan untuk resolusi terurut dan kita menyebutnya "order penyederhanaan."

Definisi 6.3.7 (Relasi Stabil dan Monotonik) Suatu relasi biner R atas S , di mana S menunjukkan himpunan istilah, atom, dan klausa, dikatakan stabil jika untuk setiap $s, t \in S$, $R(s, t)$ menyiratkan $R(s\sigma, t\sigma)$ untuk setiap substitusi $\sigma: X \rightarrow T(F, X)$. R dikatakan monotonik jika $R(s, t)$ menyiratkan $R(f(\dots, s, \dots), f(\dots, t, \dots))$ untuk setiap simbol f , di mana f adalah fungsi atau simbol predikat atau operator Boolean.

Untuk urutan stabil atas suku-suku, kita tidak dapat memiliki $f(y) > x$ karena $f(y)\sigma > x\sigma$ tidak berlaku untuk $\sigma = [x \sim f(y)]$.

Definisi 6.3.8 (*Urutan Penyederhanaan*) Suatu urutan parsial $\geq$ atas suku-suku atau klausa disebut urutan penyederhanaan jika urutan tersebut berdasar, stabil, dan monoton.

Dalam praktiknya, urutan penyederhanaan sering kali dibangun dari prioritas atas simbol fungsi dan predikat. Kita dapat memperluas prioritas apa pun atas Σ ke istilah, atom, dan klausa, dengan beberapa cara berbeda.

- LPO (Lexicographic Path Order). Urutan istilah ditentukan sepenuhnya oleh prioritas simbol.
- RPO (Recursive Path Order). Seperti LPO, urutan ini diinduksi dari prioritas.
- KBO (Knuth–Bendix Order). Urutan ini menggunakan fungsi pembobotan pada simbol serta prioritas simbol. Fungsi pembobotan digunakan terlebih dahulu, dan prioritas simbol digunakan untuk memecahkan kebuntuan.

LPO: Lexicographic Path Order

Untuk setiap istilah atau klausa t , misalkan $var(t)$ menunjukkan himpunan variabel yang muncul dalam t .

Definisi 6.3.9 ($>_{lpo}$) Urutan jalur leksikografis $>_{lpo}$ didefinisikan secara rekursif sebagai berikut: $s >_{lpo} t$ jika $s \neq t$ dan salah satu dari (a) $t \in var(s)$ atau (b) $s = f(s_1, \dots, s_m) >_{lpo} t = g(t_1, \dots, t_n)$ dengan salah satu kondisi berikut:

1. $f > g$ dan $s >_{lpo} t_i$ untuk $1 \leq i \leq n$, atau
2. $f \approx g$, $[s_1, \dots, s_m] >_{lex} [t_1, \dots, t_n]$, dan $s >_{lpo} t_i$ untuk $1 \leq i \leq n$, atau
3. $f < g$ dan terdapat $j, 1 \leq j \leq m, s_j \geq_{lpo} t$.

Contoh 6.3.10 Misalkan $*$ $>$ $+$, s adalah $x * (y + z)$ dan t adalah $(x * y) + (x * z)$. Maka $s >_{lpo} t$ berdasarkan kondisi (b)1, karena $*$ $>$ $+$, $s >_{lpo} x * y$ dan $s >_{lpo} x * z$, keduanya berdasarkan kondisi (b)2. Artinya, dari $[x, (y + z)] >_{lpo}^{lex} [x, y]$ dan $[x, (y + z)] >_{lpo}^{lex} [x, z]$.

Proposisi 6.3.11 $>_{lpo}$ adalah urutan penyederhanaan dan bersifat total pada istilah dasar, jika prioritas $>$ adalah urutan yang baik.

Pembuktiannya diserahkan sebagai latihan.

RPO: Urutan Jalur Rekursif

Definisi 6.3.12 ($>_{rpo}$) Urutan jalur rekursif $>_{rpo}$ atas $T(F, X)$ didefinisikan secara rekursif sebagai berikut: $s >_{rpo} t$ jika $s \neq t$ dan salah satu dari (a) $t \in var(s)$ atau (b) $s = f(s_1, \dots, s_m) >_{rpo} t = g(t_1, \dots, t_n)$ dengan salah satu kondisi berikut:

1. $f > g$ dan $s >_{rpo} t_i$ untuk $1 \leq i \leq n$, atau

2. $f \approx g$ dan $\{s_1, \dots, s_m\} >_{rpo}^{mul} \{t_1, \dots, t_n\}$, atau
3. $f < g$ dan terdapat $j, 1 \leq j \leq m, s_j \geq_{rpo} t$.

Contoh 6.3.13 Misalkan s adalah $i(x * y)$ dan t adalah $i(y) * i(x)$ dengan $i > *$. Untuk menunjukkan $s >_{rpo} t$, kondisi (b)1 berlaku karena $i > *$, dan kita perlu menunjukkan $s >_{rpo} i(x)$ dan $s >_{rpo} i(y)$. Untuk menunjukkan $s >_{rpo} i(x)$, kondisi (b)2 berlaku dan kita perlu $x * y >_{rpo}^{mul} x$, yang berlaku berdasarkan (a). Bukti $s >_{rpo} i(y)$ identik dengan bukti $s >_{rpo} i(x)$. Sebagai perbandingan, $s >_{lpo} t$ berlaku dengan bukti yang sama.

Untuk menunjukkan $t >_{rpo} s$, kita memerlukan prioritas baru di mana $* > i$ dan menggunakan $\{i(x), i(y)\} >_{rpo}^{mul} \{x, y\}$ (yang benar). Sebagai perbandingan, kita tidak memiliki $t >_{lpo} s$ dengan $* > i$, karena $[i(y), i(x)] >_{lpo}^{lex} [x, y]$ tidak benar.

Proposisi 6.3.14 $>_{rpo}$ adalah urutan penyederhanaan, jika prioritas $>$ adalah urutan total yang berdasar.

Contoh 6.3.15 Misalkan $s = (x * y) * z$ dan $t = x * (y * z)$, maka $s >_{lpo} t$ karena kondisi (b)2: $[x * y, z] >_{rpo}^{lex} [x, y * z]$. Namun, $>_{rpo}$ tidak dapat membandingkan kedua istilah ini, karena $\{x * y, z\}$ dan $\{x, y * z\}$ tidak dapat dibandingkan dengan $>_{rpo}^{mul}$. Bahkan, $>_{rpo}$ tidak dapat membandingkan $(a * a) * a$ dan $a * (a * a)$, di mana a adalah konstanta. Jadi, secara umum, $>_{rpo}$ bukanlah urutan total pada istilah dasar.

Mari kita definisikan $s \approx_{rpo} t$ jika $s = t$ atau $s = f(s_1, \dots, s_m), t = f(t_1, \dots, t_n), f \approx g$ dan $\{s_1, \dots, s_m\} \approx_{rpo}^{mul} \{t_1, \dots, t_n\}$, dan $s \geq_{rpo} t$ jika dan hanya jika $s >_{rpo} t$ atau $s \approx_{rpo} t$. Misalnya, $(a * a) * a \approx_{rpo} a * (a * a)$. Jelas, perluasan $\geq_{rpo}$ ini hanya memberi kita kelas kesetaraan terbatas dan mempertahankan keteraturan. Perluasan ini akan memungkinkan kita untuk membandingkan lebih banyak pasangan istilah daripada sebelumnya dan membuat $>_{rpo}$ lebih berguna dalam praktik. Misalnya, ini akan memungkinkan kita untuk menunjukkan $f(b, (a * a) * a) >_{rpo} f(a, a * (a * a))$ ketika $b > a$.

Relasi $\approx_{rpo}$ juga memungkinkan kita untuk menggabungkan $>_{rpo}$ dengan urutan penyederhanaan lain, misalnya $>_{lpo}$: Ketika $s \approx_{rpo} t$ dan $s \neq t$, kita menggunakan hasil dari $s >_{lpo} t$ untuk $s >_{rpo} t$. Ide ini membuat $>_{rpo}$ total pada istilah dasar.

Perbedaan $>_{lpo}$ dan $>_{rpo}$ terletak pada kondisi (b)2 dari LPO dan RPO ketika $f \approx g$: Yang pertama menggunakan perluasan leksikografis dan yang terakhir menggunakan perluasan multiset.

Definisi 6.3.16 LPO dan RPO digabungkan menjadi satu urutan penyederhanaan $>_{lrpo}$ berdasarkan status simbol fungsi: Untuk setiap fungsi f , $aritas(f) >$

1: $status(f) \in \{l, r, m\}$. Jika $f \approx g$, maka $status(f) = status(g)$. Kondisi (b)2 dari LPO (atau RPO) digantikan oleh kondisi berikut:

- $status(f) = l$: Ketika $f \approx g$, kondisi (b)2 dari LPO digunakan.
- $status(f) = r$: Ketika $f \approx g$, kondisi (b)2 dari LPO digunakan dan perluasan leksikografis berlaku pada argumen f dan g dari kanan ke kiri.
- $status(f) = m$: Ketika $f \approx g$, kondisi (b)2 dari RPO digunakan.

Penggunaan status memungkinkan kita untuk membandingkan lebih banyak pasangan istilah dalam arah yang diinginkan. Misalnya, agar $sub(x, s(y)) >_{lrpo} sub(pre(x), y)$, kita misalkan $status(sub) = r$; sehingga, $[x, s(y)] >_{lrpo}^{lex} [pre(x), y]$.

KBO: Urutan Knuth–Bendix

Urutan Knuth–Bendix ($>_{kbo}$), yang diciptakan oleh Donald Knuth dan Peter Bendix pada tahun 1970, memberikan angka, yang disebut bobot, kepada setiap fungsi, predikat, atau simbol variabel, dan menggunakan jumlah bobot untuk membandingkan istilah. Ketika dua istilah memiliki bobot yang sama, relasi prioritas digunakan untuk memecahkan kebuntuan. Penggunaan bobot memberi kita fleksibilitas sekaligus kerumitan.

Definisi 6.3.17 (Fungsi Bobot) Suatu fungsi $w: X \cup \Sigma \rightarrow N$ dikatakan sebagai fungsi bobot jika memenuhi (i) $w(c) > 0$ untuk setiap konstanta $t \in \Sigma$, dan (ii) terdapat paling banyak satu simbol fungsi uner $f \in \Sigma$ sedemikian sehingga $w(f) = 0$ dan f harus maksimal dalam prioritas jika $w(f) = 0$.

Fungsi bobot w dapat diperluas menjadi fungsi atas suku atau rumus sebagai berikut:

$$w(x) = w_0 \text{ for any variable } x;$$

$$w(f(t_1, \dots, t_n)) = w(f) + w(t_1) + \dots + w(t_n)$$

di mana $w_0 > 0$ menunjukkan bobot minimal dari semua konstanta.

Definisi 6.3.18 ($>_{kbo}$) Misalkan w adalah fungsi bobot dan $mv(t)$ adalah multiset variabel yang muncul dalam t . Orde Knuth–Bendix $>_{kbo}$ didefinisikan secara rekursif sebagai berikut: $s >_{kbo} t$ jika $w(s) \geq w(t)$ dan $mv(s) \supsetneq mv(t)$; $s >_{kbo} t$ jika $s \geq_{kbo} t$ dan salah satu dari (a) $w(s) > w(t)$ atau (b) $s \neq t$ dan t adalah variabel, atau (c) $s = f(s_1, \dots, s_m)$, $t = g(t_1, \dots, t_n)$, dan salah satu dari dua kondisi berikut benar:

1. $f > g$, atau
2. $f \approx g$ dan $[s_1, \dots, s_m] >_{kbo}^{lex} [t_1, \dots, t_n]$.

Kami ingin menunjukkan bahwa kondisi $mv(s) \supsetneq mv(t)$ adalah syarat yang diperlukan. Misalkan $s = f(g(x), y)$ dan $t = f(y, y)$. Jika $w(g) > 0$, maka $w(s) > w(t)$, tetapi $mv(s) =$

$\{x, y\} \supseteq \{y, y\} = mv(t)$ tidak berlaku. Jika kita mengabaikan kondisi $mv(s) \supseteq mv(t)$, maka $s >_{kbo} t$ akan menyiratkan $s\theta >_{kbo} t\theta$ untuk $\theta = [y \sim g(x)]$, di mana $s\theta = t\theta = f(g(x), g(x))$.

Jika $w(s) = w(t)$, maka kondisi (b), yaitu $s \neq t$ dan t adalah variabel, menyiratkan bahwa $s = f^i(x)$ dan $t = x$, di mana $i > 0, w(f) = 0$. Salah satu alasan kita membutuhkan fungsi dengan bobot 0 adalah karena kita menginginkan $i(x * y) >_{kbo} i(x) * i(y)$ (lihat Contoh 6.3.13) dan hal itu dapat dilakukan dengan $w(i) = 0$ dan $i > *$.

Contoh 6.3.19 Misalkan $w(f) = w(a) = 1, w(b) = 2$, maka $f(x, b, a) >_{kbo} f(a, a, b)$ berdasarkan kondisi (c)2, karena $x \geq_{kbo} a$ dan $b >_{kbo} a$. Demikian pula, $p(x, b, y) >_{kbo} p(a, a, y)$. Perhatikan bahwa baik $>_{lpo}$ maupun $>_{rpo}$ tidak dapat membandingkan $p(x, b, a)$ dan $p(a, a, b)$, atau $p(x, b, y)$ dan $p(a, a, y)$.

Proposisi 6.3.20 $>_{kbo}$ adalah urutan penyederhanaan dan bersifat total pada suku dasar, jika preseden $>$ adalah urutan yang baik.

Contoh 6.3.21 Misalkan relasi presedennya adalah $b/1 > a/1 > e/0$, di mana a dan b adalah simbol fungsi uner dan e adalah konstanta. Kita akan menulis $a(a(b(a(e))))$ sebagai $aaba$ untuk singkatnya. Misalkan $w(e) = 1$ dan bobot a dan b , serta 13 suku terkecil, kecuali baris terakhir, tercantum (dalam urutan $<_{kbo}$) dalam tabel berikut:

$w(a)$	$w(b)$	The least 13 terms in the order of $<_{kbo}$
1	1	$e, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, \dots$
2	1	$e, b, a, bb, ab, ba, bbb, aa, abb, bab, bba, bbbb, abb, \dots$
2	3	$e, a, b, aa, ab, ba, aaa, bb, aab, aba, baa, aaaa, abb, \dots$
2	0	$e, b, bb, \dots, b^i, \dots, a, ab, \dots, ab^i, \dots, ba, bab, babb, \dots$

Di sisi lain, daftar istilah dalam urutan menaik $>_{lpo}$ (atau $>_{rpo}$) adalah

$$e, a, \dots, a^i, \dots, b, ab, \dots, a^i b, \dots, ba, aba, \dots, a^i ba, \dots, \\ baa, abaa, \dots, a^i baa, \dots$$

Perhatikan bahwa semua urutan yang diberikan oleh KBO, LPO, atau RPO ini beralasan.

Contoh di atas menunjukkan fleksibilitas $>_{kbo}$, yang menyediakan berbagai cara untuk membandingkan dua suku dengan mengubah fungsi bobot sesuai kebutuhan. Namun, $>_{kbo}$ kurang fleksibel dalam membandingkan suku seperti $x * (y + z)$ dan $x * y + x * z$. Kita tidak dapat membuat $x * (y + z) >_{kbo} x * y + x * z$, sedangkan $x * (y + z) >_{lpo} x * y + x * z$ jika $+ >$ dan $x * y + x * z >_{lpo} x * (y + z)$ jika $+ > *$.

Sekarang, ketika kita mengatakan untuk menggunakan urutan penyederhanaan $>$, kita dapat menggunakan $>_{lpo}$, $>_{rpo}$, atau $>_{kbo}$.

Contoh 6.3.22 Misalkan operator $\leftrightarrow, \oplus, \rightarrow, \neg, \vee, \wedge$ diurutkan menurun berdasarkan prioritas $>$, maka sisi kanan dari setiap relasi ekuivalensi berikut lebih kecil dari sisi kiri yang bersesuaian sebesar $>_{lpo}$ atau $>_{rpo}$.

$$\begin{aligned} A \oplus B &\equiv (A \vee B) \wedge (\neg A \vee \neg B); \\ A \leftrightarrow B &\equiv (A \vee \neg B) \wedge (\neg A \vee B); \\ A \rightarrow B &\equiv \neg A \vee B; \\ \neg \neg A &\equiv A; \\ \neg(A \vee B) &\equiv \neg A \wedge \neg B; \\ \neg(A \wedge B) &\equiv \neg A \vee \neg B; \\ A \vee (B \wedge C) &\equiv (A \vee B) \wedge (A \vee C) \end{aligned}$$

Relasi kesetaraan ini digunakan untuk menerjemahkan sebuah formula ke dalam CNF. Dengan demikian, urutan istilah seperti $>_{rpo}$ dan $>_{lpo}$ dapat digunakan untuk menunjukkan penghentian proses konversi formula ke dalam CNF.

Jika kita ingin mengkonversi formula ke dalam DNF, maka relasi prioritasnya harus $\wedge > \vee$, sehingga penghentian konversi formula ke dalam DNF dapat ditunjukkan oleh $>_{rpo}$ atau $>_{lpo}$.

Kelengkapan Resolusi Terurut

Misalkan $>$ adalah urutan penyederhanaan, yang merupakan urutan parsial yang mapan dan stabil atas himpunan atom dan total atas himpunan atom dasar, dan kita akan menggunakan $>$ dalam resolusi terurut. Diberikan $>$, sebuah atom $A \in S$ dikatakan maksimal jika tidak ada $B \in S$ sedemikian sehingga $B > A$. A dikatakan minimal jika tidak ada $B \in S$ sedemikian sehingga $A > B$. S dapat memiliki beberapa atom minimal atau maksimal. Bahkan, sebuah atom A dapat bersifat minimal dan maksimal dalam S jika A tidak dapat dibandingkan dengan $>$ dengan atom lain dari S . Literal dibandingkan dengan membandingkan atom-atomnya. Karena sebuah klausa diwakili oleh himpunan literal, kita dapat mengatakan sebuah literal bersifat maksimal atau minimal dalam sebuah klausa. Misalnya, tiga atom dalam $(\overline{p(x,y)} \mid \overline{p(y,z)} \mid p(x,z))$ tidak dapat dibandingkan di bawah $>$ stabil apa pun, sehingga setiap atom bersifat minimal dan maksimal.

Definisi 6.3.23 (Resolusi Terurut) Diberikan urutan penyederhanaan $>$ atas himpunan atom, resolusi terurut adalah resolusi dengan kondisi bahwa A adalah atom maksimal dalam $(A \mid \alpha)$ dan B adalah atom maksimal dalam $(\bar{B} \mid \beta)$, di mana σ adalah mgu dari A dan B :

$$\frac{(A \mid \alpha) \quad (\bar{B} \mid \beta)}{(\alpha \mid \beta)\sigma}$$

Diberikan himpunan S klausa, misalkan S^* menyatakan himpunan klausa yang dijenuhkan oleh resolusi terurut dan pempfaktoran. Artinya, setiap resolven dari resolusi terurut pada dua

klausa mana pun dari S^* berada di S^* dan setiap klausa dari pemfaktoran pada klausa mana pun di S^* juga berada di S^* . Misalkan GC adalah himpunan semua contoh dasar dari S^* .

Lemma 6.3.24 GC dijenuhkan oleh resolusi terurut.

Bukti Misalkan $(g \mid \alpha)$ dan $(\bar{g} \mid \beta)$ berada di GC , di mana g adalah atom maksimal dalam kedua klausa. Resolven dari kedua klausa tersebut adalah $(\alpha \mid \beta)$. Kita perlu menunjukkan bahwa $(\alpha \mid \beta) \in GC$.

Misalkan $(g \mid \alpha)$ adalah contoh dari $(A \mid \alpha') \in S^*$, yaitu, $g = A\theta$ dan $\alpha = \alpha'\theta$ untuk beberapa substitusi θ . Kita mengasumsikan bahwa A adalah satu-satunya literal dalam $(A \mid \alpha')$ sedemikian sehingga $A\alpha = g$; jika tidak, kita menggunakan pemfaktoran untuk mencapai kondisi ini. Demikian pula, misalkan $(\bar{g} \mid \beta)$ adalah sebuah contoh dari $(\bar{B} \mid \beta') \in S^*$, yaitu, $\bar{g} = \bar{B}\gamma$ dan $\beta = \beta'\gamma$ untuk beberapa γ , dan B adalah satu-satunya literal dalam $(\bar{B} \mid \beta')$ sedemikian sehingga $B\gamma = g$.

Karena $g = A\theta = B\gamma$ adalah maksimal, A dan B harus dapat disatukan dan maksimal dalam $(A \mid \alpha')$ dan $(\bar{B} \mid \beta')$, masing-masing. Misalkan λ adalah mgu dari A dan B , maka resolven dari $(A \mid \alpha')$ dan $(\bar{B} \mid \beta')$ dengan resolusi terurut adalah $(\alpha' \mid \beta')\lambda$, yang harus berada di S^* . Karena $\theta = \lambda\theta'$ dan $\gamma = \lambda\gamma'$ untuk beberapa θ' dan γ' , kita harus memiliki $(\alpha \mid \beta) = (\alpha' \mid \beta')\lambda\theta'\gamma'$, yang merupakan contoh dari $(\alpha' \mid \beta')\lambda$. Oleh karena itu, $(\alpha \mid \beta) \in GC$.

Teorema 6.3.25 (Kelengkapan Refutasiional Resolusi Terurut) S tidak dapat dipenuhi jika dan hanya jika klausa kosong berada di S^* .

Bukti Jika klausa kosong berada di S^* , karena $S \models \perp$, S harus tidak dapat dipenuhi.

Jika klausa kosong tidak ada di S^* , kita akan membangun model Herbrand untuk S dari GC , yang merupakan himpunan semua contoh dasar S^* .

Misalkan GA adalah himpunan atom dasar yang muncul di GC . Klausa kosong tidak ada di GC karena setiap contoh klausa yang tidak kosong tidak mungkin kosong. Berdasarkan asumsi, $>$ adalah urutan yang baik di atas GA .

Kita mulai dengan model kosong H di mana tidak ada atom yang memiliki nilai kebenaran. Kemudian kita menambahkan literal dasar ke dalam H satu per satu, dimulai dari atom minimal, yang ada karena $>$ adalah urutan yang baik. Misalkan g adalah atom apa pun di GA , dan

$$\begin{aligned} pc(g) &= \{(g \mid \alpha) : g \text{ is maximal in } (g \mid \alpha) \in G\} \\ nc(g) &= \{(\bar{g} \mid \beta) : g \text{ is maximal in } (\bar{g} \mid \beta) \in G\} \\ upto(g) &= \bigcup_{g \in GA, g \geq g'} (pc(g) \cup nc(g')) \end{aligned}$$

Artinya, $pc(g)$ berisi semua klausa yang mengandung g sebagai literal maksimalnya; $nc(g)$ berisi semua klausa yang mengandung $\bar{g}$ sebagai literal maksimalnya; dan $upto(g)$ berisi semua klausa dari GC yang literal maksimalnya kurang dari atau sama dengan g . Kami mengklaim bahwa setelah g atau $\bar{g}$ ditambahkan ke H , sifat berikut ini benar:

Klaim: H adalah model dari $upto(g)$ setelah g atau $\bar{g}$ ditambahkan ke H .

Misalkan g_1 adalah atom minimal dari GA . Kita tidak dapat memiliki (g_1) dan $(\bar{g}_1)$ di GC ; jika tidak, klausa kosong pasti telah dihasilkan dari (g_1) dan $(\bar{g}_1)$ karena GC jenuh. Jika $(g_1) \in GC$, kita menambahkan g_1 ke H (artinya $H(g_1) = 1$); jika tidak, kita menambahkan $\bar{g}_1$ ke H (artinya $H(g_1) = 0$). Sangat mudah untuk memeriksa bahwa H adalah model dari $upto(g_1)$ (yang mungkin kosong), sehingga klaim tersebut benar untuk g_1 . Ini adalah kasus dasar induksi berdasarkan $>$.

Misalkan g adalah atom minimal dalam GA yang tidak memiliki nilai kebenaran dalam H . Asumsikan sebagai hipotesis induksi bahwa klaim tersebut benar untuk semua $g' < g$. Jika ada klausa $(g \mid \alpha) \in pc(g)$ sedemikian sehingga $H(\alpha) = 0$, kita tambahkan g ke dalam H ; jika tidak, tambahkan $\bar{g}$ ke dalam H .

Misalkan g ditambahkan ke dalam H , maka setiap klausa dalam $pc(g)$ akan benar di bawah H . Untuk setiap klausa $(\bar{g} \mid \beta) \in nc(g)$, maka ada resolusi terurut antara $(g \mid \alpha)$ dan $(\bar{g} \mid \beta)$, dan resolvennya adalah $(\alpha \mid \beta)$. Berdasarkan Lemma 6.3.24, $(\alpha \mid \beta) \in GC$. Karena setiap literal g' dari $(\alpha \mid \beta)$ kurang dari g , berdasarkan hipotesis induksi, $(\alpha \mid \beta)$ benar di H . Karena $H(\alpha) = 0$, kita harus memiliki $H(\beta) = 1$. Itu berarti tidak ada klausa dalam $nc(g)$ yang akan salah di H .

Sekarang misalkan $\bar{g}$ ditambahkan ke H . Analisis di atas juga berlaku dengan mengubah peran g dan $\bar{g}$. Dalam kedua kasus, semua klausa dalam $pc(g)$ dan $nc(g)$ benar di H setelah menambahkan g atau $\bar{g}$ ke H . Dengan demikian, klaim tersebut benar untuk setiap $g \in GA$. Setelah setiap atom dalam GA diproses, kita telah menemukan model H dari GC berdasarkan klaim tersebut. Karena GC adalah himpunan semua instance dasar S^* , H adalah model Herbrand dari S^* atau S . Dengan kata lain, S dapat dipenuhi.

Perhatikan bahwa konstruksi model dalam bukti di atas bukanlah algoritmik, karena GC pada umumnya tak terbatas. Induksi tersebut sah karena didasarkan pada tatanan $>$ yang beralasan.

Sekarang mudah untuk membuktikan Teorema 6.2.7: Jika klausa kosong dihasilkan dari S , maka S harus tidak dapat dipenuhi; jika S tidak dapat dipenuhi, maka klausa kosong akan dihasilkan oleh resolusi terurut, yang merupakan kasus khusus dari resolusi.

Bukti tradisional Teorema 6.2.7 menggunakan teorema Herbrand:

Teorema 6.3.26 (Teorema Herbrand) Suatu himpunan S klausa tidak dapat dipenuhi jika dan hanya jika terdapat himpunan terbatas yang tidak dapat dipenuhi dari instansi dasar S .

Dari teorema Herbrand, bukti Teorema 6.2.7 adalah sebagai berikut:

1. Jika S tidak dapat dipenuhi, maka menurut teorema Herbrand, terdapat himpunan terbatas yang tidak dapat dipenuhi G dari instansi dasar S .
2. Resolusi untuk logika proposisional akan menemukan bukti dari G , dengan memperlakukan setiap atom dasar dalam G sebagai variabel proposisional.
3. Bukti resolusi dasar ini dapat diangkat menjadi bukti resolusi umum di S , menggunakan lemma yang mirip dengan Lemma 6.3.24.

Bukti kelengkapan berdasarkan urutan penyederhanaan dan tidak menggunakan teorema Herbrand.

6.4 PROVER9: PEMBUKTI TEOREMA RESOLUSI

Prover9 adalah pembukti teorema otomatis lengkap untuk logika orde pertama berdasarkan resolusi. Prover9 adalah penerus pembukti Otter; baik Otter maupun Prover9 diciptakan oleh William McCune (1953–2011).

Rumus Masukan ke Prover9

Prover9 memiliki mode otomatis penuh di mana pengguna cukup memberikan rumus yang mewakili masalah tersebut. Cara yang baik untuk mempelajari Prover9 adalah dengan menelusuri dan mempelajari contoh file masukan dan keluaran yang tersedia dengan distribusi Prover9. Mari kita lihat sebuah contoh.

Contoh 6.4.1 Setelah kita dapat menentukan teka-teki berikut dalam logika orde pertama, maka menemukan solusi dengan Prover9 adalah hal yang sepele:

Jack memiliki seekor anjing. Setiap pemilik anjing adalah pencinta hewan. Tidak ada pencinta hewan yang membunuh hewan. Jack atau Curiosity membunuh kucing yang bernama Tuna. Apakah Curiosity membunuh kucing itu?

Rumus orde pertama berikut berasal dari pernyataan teka-teki tersebut:

- (1) $\exists x(\text{dog}(x) \wedge \text{owns}(\text{Jack}, x))$
- (2) $\forall x(\exists y(\text{dog}(y) \wedge \text{owns}(x, y)) \rightarrow \text{animalLover}(x))$
- (3) $\forall x(\text{animalLover}(x) \rightarrow (\forall y(\text{animal}(y) \rightarrow \neg \text{kills}(x, y))))$
- (4) $\text{kills}(\text{Jack}, \text{Tuna}) \vee \text{kills}(\text{Curiosity}, \text{Tuna})$
- (5) $\text{cat}(\text{Tuna})$
- (6) $\forall x(\text{cat}(x) \rightarrow \text{animal}(x))$

Tujuannya adalah untuk menunjukkan bahwa $\text{kills}(\text{Curiosity}, \text{Tuna})$. Rumus-rumus di atas dapat ditulis dalam sintaks Prover9 sebagai berikut:

```
exists x (dog(x) & owns(Jack, x)).
all x (exists y (dog(y) & owns(x, y)) -> animalLover(x)).
all x (animalLover(x) -> (all y (animal(y) -> -kills(x, y)))).
```



```
kills(Jack, Tuna) | kills(Curiosity, Tuna).
cat(Tuna).
all x (cat(x) -> animal(x)).
```

Kita dapat menggunakan *kills(Curiosity, Tuna)* sebagai tujuan atau menambahkan *¬kills(Curiosity, Tuna)* (negasi dari tujuan) ke dalam himpunan klausa sehingga Prover9 akan menemukan bukti resolusi.

Dari contoh ini, kita dapat melihat bahwa setiap rumus dalam Prover9 diakhiri dengan “.”, dan kita menggunakan “->” untuk $\rightarrow$, “|” untuk $\vee$, “&” untuk $\wedge$, “-” untuk $\neg$, “all” untuk $\forall$, dan “exists” untuk $\exists$. Bahkan, “all” yang paling atas bersifat opsional karena variabel bebas diasumsikan dikuantifikasi secara universal. Prover9 akan mengubah rumus dalam daftar “asumsi” dan negasi rumus dalam daftar “tujuan” menjadi klausa sebelum resolusi dipanggil.

Perhatikan bahwa Prover9 menggunakan prioritas yang sama untuk operator Boolean dan kuantifikasi, “semua” dan “ada,” seperti yang ada dalam buku ini untuk menghilangkan beberapa tanda kurung. Misalnya,

```
exists x dog(x) & owns(Jack, x)
```

mewakili rumus $(\exists x \text{ dog}(x)) \wedge \text{owns}(\text{Jack}, x)$, bukan $\exists x (\text{dog}(x) \wedge \text{owns}(\text{Jack}, x))$. Untuk menghindari kebingungan, sisipkan sepasang tanda kurung ketika Anda tidak yakin tentang cakupan variabel terkuantifikasi.

Prover9 mengimplementasikan resolusi (Prosedur 6.2.9) dengan fitur-fitur tambahan. Daftar yang diberikan disebut daftar “sos” (set of support) dan daftar yang disimpan disebut daftar “usable”. Anda dapat menggunakan daftar “usable” dan daftar “sos” sebagai berikut:

```
formulas(usable). % the kept list
...
end_of_list.
formulas(sos). % the given list
...
end_of_list.
```

Resolusi selalu dilakukan antara klausa dari daftar sos (daftar yang diberikan) dan beberapa klausa dari daftar yang dapat digunakan (daftar yang disimpan). Setelah semua resolven antara klausa ini dan daftar yang dapat digunakan telah dihitung, klausa ini berpindah dari daftar sos ke daftar yang dapat digunakan. Dengan kata lain, resolusi antara klausa dalam daftar yang dapat digunakan asli dihilangkan. Perintah Prover9 dasar pada mesin Linux akan terlihat seperti:

```
prover9 -f Tuna.in > Tuna.out
```

atau

```
prover9 < Tuna.in > Tuna.out
```

di mana file bernama "Tuna.in" berisi input untuk Prover9 dan file bernama "Tuna.out" berisi output dari Prover9. Jika file "Tuna.in" berisi rumus-rumus dalam Contoh 6.4.1, maka file "Tuna.out" akan berisi bukti berikut:

```
===== PROOF =====
% Proof 1 at 0.03 (+ 0.05) seconds.
% Length of proof is 19.
% Level of proof is 6.
% Maximum clause weight is 6.
% Given clauses 0.

1 (exists x (dog(x) & owns(Jack,x))). [assumption].
2 (all x ((exists y (dog(y) & owns(x,y))) -> animalLover(x))).
   [assumption].
3 (all x (animalLover(x) -> (all y (animal(y) -> -kills(x,y)))))
   [assumption].
4 (all x (cat(x) -> animal(x))). [assumption].
5 kills(Curiosity,Tuna). [goal].
6 -dog(x) | -owns(y,x) | animalLover(y). [clausify(2)].
7 dog(c1). [clausify(1)].
8 -owns(x,c1) | animalLover(x). [resolve(6,a,7,a)].
9 owns(Jack,c1). [clausify(1)].
10 animalLover(Jack). [resolve(8,a,9,a)].
11 -animalLover(x) | -animal(y) | -kills(x,y). [clausify(3)].
12 -cat(x) | animal(x). [clausify(4)].
13 cat(Tuna). [assumption].
14 animal(Tuna). [resolve(12,a,13,a)].
15 -animal(x) | -kills(Jack,x). [resolve(10,a,11,a)].
16 kills(Jack,Tuna) | kills(Curiosity,Tuna). [assumption].
17 -kills(Curiosity,Tuna). [deny(5)].
18 -kills(Jack,Tuna). [resolve(14,a,15,a)].
19 $F. [back_unit_del(16),unit_del(a,18),unit_del(b,17)].
===== end of proof =====
```

Dalam Prover9, klausa kosong dilambangkan dengan \$F. Dalam setiap klausa, literal diberi nomor dengan $a, b, c, \dots$. Misalnya, untuk menghasilkan klausa 18, $\text{-kills(Jack, Tuna)}$, dari

```
14 animal(Tuna). [resolve(12,a,13,a)].
15 -animal(x) | -kills(Jack,x). [resolve(10,a,11,a)].
```

Prove9 melakukan resolusi pada literal pertama (a) dari klausa 14 dan literal pertama (a) dari klausa 15, dengan mgu $x \sim \text{Tuna}$, dan resolvennya adalah

18 -kills(Jack,Tuna). [resolve(14,a,15,a)].

Pesan yang diikuti oleh setiap klausa dapat diperiksa oleh manusia atau mesin, untuk memastikan kebenaran bukti.

Aturan dan Opsi Inferensi

Terdapat dua jenis parameter dalam Prover9: flag Boolean dan parameter numerik. Untuk mengubah nilai default dari jenis pertama, gunakan `set(flag)` atau `clear(flag)`. Yang terakhir dapat diubah dengan `assign(parameter, value)`. Misalnya, mode otomatis default diatur oleh

`set(auto).`

Mengaktifkan "auto" akan menyebabkan daftar flag lain diaktifkan, termasuk pengaturan "hyper-resolution" sebagai aturan inferensi utama untuk resolusi. Hyper-resolution mengurangi jumlah resolven perantara dengan menggabungkan beberapa langkah resolusi menjadi satu langkah inferensi.

Definisi 6.4.2 (*Hyper-Resolution*) Hyper-resolution positif terdiri dari urutan resolusi positif antara satu klausa non-positif (disebut nukleus) dan sekumpulan klausa positif (disebut satelit), hingga klausa positif atau klausa kosong dihasilkan.

Jumlah resolusi positif dalam hyper-resolution sama dengan jumlah literal negatif dalam klausa nukleus.

Contoh 6.4.3 Misalkan klausa inti adalah $(\overline{p(x)} \mid \overline{q(x)} \mid r(x))$, dan klausa satelit adalah $(q(a) \mid r(b))$ dan $(p(a) \mid r(c))$. Resolusi pertama antara inti dan satelit pertama menghasilkan $(\overline{p(a)} \mid r(a) \mid r(b))$. Resolusi kedua antara resolven dan satelit kedua menghasilkan resolven kedua $(r(a) \mid r(b) \mid r(c))$, yang merupakan hasil dari hiper-resolusi positif antara inti dan satelit.

Hiper-resolusi negatif dapat didefinisikan serupa dengan mengganti "positif" dengan "negatif." Hiper-resolusi berarti hiper-resolusi positif dan negatif.

Definisi 6.4.4 (Resolusi-ur) Resolusi unit-hasil (*ur*) terdiri dari urutan resolusi unit antara satu klausa non-unit (disebut nukleus) dan sekumpulan klausa unit (disebut satelit), hingga dihasilkan klausa unit atau klausa kosong.

Contoh 6.4.5 Pada Bagian 3.2, kita telah membahas sistem Hilbert yang terdiri dari tiga aksioma dan satu aturan inferensi, yaitu modus ponens. Untuk membuktikan $x \rightarrow x$ dalam sistem Hilbert, kita dapat menggunakan masukan berikut:

```
op(400, infix_right, ["->"]). % infix operator

formulas(usable).
P((n(y) -> n(x)) -> (x -> y)).
-P(x) | -P(x -> y) | P(y).
end_of_list.

formulas(sos).
P(x -> (y -> x)).
P((x -> (y -> z)) -> ((x -> y) -> (x -> z))).
-P(a -> a).
end_of_list.
```

Prover9 akan menghasilkan bukti berikut:

```
1 -P(x) | -P(x -> y) | P(y). [assumption].
2 P(x -> y -> x). [assumption].
3 P((x -> y -> z) -> (x -> y) -> x -> z). [assumption].
4 -P(a -> a). [assumption].
5 P(x -> y -> z -> y). [hyper(1,a,2,a,b,2,a)].
7 P((x -> y) -> x -> x). [hyper(1,a,2,a,b,3,a)].
15 P(x -> x). [hyper(1,a,5,a,b,7,a)].
16 $F. [resolve(15,a,4,a)].
```

Untuk contoh di atas, ur-resolution akan menghasilkan bukti yang sama dengan hyper-resolution. Hyper-resolution adalah aturan inferensi default untuk mode otomatis. Jika Anda ingin melihat bukti resolusi biner, Anda harus mematikan "auto" dan mengaktifkan "binary_resolution":

```
clear(auto).
set(binary_resolution).
```

Dengan mematikan hyper-resolution, Prover9 akan menghasilkan bukti berikut:

```
1 -P(x) | -P(x -> y) | P(y). [assumption].
2 P(x -> y -> x). [assumption].
3 P((x -> y -> z) -> (x -> y) -> x -> z). [assumption].
4 -P(a -> a). [assumption].
5 -P(x) | P(y -> x). [resolve(2,a,1,b)].
6 -P(x -> y -> z) | P((x -> y) -> x -> z). [resolve(3,a,1,b)].
10 P((x -> y) -> x -> x). [resolve(6,a,2,a)].
19 P(x -> (y -> z) -> y -> y). [resolve(10,a,5,a)].
```

```

20 -P(x -> y) | P(x -> x). [resolve(10,a,1,b)].
45 P(x -> x). [resolve(20,a,19,a)].
46 $F. [resolve(45,a,4,a)].

```

Karena resolusi positif lengkap dan resolusi unit tidak lengkap, maka resolusi hiper lengkap dan resolusi hasil unit tidak lengkap. Untuk menggunakan “resolusi hasil unit,” gunakan perintah

```
set(ur_resolution).
```

Untuk menggunakan “resolusi hiper” di Prover9, gunakan perintah

```
set(hyper_resolution).
```

Opsi ini akan menyebabkan flag `pos_hyper_resolution` dan `neg_hyper_resolution` menjadi true.

Aturan inferensi resolusi yang disediakan oleh Prover9 meliputi “resolusi biner,” “resolusi hiper,” “resolusi terurut,” dan “resolusi hasil unit.” Untuk menggunakan “resolusi terurut,” gunakan perintah

```
set(ordered_res).
```

Opsi ini memberikan batasan pada aturan inferensi resolusi biner dan hiper. Ini menyatakan bahwa literal yang diselesaikan dalam satu atau lebih induk harus maksimal dalam klausa. Melanjutkan dari contoh sebelumnya, jika kita mengaktifkan flag resolusi terurut, bukti yang dihasilkan oleh Prover9 akan menjadi sebagai berikut.

```

1 -P(x) | -P(x -> y) | P(y). [assumption].
2 P(x -> y -> x). [assumption].
3 P((x -> y -> z) -> (x -> y) -> x -> z). [assumption].
4 -P(a -> a). [assumption].
5 -P(x) | P(y -> x). [resolve(2,a,1,b)].
6 -P(x -> y -> z) | P((x -> y) -> x -> z). [resolve(3,a,1,b)].
8 P(x -> y -> z -> y). [resolve(5,a,2,a)].
9 P(x -> y -> z -> u -> z). [resolve(8,a,5,a)].
10 P(x -> y -> z -> u -> w -> u). [resolve(9,a,5,a)].
14 P((x -> y) -> x -> x). [resolve(6,a,2,a)].
19 -P(x -> y) | P(x -> x). [resolve(14,a,1,b)].
21 P(x -> x). [resolve(19,a,10,a)].
22 $F. [resolve(21,a,4,a)].

```

Urutan Penyederhanaan dalam Prover9

Prover9 memiliki beberapa metode yang tersedia untuk membandingkan istilah atau literal. Urutan istilah adalah urutan parsial (dan terkadang total pada istilah dasar), dan

digunakan untuk memutuskan literal mana dalam klausa yang dapat diterima untuk penerapan resolusi berurutan. Beberapa aturan resolusi mengharuskan beberapa literal menjadi maksimal dalam klausanya.

Prioritas simbol adalah urutan total pada simbol fungsi dan predikat (termasuk konstanta). Fungsi pembobotan simbol memetakan simbol ke bilangan bulat non-negatif. Prover9 mendukung tiga urutan penyederhanaan: LPO (urutan jalur leksikografis), RPO (urutan jalur rekursif), dan KBO (urutan Knuth–Bendix), yang diperkenalkan di Bagian 6.3.2.

Berikut adalah perintah untuk memilih urutan penyederhanaan.

```
assign(order, string). % default string=lpo, range [lpo,rpo,kbo]
```

Opsi ini digunakan untuk memilih urutan utama yang akan digunakan untuk menentukan literal maksimal dalam klausa. Pilihannya adalah “lpo” (urutan jalur leksikografis), “rpo” (urutan jalur rekursif), dan “kbo” (urutan Knuth–Bendix).

Prioritas simbol default (untuk LPO, RPO, dan KBO) diberikan oleh aturan berikut (secara berurutan).

- Simbol fungsi < simbol predikat non-kesetaraan.
- Untuk simbol fungsi: $c/0 < f/2 < g/1 < h/3 < i/4 < \dots$, di mana c adalah konstanta apa pun, f adalah fungsi apa pun dengan aritas 2, g adalah fungsi apa pun dengan aritas 1, ... (perhatikan posisi $g/1$).
- Untuk simbol predikat: aritas lebih rendah < aritas lebih tinggi.
- Simbol non-Skolem < simbol Skolem.
- Untuk simbol Skolem, indeks yang lebih rendah adalah yang lebih kecil.
- Untuk simbol non-Skolem, lebih banyak kemunculan < lebih sedikit kemunculan.
- Urutan ASCII leksikal (fungsi strcmp() UNIX).

Perintah `function_order` dan `predicate_order` dapat digunakan untuk mengubah prioritas simbol default. Perintah ini berisi daftar simbol yang diurutkan berdasarkan prioritas dari yang terkecil hingga yang terbesar. Misalnya,

```
predicate_order([=, <=, P, Q]).           % = < <= < P < Q
function_order([a, b, c, +, *, h, g]).     % a < b < c < + < * < h < g
```

Kita membutuhkan dua perintah terpisah untuk mendefinisikan prioritas, karena simbol predikat diasumsikan selalu lebih besar daripada simbol fungsi dalam prioritas. Prioritas simbol yang digunakan untuk suatu masalah selalu dicetak dalam file output (di bagian INPUT PROSES).

Pustaka TPTP

Prover9 telah membuktikan secara otomatis banyak teorema dalam pustaka TPTP, di mana TPTP merupakan singkatan dari Thousands of Problems for Theorem Provers (Ribuan Masalah untuk Pembukti Teorema). TPTP dikelola oleh Geoff Sutcliffe di Universitas Miami. Menurut situs web TPTP, tptp.org, TPTP berisi ribuan soal uji di lebih dari 50 domain dan mendukung format input untuk lebih dari 50 sistem pembuktian teorema otomatis (ATP). TPTP

telah digunakan untuk mendukung banyak kompetisi ATP. TPTP menyediakan fungsi-fungsi berikut kepada komunitas ATP:

- Pustaka komprehensif soal uji yang tersedia saat ini, untuk memberikan gambaran umum dan mekanisme referensi yang sederhana dan tidak ambigu.
- Daftar referensi komprehensif dan informasi menarik lainnya untuk setiap soal.
- Contoh soal generik dengan ukuran sembarang (misalnya, soal N-ratu).
- Sebuah alat bantu untuk mengkonversi soal-soal ke format sistem ATP yang sudah ada.
- Pedoman umum yang menguraikan persyaratan untuk evaluasi sistem ATP.
- Standar untuk input dan output sistem ATP.

Bagi mereka yang tertarik pada pembuktian teorema otomatis, TPTP adalah tempat yang kaya untuk dieksplorasi.

6.5 LATIHAN SOAL

1. Misalkan $\theta = [x \sim u, y \sim f(g(u)), z \sim f(u)]$ dan $\sigma = [u \sim x, x \sim f(u), y \sim a, z \sim f(u)]$. Hitung $\theta\sigma$ dan $\sigma\theta$. Di antara $\theta, \sigma, \theta\sigma$, dan $\sigma\theta$, substitusi mana yang idempoten?
2. Gunakan algoritma unifikasi berbasis aturan *unify* untuk memutuskan apakah pasangan istilah berikut dapat diunifikasi atau tidak dan berikan setiap keadaan dalam *unify* setelah menerapkan aturan transformasi.
 - (a) $p(x, a) \doteq p(f(y), y)$
 - (b) $p(x, f(x)) \doteq p(f(y), y)$
 - (c) $p(f(a), g(x)) \doteq p(y, y)$
 - (d) $q(x, y, f(y)) \doteq q(u, h(v, v), u)$
 - (e) $q(a, x, f(g(y))) \doteq q(z, f(z), f(u))$
3. Bagaimana cara memodifikasi *unify*(s, t) sehingga dapat digunakan untuk memutuskan apakah suatu himpunan S istilah dapat diunifikasi? Artinya, temukan substitusi σ yang idempoten dan paling umum, serta istilah t sedemikian sehingga $s\sigma = t$ untuk setiap $s \in S$.
4. Temukan mgu dari $s = f(f(\dots f(f(a, x_1), x_2), \dots), x_n)$ dan $t = f(x_n, f(x_{n-1}, f(\dots, f(x_1, a) \dots)))$ untuk $n = 5$.
5. Buat grafik istilah $G = (V, E)$ untuk setiap pasangan istilah berikut:
 - (a) $s = p(x, a)$ dan $t = p(f(y), y)$
 - (b) $s = p(x, f(x))$ dan $t = p(f(y), y)$
 - (c) $s = q(x, y, f(y))$ dan $t = q(u, h(v, v), u)$
 - (d) $s = q(a, x, f(g(y)))$ dan $t = q(z, f(z), f(u))$
 - (e) $s = f(f(f(f(a, x_1), x_2), x_3), x_4)$ dan $t = f(x_4, f(x_3, f(x_2, f(x_1, a))))$
 dan tunjukkan kelas-kelas kesetaraan V setelah memanggil *unify*₁ dengan sukses.
6. Berikan pseudo-code untuk prosedur *postOccurCheck* yang digunakan dalam *lunify*. Berapa kompleksitas algoritma Anda?
7. Rancang algoritma yang efisien dan berikan pseudo-code-nya untuk membuat mgu ketika prosedur *lunify* mengembalikan nilai true. Berapa kompleksitas algoritma Anda?

8. Temukan contoh tandingan untuk menunjukkan bahwa KBO tidak berdasar dengan baik ketika kondisi berikut salah: “terdapat paling banyak satu simbol fungsi uner $f \in \Sigma$ sedemikian sehingga $w(f) = 0$ dan f harus maksimal dalam prioritas jika $w(f) = 0$.”
9. Tunjukkan bahwa semua sisi kiri s dari setiap relasi ekuivalensi dalam Contoh 6.3.22 lebih besar dari sisi kanannya t dengan $>_{lpo}$, dan jelaskan prioritas apa yang digunakan dan kondisi $>_{lpo}$ apa yang digunakan untuk $s >_{lpo} t$.
10. Berikan prioritas untuk $\{\times, +\}$ dan tunjukkan bahwa semua sisi kiri s dari setiap persamaan berikut lebih besar dari sisi kanannya t dengan $>_{lpo}$. Jelaskan kondisi $>_{lpo}$ apa yang digunakan untuk $s >_{lpo} t$.

$$\begin{aligned}(x \times y) \times z &= x \times (y \times z) \\ x \times (y + z) &= (x \times y) + (x \times z) \\ (y + z) \times x &= (y \times x) + (z \times x)\end{aligned}$$

Bisakah Anda menggunakan $>_{rpo}$ atau $>_{kbo}$ untuk mencapai hasil yang sama? Mengapa?

11. Pilih dengan justifikasi urutan penyederhanaan dari $>_{lpo}$, $>_{rpo}$, dan $>_{kbo}$ sedemikian sehingga sisi kiri s lebih besar dari sisi kanan t untuk setiap persamaan berikut (yang mendefinisikan fungsi Ackermann):

$$\begin{aligned}A(0, y) &= s(y) \\ A(s(x), 0) &= A(x, s(0)) \\ A(s(x), s(0)) &= A(x, A(s(x), y))\end{aligned}$$

12. Masalah Klub Hoofers mengasumsikan pernyataan berikut:

Tony, Tom, dan Liz adalah anggota Klub Hoofers. Setiap anggota Klub Hoofers adalah pemain ski atau pendaki gunung atau keduanya. Tidak ada pendaki gunung yang menyukai hujan, dan semua pemain ski menyukai salju. Liz tidak menyukai apa pun yang disukai Tony dan menyukai apa pun yang tidak disukai Tony. Tony menyukai hujan dan salju.

Dan pertanyaan yang diajukan adalah: “Apakah ada anggota Klub Hoofers yang merupakan pendaki gunung tetapi bukan pemain ski?” Silakan temukan bukti resolusi berikut untuk masalah Klub Hoofers: (a) resolusi unit, (b) resolusi input, (c) resolusi positif, (d) resolusi negatif, dan (e) resolusi linier.

13. Gunakan Prover9 untuk menjawab pertanyaan dalam masalah Klub Hoofers pada soal sebelumnya. Harap siapkan input untuk Prover9 dan serahkan file output dari Prover9.
14. Gunakan resolusi biner, resolusi hiper, dan resolusi ur, masing-masing, dari Prover9 untuk menjawab pertanyaan “Apakah John bahagia?” dari pernyataan-pernyataan berikut:

Siapa pun yang lulus ujian logika dan memenangkan lotre bahagia. Tetapi siapa pun yang belajar atau beruntung dapat lulus semua ujiannya. John tidak belajar tetapi dia beruntung. Siapa pun yang beruntung memenangkan lotre.

15. Gunakan resolusi hiper dan resolusi ur, masing-masing, dari Prover9 untuk menunjukkan bahwa “alat penyiram air sedang menyala” dari pernyataan berikut:

Seseorang yang tinggal di rumah Anda basah. Jika seseorang basah, itu karena hujan, alat penyiram air, atau keduanya. Jika seseorang basah karena alat penyiram air, alat penyiram air pasti menyala. Jika seseorang basah karena hujan, orang itu pasti tidak membawa payung. Ada payung di rumah Anda, yang tidak ada di lemari. Payung yang tidak ada di lemari pasti dibawa oleh seseorang yang tinggal di rumah itu. Tidak ada seorang pun yang membawa payung.

Harap siapkan input untuk Prover9 dan serahkan file output dari Prover9.

16. Mengikuti pendekatan yang diilustrasikan dalam Contoh 6.4.5, gunakan Prover9 untuk membuktikan bahwa properti berikut ini benar dalam sistem Hilbert:

$$(a) (x \rightarrow y) \rightarrow ((y \rightarrow z) \rightarrow (x \rightarrow z))$$

$$(b) (x \rightarrow (y \rightarrow z)) \rightarrow (y \rightarrow (x \rightarrow z))$$

$$(c) \neg x \rightarrow (x \rightarrow y)$$

$$(d) x \rightarrow (\neg x \rightarrow y)$$

$$(e) \neg \neg x \rightarrow x$$

$$(f) x \rightarrow \neg \neg x$$

$$(g) (x \rightarrow \neg x) \rightarrow \neg x$$

$$(h) (\neg x \rightarrow x) \rightarrow x$$

$$(i) (x \rightarrow y) \rightarrow (\neg y \rightarrow \neg x)$$

BAB 7

LOGIKA ORDE PERTAMA DENGAN KESETARAAN

Terdapat banyak relasi kesetaraan dalam logika. Pada tingkat semantik, kita memiliki kesetaraan logis dan equisatisfiability, yang dilambangkan dengan simbol $\equiv$ dan $\approx$, masing-masing, baik untuk logika proposisional maupun logika orde pertama. Pada tingkat sintaksis, kita memiliki operator logika $\leftrightarrow$ untuk menyatakan kesetaraan antara dua pernyataan. Bahasa orde pertama memiliki dua jenis objek, yaitu rumus dan istilah, dan kita menggunakan $\leftrightarrow$ untuk rumus. Simbol konvensional untuk kesetaraan istilah adalah "=", yang merupakan simbol predikat. Pada bab sebelumnya, kita telah mempelajari logika orde pertama tanpa kesetaraan dan telah menggunakan "=" sebagai relasi identitas di luar logika orde pertama. Pada bab ini, kita akan mempelajari logika orde pertama dengan kesetaraan, yaitu, kita akan mempelajari "=" sebagai simbol predikat dalam logika orde pertama.

7.1 KESAMAAN ISTILAH

Kesamaan adalah relasi yang ada di mana-mana dan penting di setiap bidang matematika. Misalnya, bagaimana Anda akan menentukan bahwa "untuk setiap x , terdapat y yang unik sedemikian sehingga relasi $p(x, y)$ berlaku"? Jika arti dari $p(x, y)$ adalah " x adalah ibu dari y ," maka bagaimana Anda akan menyatakan dalam logika orde pertama bahwa "ibu setiap orang adalah unik"? Jika arti dari $p(x, y)$ adalah $f(x) = y$, maka bagaimana Anda akan menyatakan bahwa "untuk setiap x , $f(x)$ memiliki nilai yang unik"? Dengan menggunakan "=", jawabannya mudah:

$$\forall x \exists y (p(x, y) \wedge \forall z (p(x, z) \rightarrow y = z))$$

Rumus di atas sering dilambangkan dengan $\forall x \exists! y p(x, y)$ dalam matematika. Penggunaan "=" memberi kita daya ekspresi yang lebih tinggi untuk menentukan berbagai pernyataan:

- Kita dapat menyatakan "terdapat setidaknya dua elemen untuk x sedemikian sehingga $A(x)$ berlaku" sebagai

$$\exists x \exists y x = y \wedge A(x) \wedge A(y)$$

- Kita dapat menyatakan "terdapat paling banyak dua elemen untuk x sedemikian sehingga $A(x)$ berlaku" sebagai

$$\forall x \forall y \forall z ((A(x) \wedge A(y) \wedge A(z)) \rightarrow (x = y \vee y = z \vee x = z))$$

Ini menyatakan bahwa jika kita memiliki tiga elemen yang memenuhi $A(x)$, maka dua di antaranya harus sama.

- Kita dapat menyatakan “terdapat tepat dua elemen untuk x sedemikian sehingga $A(x)$ berlaku” sebagai gabungan dari dua pernyataan di atas.

Aksioma Kesamaan

Aksioma kesamaan terdiri dari lima jenis rumus; masing-masing cukup terkenal sehingga memiliki nama tersendiri. Berikut ini, variabel yang muncul dalam rumus diasumsikan terkuantifikasi secara universal:

- **Refleksivitas:**
 $x = x$. Suatu objek selalu sama dengan dirinya sendiri.
- **Komutativitas:**
 $(x = y) \rightarrow (y = x)$. Tidak masalah bagaimana parameter dari $=$ diurutkan. Artinya, $=$ bersifat komutatif.
- **Transitivitas:**
 $(x = y) \wedge (y = z) \rightarrow (x = z)$. Kesamaan dapat disebar. Artinya, $=$ bersifat transitif.
- **Monotonisitas fungsi:**
Untuk $1 \leq i \leq k, (x_i = y_i) \rightarrow (f(x_1, \dots, x_k) = f(y_1, \dots, y_k))$. Jika argumen suatu fungsi sama secara berpasangan, maka suku-suku yang tersusun akan sama. Artinya, $=$ adalah relasi monotonik atas suku-suku.
- **Monotonisitas predikat:**
Untuk $1 \leq i \leq k, (x_i = y_i) \wedge p(x_1, \dots, x_k) \rightarrow p(y_1, \dots, y_k)$. Jika argumen suatu predikat sama secara berpasangan, maka atom-atom yang dihasilkan secara logis setara.

Tiga aksioma pertama bersama-sama menyatakan bahwa “ $=$ ” adalah relasi kesetaraan. Aksioma keempat dan kelima sangat mirip dan memiliki nama yang sama, yaitu monotonisitas. Aturan monotonisitas pertama mengatur suku-suku dan aturan monotonisitas kedua mengatur rumus. Keduanya menegaskan bahwa memiliki argumen yang sama memastikan hasil yang sama. Faktanya, ini adalah skema aksioma karena kita membutuhkan versi untuk setiap fungsi f dan setiap predikat p . Dengan kata lain, kita akan membutuhkan versi yang berbeda dari aksioma keempat dan kelima dalam bahasa orde pertama yang berbeda. Terkadang, kita dapat membagi aksioma monotonisitas menjadi serangkaian aksioma untuk setiap argumen. Misalnya, jika aritas predikat p adalah 2, kita memperoleh serangkaian dua aksioma dari predikat monotonisitas:

$$\begin{aligned} (x_1 = y_1) \wedge p(x_1, x_2) &\rightarrow p(y_1, x_2) \\ (x_2 = y_2) \wedge p(x_1, x_2) &\rightarrow p(x_1, y_2) \end{aligned}$$

Aksioma monotonisitas ini tidak boleh disamakan dengan aturan substitusi, yang hanya memperbolehkan monotonisitas suku untuk variabel saja.

Kelima jenis aksioma tersebut dapat dengan mudah diubah menjadi klausa sehingga kita dapat menggunakan resolusi untuk membuktikan teorema yang melibatkan $=$.

Contoh 7.1.1 Misalkan $f(a) = b$ dan $f(b) = a$, jika kita tidak diperbolehkan menggunakan "substitusi sama dengan sama," bagaimana Anda dapat membuktikan bahwa $f(f(a)) = a$? Jawabannya adalah resolusi. Karena resolusi adalah pembuktian sanggahan, kita perlu menambahkan negasi dari $f(f(a)) = a$, yaitu $f(f(a)) \neq a$, ke dalam himpunan klausa dari aksioma kesamaan dan premis dan menunjukkan bahwa himpunan klausa tersebut tidak dapat dipenuhi:

1. $(f(a) = b)$	<i>premis</i>
2. $(f(b) = a)$	<i>premis</i>
3. $(f(f(a)) \neq a)$	<i>negasi dari tujuan</i>
4. $(x \neq y \mid y \neq z \mid x = z)$	<i>transitivitas</i>
5. $(x \neq y \mid f(x) = f(y))$	<i>monotonisitas</i>
6. $(f(f(a)) = f(b))$	<i>resolven dari 1 dan 5</i>
7. $(f(b) \neq z \mid f(f(a)) = z)$	<i>resolven dari 2 dan 7</i>
8. $(f(f(a)) = a)$	<i>resolven dari 6 dan 4</i>
9. $()$	<i>resolven dari 3 dan 8</i>

Ini menunjukkan Jelas bahwa tanpa aksioma kesamaan, kita tidak dapat memiliki bukti resolusi $f(f(a)) = a$, dan bukti resolusi menjadi rumit jika kita tidak dapat menggunakan "substitusi yang sama dengan yang sama."

Semantik dari "="

Simbol kesamaan dimaksudkan untuk memodelkan kesamaan objek dalam suatu domain. Misalnya, ketika kita mengatakan " $2 + 0 = 2$ " atau $add(s(s(0)), 0) = s(s(0))$, kita bermaksud bahwa " $2 + 0$ " dan " 2 " mewakili objek yang sama dalam suatu domain. Artinya, $s = t$ benar jika s dan t mewakili objek yang "sama" atau "identik" dalam domain apa pun. Kita menegaskan bahwa dua deskripsi yang berbeda merujuk pada objek yang sama. Karena gagasan identitas dapat diterapkan pada hampir semua domain objek, kesamaan sering diasumsikan ada di mana-mana dalam setiap logika. Namun, pembicaraan tentang "kesamaan" atau "identitas" menimbulkan pertanyaan filosofis yang rumit. "Apakah saya orang yang sama seperti tiga hari yang lalu?"

Kita membutuhkan cara yang sederhana dan jelas untuk mendefinisikan arti dari $=$. Untuk bahasa orde pertama $L = (P, F, X, Op)$, model Herbrand untuk CNF A dapat direpresentasikan secara sederhana oleh himpunan H atom dasar sedemikian sehingga hanya atom-atom dalam H yang diinterpretasikan sebagai benar. Karena adanya aksioma kesamaan, himpunan H harus memiliki beberapa sifat.

Definisi 7.1.2 (Kongruensi) Suatu relasi $=_m$ pada himpunan istilah $T(F, X)$ disebut kongruensi jika $=_m$ adalah relasi ekuivalensi yang tertutup terhadap instansiasi:

$$s\sigma =_m t\sigma \text{ if } s =_m t$$

untuk setiap substitusi $\sigma: X \rightarrow T(F, X)$ dan tertutup di bawah monotonisitas

$$f(s_1, \dots, s_k) =_m f(t_1, \dots, t_k) \text{ if for any } 1 \leq i \leq k, s_i =_m t_i$$

untuk setiap fungsi f/k .

Ingat bahwa relasi biner R stabil jika $R(s, t)$ menyiratkan $R(s\sigma, t\sigma)$ untuk setiap substitusi σ (Definisi 6.3.7). Menurut definisi, R stabil jika dan hanya jika R tertutup terhadap instansiasi.

Menurut definisi, $=$ adalah kongruensi karena merupakan relasi ekuivalensi yang tertutup terhadap instansiasi dan monotonisitas. Diberikan himpunan A klausa yang berisi semua aksioma kesamaan, jika A dapat dipenuhi, maka setiap model A menginduksi kongruensi seperti yang ditunjukkan oleh proposisi berikut.

Proposisi 7.1.3 Misalkan H adalah model Herbrand dari A yang berada dalam CNF dan berisi semua aksioma kesamaan. Definisikan $s =_H t$ jika dan hanya jika $(s = t) \in H$ untuk setiap $s, t \in T(F)$, himpunan istilah dasar yang dibangun dari F . Maka, $=_H$ adalah kongruensi atas $T(F)$.

Bukti Tiga aksioma kesamaan pertama memastikan bahwa $=_H$ adalah relasi ekuivalensi, dan aksioma monotonisitas memastikan bahwa $=_H$ bersifat monoton. $=_H$ tertutup terhadap instansiasi karena H tidak mengandung variabel.

Karena relasi kongruensi $=_m$ adalah relasi ekuivalensi, kita dapat mempartisi $T(F)$ dengan $=_m$ menjadi kelas-kelas ekuivalensi, yang juga disebut kelas-kelas kongruensi, sedemikian sehingga setiap kelas mengandung objek-objek yang setara. Untuk setiap $s \in T(F)$, misalkan

$$[s] = \{t \mid s =_m t, t \in T(F)\},$$

misalkan menyatakan kelas kekongruan dari s .

$$T(F)/=_m = \{[s] \mid s \in T(F)\},$$

himpunan semua kelas kekongruenan, yang juga disebut basis Herbrand modulo $=_m$.

Contoh 7.1.4 Misalkan $F = \{0/0, s/1\}$, dan A berisi $s(s(s(x))) = x$ dan aksioma kesamaan. Misalkan H adalah model Herbrand dari A di mana $(0 = s(0)) \notin H$, maka

$$\begin{aligned} T(F) &= \{0, s(0), s(s(0)), s(s(s(0))), s^4(0), \dots\} \\ [0] &= \{0, s(s(s(0))), s^6(0), s^9(0), \dots\}, \\ [s(0)] &= \{s(0), s^4(0), s^7(0), s^{10}(0), \dots\}, \\ [s(s(0))] &= \{s(s(0)), s^5(0), s^8(0), s^{11}(0), \dots\}, \\ T(F)/=_H &= \{[0], [s(0)], [s(s(0))]\} \end{aligned}$$

Dalam contoh ini, basis Herbrand modulo $=_H$, $T(F)/=_H$, kebetulan terbatas.

Sekarang, mudah untuk memeriksa apakah dua istilah dasar s dan t sama atau tidak di bawah H : $(s = t) \in H$ jika dan hanya jika $t \in [s]$ (atau $s \in [t]$), yaitu, $s =_H t$. Bahkan, ketika kita mendefinisikan model Herbrand untuk suatu rumus dengan kesamaan, pertama-tama kita dapat mendefinisikan kongruensi $=_m$ dan kemudian menganggap $T(F)/=_m$ sebagai domain dari model Herbrand dengan kesamaan: $(s = t) \in H$ jika dan hanya jika $s =_m t$. Dengan cara ini, kita tidak perlu memperhatikan aksioma kesamaan secara eksplisit.

Contoh 7.1.5 Misalkan $F = \{0/0, s/1, -/1\}$, dan A berisi $-(0) = 0, -(-x) = x, s(-(s(x))) = -x$, dan aksioma kesamaan:

$$\begin{aligned} T(F) &= \{0, s(0), -(0), s(s(0)), s(-(0)), -(-(0)), s(s(s(0))), \dots\} \\ [0] &= \{0, -(0), -(-(0)), -(-(-(0))), s(-(s(0))), -^4(0), -^5(0), \dots\} \\ [s(0)] &= \{s(0), s(-(0)), s(-(-(0))), -(-(s(0))), s(s(-(s(0)))), \dots\}, \\ [-s(s(0))] &= \{-s(s(0)), -s(-(s(0))), s(-(s(s(0)))) - (-(-(s(0)))), -s(-(-(0))), \dots\}, \\ [s(s(0))] &= \{s(s(0)), s(s(-(0))), s(s(-(-(0)))), s(s(s(-(s(0))))), \dots\} \\ T(F)/=_A &= \{[0], [s(0)], [-s(0)], [s(s(0))], [-s(s(0))], \dots\} \end{aligned}$$

Dalam contoh ini, setiap kelas kekongruenan $[t]$ adalah tak terhingga, dan kumpulan semua kelas kekongruenan, yaitu $T(F)/=_A$, juga tak terhingga. Model alami dari A adalah himpunan $\mathbb{Z}$ bilangan bulat, di mana bilangan bulat non-negatif n diwakili oleh $s^n(0)$ dan bilangan bulat negatif $-n$ oleh $-(s^n(0))$ untuk $n > 0$. Arti s adalah "menambahkan satu" dan $-$ adalah tanda minus. Kita dapat menggunakan $T(F)/=_A$ sebagai basis Herbrand untuk mendefinisikan model Herbrand. Namun, akan merepotkan untuk mendefinisikan operasi biasa seperti $+$, $-$, $*$, dll., dalam pengaturan ini. Solusi alternatif adalah menggunakan $\{0, s, p\}$ dengan $\{s(p(x)) = x, p(s(x)) = x\}$ sebagai blok penyusun untuk himpunan bilangan bulat: bilangan bulat negatif $-n$ diwakili oleh $p^n(0)$ dan bilangan bulat positif n oleh $s^n(0)$.

Teori Persamaan

Suatu himpunan persamaan hanyalah himpunan klausa unit positif di mana satu-satunya simbol predikat adalah " $=$ ".

Contoh 7.1.6 Dalam aljabar modern, sebuah grup adalah $G = (S, *)$, di mana S adalah himpunan elemen dan $*$ adalah operator biner yang memenuhi sifat-sifat bahwa (a) $*$ bersifat asosiatif dan tertutup pada S ; (b) terdapat elemen identitas di S ; dan (c) setiap elemen S memiliki invers di S . Dengan menggunakan pendekatan logika orde pertama, sifat penutupan bersifat implisit; identitas dilambangkan dengan e ; untuk setiap $x \in S$, invers dari x dilambangkan dengan $i(x)$. Kita hanya membutuhkan tiga persamaan berikut sebagai aksioma:

(1)	$e * x = x,$	e adalah elemen identitas kiri
(2)	$i(x) * x = e,$	$i(x)$ adalah invers kiri dari x
(3)	$(x * y) * z = x * (y * z)$	$*$ bersifat asosiatif

Dari ketiga persamaan ini, kita dapat membuktikan banyak sifat menarik dari teori grup, seperti $i(i(x)) = x$, $x * e = x$, $x * i(x) = e$, dll.

Definisi 7.1.7 Diberikan himpunan persamaan E , relasi $=_E$ didefinisikan secara rekursif sebagai berikut:

1. $s =_E t$ if $s = t \in E$
2. $t =_E t$ if $t \in T(F, X)$
3. $s =_E t$ if $t =_E s$
4. $s =_E t$ if $s =_E r, r =_E t$
5. $f(\dots, s_i, \dots) =_E f(\dots, t_i, \dots)$ if $s_i =_E t_i$ and $f \in F$
6. $s\sigma =_E t\sigma$ if $s =_E t$ and $\sigma: X \rightarrow T(F, X)$ is any substitution

Masalah kekongruenan logika orde pertama dengan kesamaan adalah memutuskan apakah $s =_E t$ atau tidak untuk setiap istilah s dan t .

Ingat bahwa setiap relasi biner R , termasuk $=_E$, dapat dianggap sebagai himpunan pasangan item, yaitu $\{(s, t) \mid R(s, t)\}$. Demikian pula, baik E maupun $=_E$ juga dapat dianggap sebagai himpunan pasangan istilah dan $E \subseteq =_E$.

Definisi 7.1.8 Diberikan relasi biner R pada $T(F, X)$, R dikatakan tertutup terhadap hal-hal berikut:

- *Refleksivitas* jika $(t, t) \in R$ untuk setiap $t \in T(F, X)$
- *Komutativitas* jika $(t, s) \in R$ setiap kali $(s, t) \in R$
- *Transitivitas* jika $(r, t) \in R$ setiap kali $(r, s), (s, t) \in R$
- *Monotonisitas* jika $(f(s_1, \dots, s_k), f(t_1, \dots, t_k)) \in R$ setiap kali $(s_i, t_i) \in R$ untuk $1 \leq i \leq k$
- *Instansiasi* jika $(s\sigma, t\sigma) \in R$ setiap kali $(s, t) \in R$ untuk setiap $\sigma: X \rightarrow T(F, X)$

Penutupan transitif adalah himpunan bagian terkecil dari R yang tertutup terhadap transitivitas; Penutupan ekuivalensi adalah himpunan bagian minimum dari R yang tertutup terhadap refleksivitas, komutativitas, dan transitivitas; penutupan kongruensi adalah penutupan ekuivalensi minimum dari R yang juga tertutup terhadap monotonisitas dan instansiasi.

Perhatikan bahwa monotonisitas dan instansiasi adalah dua konsep yang berbeda. Jika $(f(a, x), f(b, x)) \in R$, monotonisitas menambahkan $(g(f(a, x)), g(f(b, x)))$ ke dalam R dan instansiasi menambahkan $(f(a, g(x)), f(b, g(x)))$ ke dalam R .

Proposisi 7.1.9 Untuk setiap himpunan persamaan E , $=_E$ adalah kongruensi minimum yang mengandung E .

Bukti Mari kita periksa kondisi agar $s =_E$ benar: kondisi 1 mengatakan $E \subset =_E$; kondisi 2–4 mengatakan $=_E$ adalah relasi ekuivalensi. Kondisi 5 dan 6 mengatakan $=_E$ tertutup terhadap monotonisitas dan instansiasi.

Contoh 7.1.10 Menggunakan F dari Contoh 7.1.4, misalkan $E = \{s(s(s(x))) = x\}$ dan $X = \{x\}$. Karena $=_E$ adalah suatu kekongruenan, kita dapat memeriksa bahwa

$$T(F)/=_E = T(F)/=_H = \{[0], [s(0)], [s(s(0))]\},$$

di mana $[0] = \{0, s(s(s(0))), s^6(x), \dots\}$, $[s(0)] = \{s(0), s^4(x), s^7(x), \dots\}$, dan $[s(s(0))] = \{s(s(0)), s^5(0), s^8(0), \dots\}$.

$$T(F, X)/=_E = \{[0], [s(0)], [s(s(0))], [x], [s(x)], [s(s(x))]\},$$

di mana $[x] = \{x, s(s(s(x))), s^6(x), \dots\}$, $[s(x)] = \{s(x), s^4(x), s^7(x), \dots\}$, dan $[s(s(x))] = \{s(s(x)), s^5(x), s^8(x), \dots\}$.

Proposisi 7.1.11 Diberikan E , misalkan E_{ax} adalah aksioma kesamaan yang terkait dengan E , maka untuk setiap $(s, t) \in =_E, E \cup E_{ax} \models (s = t)$.

Bukti (Sketsa) Periksa keenam kondisi Definisi 7.1.7 untuk $s =_E t$: kondisi 1 berlaku karena $A \models A$. Kondisi 2–5 berlaku karena E_{ax} . Kondisi 6 berlaku karena variabel bebas diasumsikan dikuantifikasi secara universal. Karena $=_E$ mengandung semua persamaan yang merupakan konsekuensi logis dari E dan aksioma kesamaan, kita menyebut $=_E$ sebagai teori E .

Contoh 7.1.12 Misalkan E adalah tiga persamaan dalam Contoh 7.1.6, kita dapat menunjukkan bahwa $x * e =_E x, x * i(x) =_E e, i(e) =_E e, i(i(x)) =_E x, i(x * y) =_E i(y) * i(x)$, dan seterusnya. Buktinya akan diberikan di bagian selanjutnya.

Diberikan dua suku, misalnya s dan t , apakah $s =_E t$? Ini adalah masalah keputusan penting dengan banyak aplikasi dalam matematika. Dalam teori komputasi, semua fungsi yang dapat dihitung dapat dibangun dengan persamaan. Sayangnya, secara umum tidak dapat diputuskan untuk menjawab $(s, t) \in =_E$; Dengan kata lain, kita tidak memiliki algoritma yang menerima E, s , dan t sebagai input dan mengembalikan "ya" jika $s =_E t$ (lihat Bab 11).

Dalam ilmu komputer, masalah penutupan kongruensi mengacu pada masalah memutuskan $s =_E t$, ketika E adalah himpunan persamaan dasar. Ini adalah masalah yang dapat diputuskan dan ada algoritma yang efisien. Ada sejumlah aplikasi yang menggunakan penutupan kongruensi. Diskusi rinci tentang topik ini dapat ditemukan di Bab 12.

7.2 SISTEM PENULISAN ULANG

Dalam Contoh 7.1.1, kita menunjukkan bagaimana membuktikan $f(f(a)) = a$ dari $f(a) = b$ dan $f(b) = a$, menggunakan resolusi dan aksioma kesamaan. Bukti yang mudah adalah mengganti $f(a)$ dalam $f(f(a)) = a$ dengan b untuk mendapatkan $f(b) = a$ dan kemudian mengganti $f(b)$ dengan a untuk mendapatkan $a = a$. Dalam contoh sederhana ini, kita memperlakukan $f(a) = b$ dan $f(b) = a$ sebagai dua aturan penulisan ulang, dan setiap penggantian adalah satu penerapan aturan penulisan ulang, yang disebut penulisan ulang.

Mengganti “sama dengan” dengan “sama dengan” adalah praktik umum dalam matematika dan logika. Misalnya, kita menggunakan relasi kesetaraan sebagai aturan penulisan ulang untuk menyederhanakan rumus ke bentuk normal. Aturan penulisan ulang bukan hanya sistem inferensi umum dalam matematika sehari-hari tetapi juga teknik komputasi yang sangat efisien, karena dapat melibatkan pencarian yang sangat sedikit. Bagian ini akan membahas secara singkat sistem penulisan ulang, termasuk diskusi tentang beberapa hasil teoritis yang menarik.

Aturan Penulisan Ulang

Mulai sekarang, kita akan menulis simbol predikat “=” sebagai “.”, sehingga “=” bukan lagi simbol predikat logika orde pertama. Oleh karena itu, persamaan $s = t$ ditulis sebagai $s . t$. Namun, $\neg(s = t)$ akan ditulis sebagai $s \neq t$. Ingat bahwa posisi suatu istilah adalah urutan bilangan bulat positif untuk menempatkan subistilah pada posisi tersebut (Definisi 5.1.4).

Definisi 7.2.1 Aturan penulisan ulang adalah pasangan terurut dari suku (l, r) , sering dilambangkan dengan $l \rightarrow r$, di mana l adalah sisi kiri dan r adalah sisi kanan dari aturan tersebut. Suatu suku t ditulis ulang menjadi t' dengan $l \rightarrow r$, yang dilambangkan dengan $t \Rightarrow t'$, jika terdapat substitusi σ dan posisi p dari t sedemikian sehingga $l\sigma = t/p$ dan $t' = t[p \sim r\sigma]$, di mana t/p menunjukkan subsuku dari t pada posisi p .

Secara matematis, baik persamaan maupun aturan penulisan ulang adalah pasangan suku. Keduanya sering disimpan sebagai pasangan suku dalam komputer.

Untuk menulis ulang t dengan $l \rightarrow r$, kita harus menemukan subsuku dari t pada posisi p dan substitusi σ sedemikian sehingga $t/p = l\sigma$. Dalam hal ini, kita mengatakan l cocok dengan t/p dan σ adalah kecocokannya. Pencocokan adalah penyatuan satu arah, dan menemukan pencocokan lebih mudah daripada menemukan penyatuan karena variabel dalam t tidak dapat dipengaruhi oleh σ . Jika kita memperlakukan variabel t sebagai konstanta, maka menemukan pencocokan sama dengan menemukan penyatuan t/p dan l . Setelah pencocokan ditemukan, kita perlu mengganti subistilah yang cocok dalam t dengan (contoh) sisi kanan aturan penulisan ulang. Perhatikan bahwa $t[p \sim s]$ menunjukkan istilah di mana subistilah pada posisi p , t/p , (lihat Definisi 5.1.4) diganti dengan s . Misalnya, jika $g(x) \rightarrow h(x)$ adalah aturan penulisan ulang, maka istilah $t = f(a, g(b))$ dapat ditulis ulang menjadi $t' = f(a, h(b))$, karena $t/1 = g(b) = g(x)\sigma$ dan $t'/1 = h(x)\sigma = h(b)$, di mana $\sigma = [x \sim b]$.

Definisi 7.2.2 Sistem penulisan ulang R adalah himpunan aturan penulisan ulang. Suatu istilah t ditulis ulang menjadi t' oleh R , yang dilambangkan dengan $t \Rightarrow_R t'$, atau hanya $t \Rightarrow t'$, jika terdapat aturan penulisan ulang dalam R yang menulis ulang t menjadi t' .

Sesuai konvensi, kita akan menggunakan $\Rightarrow_R^+$ dan $\Rightarrow_R^*$ untuk menunjukkan penutupan transitif dan penutupan refleksif dan transitif dari $\Rightarrow$, masing-masing. Artinya, $s \Rightarrow_R^+ t$ berarti bahwa s ditulis ulang menjadi t oleh R menggunakan satu atau lebih penulisan ulang; $s \Rightarrow_R^* t$ berarti bahwa s ditulis ulang menjadi t oleh R menggunakan 0 atau lebih penulisan ulang.

Meskipun penggunaan utama R adalah untuk merepresentasikan relasi kekongruenan, R dapat digunakan untuk merepresentasikan relasi transitif, stabil, dan monotonik apa pun. Ini mungkin berupa ketidaksetaraan (seperti $>$ atau $\geq$) atau implikasi.

Contoh 7.2.3 Pada Bab 1 (Bagian 1.3), kita telah melihat bahwa fungsi dapat dibangun dengan persamaan. Misalnya, kita menggunakan persamaan untuk mendefinisikan fungsi pendahulu, penjumlahan, pengurangan, dan perkalian pada bilangan asli. Kita dapat memperlakukan persamaan ini sebagai aturan penulisan ulang:

$$\begin{aligned} pre(0) &\rightarrow 0 \\ pre(s(x)) &\rightarrow x \\ add(0, y) &\rightarrow y \\ add(s(x), y) &\rightarrow s(add(x, y)) \\ sub(x, 0) &\rightarrow x \\ sub(x, s(y)) &\rightarrow sub(pre(x), y) \\ mul(0, y) &\rightarrow 0 \\ mul(s(x), y) &\rightarrow add(mul(x, y), y) \\ exp(x, 0) &\rightarrow s(0) \\ exp(x, s(y)) &\rightarrow mul(x, exp(x, y)) \end{aligned}$$

Untuk menghitung $2 \times 2 = 4$, kita menulis ulang $mul(s(s(0)), s(s(0)))$ menjadi $s(s(s(s(0))))$. Di bawah ini kami juga memberikan posisi untuk setiap penulisan ulang, dan Anda dapat menemukan aturan mana yang digunakan untuk setiap penulisan ulang:

$$\begin{aligned} &mul(s(s(0)), s(s(0))) && \text{at } \epsilon \\ \Rightarrow &add(mul(s(0), s(s(0))), s(s(0))) && \text{at } 1 \\ \Rightarrow &add(add(mul(0, s(s(0))), s(s(0))), s(s(0))) && \text{at } 1.1 \\ \Rightarrow &add(add(0, s(s(0))), s(s(0))) && \text{at } 1 \\ \Rightarrow &add(s(s(0)), s(s(0))) && \text{at } \epsilon \\ \Rightarrow &s(add(s(0), s(s(0)))) && \text{at } 1 \\ \Rightarrow &s(s(add(0, s(s(0)))) && \text{at } 1.1 \\ \Rightarrow &s(s(s(s(0)))) && \end{aligned}$$

Dalam bahasa ini, jika suatu istilah merupakan istilah dasar, tidak sulit untuk membuktikan dengan induksi bahwa proses penulisan ulang pada akhirnya akan berakhir dengan sebuah angka, yaitu istilah yang hanya berisi 0 dan s . Dengan menggunakan sistem penulisan ulang ini, kita dapat melakukan semua penjumlahan, pengurangan, dan perkalian pada bilangan asli dengan cara penulisan ulang, jika kita tidak mempermasalahkan kecepatannya.

Penghentian Penulisan Ulang

Sifat yang baik dari banyak sistem penulisan ulang, termasuk contoh sebelumnya, adalah bahwa penerapan aturan pada istilah, baik yang mendasar maupun tidak, tidak dapat berlangsung selamanya; pada akhirnya akan berakhir.

Definisi 7.2.4 Suatu sistem penulisan ulang R dikatakan berakhir jika tidak ada urutan tak hingga dari istilah $t_0, t_1, \dots, t_i, \dots$ sedemikian sehingga $t_i \Rightarrow t_{i+1}$.

Untuk membuktikan bahwa suatu sistem penulisan ulang R berakhir, kita dapat menggunakan urutan penyederhanaan $>$.

Proposisi 7.2.5 Misalkan $>$ adalah urutan penyederhanaan dan R adalah sistem penulisan ulang. Jika $l > r$ untuk setiap $l \rightarrow r$ dalam R , maka R berakhir.

Bukti Berdasarkan definisi, $>$ adalah urutan yang terdefinisi dengan baik, stabil, dan monoton pada istilah. Jika $t_i \Rightarrow t_{i+1}$ dengan $l \rightarrow r$, maka $t_i > t_{i+1}$, karena $l > r$, dan $>$ stabil dan monoton. Jika ada barisan tak hingga, $t_0, t_1, \dots, t_i, \dots$ sedemikian sehingga $t_i > t_{i+1}$, maka $>$ tidak dapat berdasar dengan baik.

Proposisi di atas memberikan kondisi yang cukup untuk penghentian sistem penulisan ulang. Pada Bagian 6.3, kita membahas urutan jalur leksikografis (LPO), urutan jalur rekursif (RPO), dan urutan Knuth-Bendix (KBO). Urutan-urutan ini adalah urutan penyederhanaan dan dapat digunakan untuk membuktikan penghentian R .

Contoh 7.2.6 Untuk menunjukkan penghentian R dalam Contoh 7.2.3, kita dapat menggunakan urutan jalur leksikografis $>_{lpo}$ dengan prioritas $s < pre < add < sub < mul$ dan setiap operator biner memiliki status kiri-ke-kanan, kecuali sub : sub harus memiliki status kanan-ke-kiri (Definisi 6.3.16), karena kita menginginkan $[x, s(y)] >^{lex} [pre(x), y]$.

Secara umum, bagaimana kita dapat menunjukkan bahwa proses penulisan ulang menggunakan R akan berakhir? Sayangnya, ini adalah masalah yang tidak dapat diputuskan. Artinya, kita tidak memiliki algoritma untuk menjawab pertanyaan ini untuk R secara umum.

Definisi 7.2.7 Diberikan sistem penulisan ulang R , jika $s \Rightarrow^* t$ dan tidak ada t' sedemikian sehingga $t \Rightarrow_R t'$, kita katakan t adalah bentuk normal dari s di R .

Mengapa terminasi merupakan sifat penting dari sistem penulisan ulang? Terminasi R memastikan bahwa ada bentuk normal untuk setiap suku. Tanpa terminasi, kita akan kesulitan mengendalikan penulisan ulang. Dalam teori komputasi, terminasi adalah pembagi untuk menentukan apakah suatu masalah keputusan dapat diputuskan atau tidak. Beberapa persamaan, seperti komutativitas $+$ atau $\vee$, tidak pernah dapat dibuat menjadi aturan terminasi.

Tidak dapat diputuskan untuk menentukan apakah suatu sistem penulisan ulang bersifat terminasi atau tidak.

Konfluensi Penulisan Ulang

Terminasi adalah salah satu sifat penting dari seperangkat aturan penulisan ulang; sifat penting lainnya adalah konfluensi.

Contoh 7.2.8 Dalam Contoh 7.1.6, kita menggunakan tiga persamaan untuk menentukan sebuah grup $(S, *)$. Persamaan-persamaan ini dapat dibuat menjadi sistem penulisan ulang R :

- (1) $e * x \rightarrow x$
- (2) $i(x) * x \rightarrow e$
- (3) $(x * y) * z \rightarrow x * (y * z)$

Ada dua cara bagi R untuk menulis ulang $(i(x) * x) * y$:

$$\underline{(i(x) * x)} * y \Rightarrow \begin{cases} (3) i(x) * (x * y) \\ (2) \underline{e} * y \Rightarrow (1)y \end{cases}$$

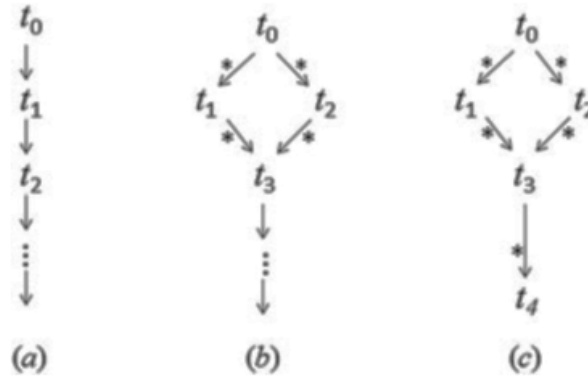
Artinya, ada lebih dari satu cara untuk menulis ulang suatu istilah. Untuk $(i(x) * x) * y$, kita memiliki dua bentuk normal: $i(x) * (x * y)$ dan y .

Definisi 7.2.9 Suatu sistem penulisan ulang R dikatakan konfluen jika, untuk setiap istilah s , jika $s \Rightarrow_R^* t_1$ dan $s \Rightarrow_R^* t_2$, maka terdapat suatu istilah t' sedemikian sehingga $t_1 \Rightarrow_R^* t'$ dan $t_2 \Rightarrow_R^* t'$.

R dikatakan kanonik jika R bersifat terminasi dan konfluen.

Gambar 7.1 mengilustrasikan konsep “terminasi,” “konfluen,” dan “kanonik.” Artinya, jika R konfluen, tidak masalah bagaimana Anda menerapkan aturan pada suatu istilah, Anda akan mendapatkan hasil yang sama. Terdapat beberapa konsep “konfluensi” yang diusulkan dalam literatur. Konsep-konsep ini setara untuk sistem penulisan ulang yang berakhiran. Terminasi R memastikan keberadaan bentuk normal untuk setiap suku; konfluensi R memastikan keunikan bentuk normal. Ketika R bersifat kanonik, Anda tidak perlu khawatir

tentang posisi mana dan memilih aturan mana untuk menulis ulang suatu suku. Pilihan apa pun dapat dibuat, dan hasil akhirnya akan sama. Mari kita nyatakan bentuk normal unik dari t dalam sistem penulisan ulang kanonik dengan $t \downarrow R$, bentuk kanonik dari t .



Gambar 7.1 Ilustrasi (a) urutan penulisan ulang yang tidak berakhiran. (b) konfluen: jika $t_0 \Rightarrow^* t_1$ dan $t_0 \Rightarrow^* t_2$, maka terdapat t_3 sedemikian sehingga $t_1 \Rightarrow^* t_3$ dan $t_2 \Rightarrow^* t_3$. (c) kanonik: jika $t_0 \Rightarrow^* t_1$ dan $t_0 \Rightarrow^* t_2$, maka terdapat bentuk normal t_4 yang unik sedemikian sehingga $t_1 \Rightarrow^* t_4$ dan $t_2 \Rightarrow^* t_4$

Definisi 7.2.10 Suatu sistem penulisan ulang R adalah prosedur keputusan untuk himpunan persamaan E jika (a) R bersifat terminasi dan (b) untuk setiap dua suku s dan t , $s =_E t$ jika dan hanya jika $s \downarrow_R = t \downarrow_R$.

Kita katakan R adalah prosedur keputusan untuk E , yaitu, R adalah prosedur keputusan untuk masalah kekongruenan (Definisi 7.1.7), karena perhitungan bentuk normal dalam R bersifat terminasi dan pemeriksaan kesamaan $s =_E t$ direduksi menjadi perbandingan bentuk normal s dan t .

Teorema 7.2.11 Misalkan R adalah sistem penulisan ulang kanonik yang memenuhi kondisi berikut:

1. Untuk setiap aturan penulisan ulang $l \rightarrow r$ dari R , $l =_E r$.
 2. Untuk setiap persamaan $s_1 = s_2$ dari E , $s_1 \downarrow_R = s_2 \downarrow_R$.
- Kemudian, R adalah prosedur pengambilan keputusan untuk E .

Bukti Karena R bersifat kanonik, maka R harus berakhir. Kita hanya perlu menunjukkan bahwa kondisi yang diberikan menyiratkan bahwa $s =_E t$ jika dan hanya jika $s \downarrow_R = t \downarrow_R$ untuk setiap dua suku s dan t .

Untuk setiap aturan $l \rightarrow r$ dari R , karena $l =_E r$, jika $s \Rightarrow t$ dengan $l \rightarrow r$, maka $s =_E t$ dengan menerapkan aturan instansiasi dan monotonisitas pada $l =_E r$. Menerapkan hasil ini pada setiap penulisan ulang, kita memiliki $s =_E s \downarrow_R$ dan $t =_E t \downarrow_R$. Jika $s \downarrow_R = t \downarrow_R$, maka $s =_E t$.

Jika $s =_E t$, periksa keenam kondisi untuk mendapatkan $s =_E t$: jika $s =_E t$ karena $s = t \in E$, maka $s \downarrow R = t \downarrow R$ berdasarkan kondisi kedua. Jika $s =_E t$ karena $t =_E s$, maka $t \downarrow R = s \downarrow R$ menyiratkan $s \downarrow R = t \downarrow R$. Jika $s =_E t$ karena $s =_E r$ dan $r =_E t$, maka $s \downarrow R = r \downarrow R = t \downarrow R$. Jika $s = f(\dots, s_i, \dots) =_E f(\dots, t_i, \dots) = t$ karena $s_i =_E t_i$, maka $s_i \downarrow R = t_i \downarrow R$ menyiratkan $s \downarrow R = t \downarrow R$. Jika $s = s'\sigma =_E t'\sigma = t$ karena $s' =_E t'$, maka $s' \downarrow R = t' \downarrow R$ menyiratkan $s \downarrow R = t \downarrow R$. Dalam semua kasus, kita memiliki $s \downarrow R = t \downarrow R$ jika $s =_E t$.

Bahkan ketika sistem penulisan ulang bersifat kanonik, kita mungkin masih ingin berhati-hati dalam memilih aturan dan posisi dalam suatu istilah untuk menerapkan aturan, guna meminimalkan jumlah upaya komputasi yang diperlukan untuk menemukan bentuk kanonik. Kita mungkin hanya ingin mengeksplorasi langkah-langkah penulisan ulang terpendek. Misalnya, dalam memilih posisi untuk penulisan ulang, kita dapat menggunakan strategi posisi terluar terlebih dahulu atau strategi posisi terdalam terlebih dahulu. Pemeriksaan yang cermat terhadap aturan yang akan diterapkan dapat mengungkapkan strategi mana yang bekerja lebih baik.

Prosedur Penyelesaian Knuth-Bendix

Pada Contoh 7.2.8, kita telah melihat bahwa suku $t = (i(x) * x) * y$ memiliki dua bentuk normal, yaitu $i(x) * (x * y)$ dan y . Sistem penulisan ulang pada contoh tersebut tidak konfluen. Pada tahun 1970, Donald Knuth dan Peter Bendix memperkenalkan sebuah prosedur yang mengambil sekumpulan persamaan dan urutan penyederhanaan sebagai masukan dan membuat aturan penulisan ulang dari persamaan sesuai dengan urutan penyederhanaan. Pasangan kritis kemudian dihitung dari aturan penulisan ulang. Jika semua pasangan kritis memiliki bentuk normal yang sama, kita memperoleh sistem penulisan ulang kanonik. Jika tidak, pasangan kritis tersebut ditambahkan ke dalam kumpulan persamaan dan proses berlanjut.

Definisi 7.2.12 Diberikan dua aturan penulisan ulang yang tidak memiliki variabel yang sama dan mungkin hanya berbeda pada nama variabel, misalnya $l_1 \rightarrow r_1$ dan $l_2 \rightarrow r_2$, pasangan kritis dari kedua aturan tersebut adalah pasangan suku (s, t) , di mana $s = r_1\sigma$, $t = l_1[p \sim r_2]\sigma$, p adalah posisi non-variabel dari l_1 , dan σ adalah mgu dari l_1/p dan l_2 . Kita mengatakan l_2 disuperposisikan pada p dari l_1 untuk menghasilkan pasangan kritis dan proses ini disebut superposisi.

Dalam Contoh 7.2.8, kita diberikan tiga aturan penulisan ulang. Jika kita mengganti nama variabel dalam aturan 3 menjadi $(x_3 * y_3) * z_3 \rightarrow x_3 * (y_3 * z_3)$, kita dapat menghitung pasangan kritis dari aturan 3 dan aturan 2, dengan menumpangkan $l_2 = i(x) * x$ pada posisi 1 dari l_3 , yaitu, menyatukan l_2 dan $l_3/1 = x_3 * y_3$ dengan mgu $\sigma = [x_3 \sim i(x), y_3 \sim x, z_3 \sim y]$ dan menghasilkan pasangan kritis $(i(x) * (x * y), e * y)$ dari $l_3\sigma = (i(x) * x) * y$ seperti yang diilustrasikan dalam Contoh 7.2.8.

Selama perhitungan pasangan kritis, dengan menerapkan mgu σ ke l_1 , $l_1\sigma$ dapat ditulis ulang menjadi $r_1\sigma$ dengan $l_1 \rightarrow r_1$. Pada saat yang sama, karena $l_1/p\sigma = l_2\sigma$, $l_1\sigma$ mengandung sebuah contoh l_2 pada posisi p , sehingga $l_1\sigma$ juga dapat ditulis ulang menjadi $l_1[p \sim l_2]\sigma$ dengan $l_2 \rightarrow r_2$. Dengan kata lain, pasangan kritis diperoleh dengan dua penulisan ulang pada $l_1\sigma$ menggunakan dua aturan tersebut, masing-masing:

$$l_1\sigma \Rightarrow \begin{cases} r_1\sigma & \text{by } l_1 \rightarrow r_1 \\ l_1[p \leftarrow r_2]\sigma & \text{by } l_2 \rightarrow r_2 \end{cases}$$

Lemma 7.2.13 Jika $l_1 =_E r_1$, $l_2 =_E r_2$, dan (s, t) adalah pasangan kritis dari $l_1 \rightarrow r_1$ dan $l_2 \rightarrow r_2$, maka $s =_E t$.

Bukti Dari $l_1 =_E r_1$, kita memiliki $l_1\sigma =_E r_1\sigma$. Dari $l_2 =_E r_2$ dan $l_1/p\sigma = l_2\sigma$, kita memiliki $l_1\sigma =_E l_1[p \sim r_2]\sigma$. Oleh karena itu, $s = r_1\sigma =_E l_1[p \sim r_2]\sigma = t$.

Contoh 7.2.14 Kita akan menunjukkan bagaimana memperoleh sistem penulisan ulang kanonik dari tiga aturan penulisan ulang dalam Contoh 7.2.8:

- (1) $e * x \rightarrow x$
- (2) $i(x) * x \rightarrow e$
- (3) $(x * y) * z \rightarrow x * (y * z)$

Urutan jalur leksikografis digunakan dengan prioritas $e < * < i$. Saat menghitung pasangan kritis dari dua aturan, jika nama variabel muncul di kedua aturan, kita mengasumsikan bahwa nama variabel di salah satu aturan secara implisit diikutsertakan oleh nomor aturan. Misalnya, jika (2) dan (3) sama-sama mengandung x , maka x dalam (3) diasumsikan sebagai x_3 (y dan z tidak diubah namanya karena tidak muncul dalam (2)). Jika kedua aturan berasal dari aturan penulisan ulang yang sama, maka satu aturan adalah salinan dari aturan lainnya dengan semua variabel diubah namanya.

Pasangan kritis pertama adalah $(i(x) * (x * y), e * y)$, yang berasal dari (2) dan (3) dan $i(x) * (x * y) = e * y$ dianggap sebagai persamaan. Setelah menulis ulang $e * y$ menjadi y , persamaan ini diorientasikan ke dalam aturan penulisan ulang keempat:

$$(4) i(x) * (x * y) \rightarrow y$$

Pasangan kritis berasal dari aturan (4) (yaitu, $i(x_4) * (x_4 * y_4) \rightarrow y_4$ adalah $l' \rightarrow r'$) dan dirinya sendiri (yaitu, $i(x) * (x * y) \rightarrow y$ adalah $l_4 \rightarrow r_4$), dan mgu dari $l'/2$ dan l_4 adalah $\sigma = [x_4 \sim i(x), y_4 \sim x * y]$:

$$i(i(x)) * \underline{i(x) * (x * y)} \Rightarrow \begin{cases} (4) x * y \\ (4) i(i(x)) * \underline{y} \end{cases}$$

Pasangan kritisnya adalah $x * y = i(i(x)) * y$, yang menghasilkan aturan kelima:

$$(5) i(i(x)) * y \rightarrow x * y$$

Pasangan kritis berasal dari aturan (4) dan (2), dan mgu dari $l_4/2$ dan l_2 adalah $\sigma = [x_4 \sim i(x), y_4 \sim x]$:

$$i(i(x)) * \underline{i(x) * x} \Rightarrow \left\{ \begin{array}{l} (4) x \\ (2) i(i(x)) * \underline{e} \end{array} \right. \rightarrow (5) x * e$$

Pasangan kritisnya adalah $x \doteq x * e$, yang menghasilkan aturan keenam:

$$(6) x * e \rightarrow x$$

Pasangan kritis berasal dari aturan (6) dan (2) dengan $\sigma = [x_6 \sim i(e)]$ dan menghasilkan aturan ketujuh:

$$(7) i(e) \rightarrow e$$

Pasangan kritis berasal dari aturan (5) dan (2) dengan $\sigma = [x_2 \sim i(x), y \sim i(x)]$:

$$i(i(x)) * i(x) \Rightarrow \left\{ \begin{array}{l} (5) x * i(x) \\ (2) e \end{array} \right.$$

dan menghasilkan aturan kedelapan

$$(8) x * i(x) \rightarrow e$$

Pasangan kritis berasal dari aturan (6) dan (5) dengan $\sigma = [x_6 \sim i(i(x)), y \sim e]$:

$$i(i(x)) * e \Rightarrow \left\{ \begin{array}{l} (6) i(i(x)) \\ (5) x * e \end{array} \right. \rightarrow (6) x$$

dan menghasilkan aturan kesembilan

$$(9) i(i(x)) \rightarrow x$$

Pasangan kritis berasal dari aturan (4) dan (3), dan mgu dari $l_4/2$ dan l_3 adalah $\sigma = [x_4 \sim x * y, y_4 \sim z]$:

$$i(x * y) * \underline{((x * y) * z)} \Rightarrow \begin{cases} (4) z \\ (3) i(x * y) * \underline{(x * (y * z))} \end{cases}$$

dan menghasilkan aturan kesepuluh

$$(10) i(x * y) * (x * (y * z)) \rightarrow z$$

Pasangan kritis berasal dari aturan (10) dan (8), dan mgu dari $l_{10}/2.2$ dan l_8 adalah $\sigma = [x_8 \sim y, z \sim i(y)]$:

$$i(x * y) * (x * \underline{(y * i(y))}) \Rightarrow \begin{cases} (10) i(y) \\ (8) i(x * y) * (x * \underline{e}) \Rightarrow (6) i(x * y) * x \end{cases}$$

dan menghasilkan aturan kesebelas

$$(11) i(x * y) * x \rightarrow i(y)$$

Pasangan kritis berasal dari aturan (11) dan (4), dan mgu dari $l_{11}/1.1$ dan l_4 adalah $\sigma = [x_{11} \sim i(x), y_{11} \sim x * y]$:

$$i(\underline{i(x) * (x * y)}) * i(x) \Rightarrow \begin{cases} (11) i(x * y) \\ (4) i(\underline{y}) * i(x) \end{cases}$$

dan menghasilkan aturan kedua belas

$$(12) i(x * y) \rightarrow i(y) * i(x)$$

Pasangan kritis terakhir berasal dari aturan (4) dan (5) dan menghasilkan aturan ketiga belas:

$$(13) x * (i(x) * y) \rightarrow y$$

Karena aturan (5), (10), dan (11) dapat ditulis ulang menjadi identitas, yaitu $t = t$, dengan aturan (9) dan (12) (dan kemudian dengan aturan lainnya), maka himpunan akhir aturan penulisan ulang adalah

$$(1) e * x \rightarrow x$$

$$(2) i(x) * x \rightarrow e$$

$$(7) i(e) \rightarrow e$$

$$(8) x * i(x) \rightarrow e$$

$$(3) (x * y) * z \rightarrow x * (y * z)$$

$$(4) i(x) * (x * y) \rightarrow y$$

$$(6) x * e \rightarrow x$$

$$(9) i(i(x)) \rightarrow x$$

$$(12) i(x * y) \rightarrow i(y) * i(x)$$

$$(13) x * (i(x) * y) \rightarrow y$$

Semua pasangan kritis dari sepuluh aturan ini dapat direduksi menjadi identitas oleh sistem penulisan ulang ini.

Contoh di atas mengilustrasikan eksekusi prosedur penyelesaian Knuth-Bendix, yang diciptakan oleh Donald Knuth dan Peter Bendix, yang dijelaskan oleh pseudo-kode berikut.

Prosedur 7.2.15 Prosedur *KnuthBendix*($E, >$) mengambil himpunan persamaan E dan urutan penyederhanaan $>$ sebagai masukan dan menghasilkan sistem penulisan ulang kanonik dari E ketika berhasil. Prosedur ini memanggil empat prosedur:

- *pickEquation*(E) memilih persamaan dari E .
- *NF*(t, R) mengembalikan bentuk normal dari suku t oleh R .
- *rewritable*(t, r) memeriksa apakah aturan penulisan ulang r dapat menulis ulang suku t .
- *criticalPairs*(r, R) mengembalikan himpunan semua pasangan kritis (dalam bentuk persamaan) antara aturan r dan setiap aturan dalam R .

proc *KnuthBendix*($E, >$)

```

1   $R := \emptyset$ 
2  while ( $E \neq \emptyset$ ) do
3    ( $s \doteq t$ ) := pickEquation( $E$ );  $E := E - \{(s \doteq t)\}$ 
4     $s := NF(s, R)$ ;  $t := NF(t, R)$  // normalize  $s$  and  $t$  by  $R$ .
5    if ( $s = t$ ) continue //identity is discarded.
6    else if ( $s > t$ )  $r := (s \rightarrow t)$ 
7    else if ( $t > s$ )  $r := (t \rightarrow s)$ 
8    else return "failure" //  $s$  and  $t$  are not comparable by  $>$ .
9     $R := R \cup \{r\}$  // add new rule into  $R$ 
10   for ( $d \rightarrow e$ )  $\in R - \{r\}$  do // inter-reduction
11     if rewritable( $d, r$ )  $R := R - \{d \rightarrow e\}$ ;  $E := E \cup \{d \doteq e\}$ 
12     if rewritable( $e, r$ )  $R := R - \{d \rightarrow e\} \cup \{(d \rightarrow NF(e, R))\}$ 
13    $E := E \cup \text{criticalPairs}(r, R)$ 
14 return  $R$ 
```

Contoh 7.2.14 adalah contoh sukses dari prosedur di atas yang mengambil tiga persamaan dan menghasilkan sistem penulisan ulang sepuluh aturan. Sistem penulisan ulang ini cukup untuk menyelesaikan masalah kata untuk grup bebas tanpa sifat lain. Masalah kata dalam aljabar abstrak adalah menemukan prosedur keputusan untuk memutuskan apakah dua suku sama menurut aksioma teori grup. Sistem penulisan ulang yang dikembalikan oleh prosedur di atas dapat berfungsi sebagai prosedur keputusan dengan memeriksa apakah kedua suku memiliki bentuk kanonik yang sama.

Penyelesaian Knuth-Bendix bukanlah algoritma karena dapat berlangsung selamanya karena kumpulan aturan dan pasangan kritis yang terus bertambah. Perhatikan bahwa prosedur dapat gagal jika suatu persamaan tidak dapat diorientasikan ke dalam aturan penulisan ulang. Ada banyak upaya untuk mengurangi kasus kegagalan. Misalnya, jika persamaan seperti $f(x, g(x))$.

$h(y, g(y))$ tidak dapat diorientasikan, kita dapat memperkenalkan konstanta baru c untuk membuat dua persamaan. $f(x, g(x)) = c$ dan $h(y, g(y)) = c$. Secara umum, jika kita memiliki persamaan $s(x, z) = t(y, z)$, di mana x tidak muncul dalam t dan y tidak muncul dalam s , kita dapat memperkenalkan fungsi baru $h(z)$ untuk membuat dua persamaan baru: $s(x, z) = h(z)$ dan $t(y, z) = h(z)$. Untuk aksioma, kita dapat mempertimbangkan penulisan ulang atas $T(F, X)/=_{AC}$, yang disebut penulisan ulang modulo AC (asosiatif dan komutatif).

Definisi 7.2.16 Suatu sistem penulisan ulang R dikatakan saling direduksi jika untuk setiap aturan $l \rightarrow r \in R$, l tidak dapat ditulis ulang oleh $R - \{l \rightarrow r\}$; R dikatakan tereduksi jika l dan r tidak dapat ditulis ulang oleh $R - \{l \rightarrow r\}$.

Lemma 7.2.17 Jika $KnuthBendix(E, >)$ mengembalikan R , maka R dikatakan tereduksi.

Bukti Baris 11 (atau 12) memastikan sisi kiri (atau kanan) dari setiap aturan dalam R tidak dapat ditulis ulang oleh aturan lain.

Ketika KnuthBendix berhasil diakhiri, kita memperoleh sistem penulisan ulang kanonik yang tereduksi.

Teorema 7.2.18 Jika $KnuthBendix(E, >)$ mengembalikan R , maka R dikatakan tereduksi dan kanonik.

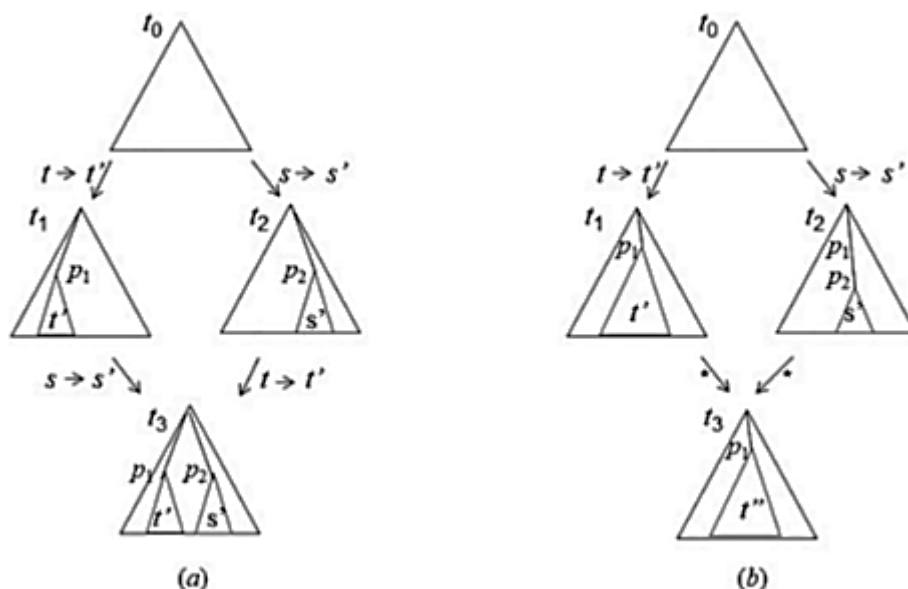
Bukti R dikatakan tereduksi oleh Lemma 7.2.17. Untuk kanonisitas, pada awalnya, R bersifat terminasi, karena untuk setiap $l \rightarrow r$ dari R , $l > r$, di mana $>$ adalah urutan penyederhanaan. Kedua, semua pasangan kritis R dihitung, dan setiap pasangan ditulis ulang ke suku yang sama oleh R (jika tidak, aturan baru akan ditambahkan ke R).

Untuk menunjukkan bahwa R konfluen, kita melakukan induksi pada t_0 berdasarkan $>$. Sebagai kasus dasar, jika t_0 tidak dapat ditulis ulang oleh R , maka R konfluen secara trivial pada t . Misalkan, untuk $i = 1, 2$, $t_0 \Rightarrow t_i$ dengan $l_i \rightarrow r_i \in R$ pada posisi p_i dari t , yaitu, $t_i = t[p_i \sim r_i \sigma_i]$.

Kasus 1: Jika p_1 dan p_2 adalah posisi yang terpisah, maka t_1 dan t_2 dapat ditulis ke suku umum $t_3 = t_0[p_1 \sim r_1 \sigma_1, p_2 \sim r_2 \sigma_2]$.

Kasus 2: Jika p_1 dan p_2 bukan posisi yang terpisah, tanpa mengurangi keumuman, anggap $p_1 = \epsilon$, maka $t_0/p_1 = t_0 = l_1 \sigma_1$, dan $t_0/p_2 = (l_1 \sigma_1)/p_2 = l_2 \sigma_2$. Ada dua subkasus yang perlu dipertimbangkan:

Kasus 2.1: Jika p_2 adalah posisi non-variabel dari l_1 , maka $(l_1\sigma_1)/p_2 = (l_1/p_2)\sigma_1 = l_2\sigma_2$, yaitu, l_1/p_2 dan l_2 dapat disatukan. Dengan demikian, terdapat pasangan kritis (s_1, s_2) antara kedua aturan sedemikian rupa sehingga $t_1 = s_{1\theta}$ dan $t_2 = s_{2\theta}$ untuk beberapa substitusi θ . Karena semua pasangan kritis ditulis ulang ke istilah yang sama oleh R , demikian pula (s_1, s_2) , serta (t_1, t_2) .



Gambar 7.2 Ilustrasi pembuktian Teorema 7.2.18: (a) dua posisi terpisah; (b) dua posisi tumpang tindih

Kasus 2.2: Ketika p_2 bukan posisi non-variabel dari l_1 , yaitu, baik p_2 atau awalan dari p_2 adalah posisi variabel dari l_1 . Sketsa pembuktiannya adalah sebagai berikut: kita dapat menemukan penerus umum dari t_1 dan t_2 dengan terlebih dahulu menerapkan $l_2 \rightarrow r_2$ ke t_2 sebanyak $n - 1$ kali jika variabel pada posisi l_1 tersebut muncul sebanyak n kali dan kemudian menerapkan $l_1 \rightarrow r_1$ sekali. Sebuah contoh yang menunjukkan kasus ini diberikan setelah pembuktian.

Dengan hipotesis induksi, R konfluen pada t_1 dan t_2 , masing-masing, karena $t_0 > t_1$ dan $t_0 > t_2$. Sekarang, kita telah menunjukkan bahwa t_1 dan t_2 ditulis ulang menjadi istilah umum dalam semua kasus; R harus konfluen pada t_0 .

Kedua kasus dalam pembuktian ini diilustrasikan pada Gambar 7.2, di mana $t \rightarrow t'$ adalah $l_1 \rightarrow r_1$ dan $s \rightarrow s'$ adalah $l_2 \rightarrow r_2$.

Contoh 7.2.19 Untuk mengilustrasikan kasus 2.2 dalam pembuktian di atas, misalkan $t = i(i(i(a))) * i(i(a))$ dan R adalah sepuluh aturan dalam Contoh 7.2.14. Kemudian, t dapat ditulis ulang menjadi e dengan aturan 2 dan menjadi $i(a) * i(i(a))$ dengan aturan 9 pada posisi 1.1. Sisi kiri aturan 2 adalah $i(x) * x$, di mana x muncul dua kali ($n = 2$).

Untuk mendapatkan suku yang sama, kita menerapkan aturan 9 sekali lagi ($n - 1 = 1$) pada $i(a) * i(i(a))$ di posisi 2 untuk mendapatkan $i(a) * a$, yang dapat ditulis ulang menjadi e dengan aturan 2.

$$i(i(i(a))) * i(i(a)) \Rightarrow \begin{cases} (2) & e \\ (9) & i(a) * i(i(a)) \end{cases} \rightarrow (9) \ i(a) * a \rightarrow (2) \ e$$

Perhatikan bahwa baik 1.1 dan 2 dari t adalah posisi variabel dari $i(x) * x$, sisi kiri aturan 2.

Teorema 7.2.20 Jika $KnuthBendix(E, >)$ mengembalikan R , maka R adalah prosedur keputusan untuk E .

Bukti (Sketsa) Teorema sebelumnya memberi tahu kita bahwa R bersifat kanonik. Berdasarkan Teorema 7.2.11, kondisi 1 dipastikan oleh Lemma 7.2.13; kondisi 2 berlaku karena setiap persamaan E diubah menjadi aturan atau disederhanakan menjadi identitas.

Kesimpulannya, sifat-sifat yang diinginkan dari sistem penulisan ulang adalah terminasi dan konfluensi. Sistem penulisan ulang dapat ditunjukkan untuk terminasi dengan bantuan urutan penyederhanaan dan konfluensi dengan melihat pasangan kritis yang dibentuk dari pasangan aturan. Pada Bab 11, kita akan melihat bahwa secara umum tidak dapat diputuskan apakah suatu sistem penulisan ulang terminasi atau tidak. Dalam praktiknya, penyelesaian Knuth-Bendix seringkali berjalan selamanya atau berhenti dengan kegagalan untuk banyak input.

Sistem Penulisan Ulang Khusus

Sistem penulisan ulang dapat digunakan untuk menempatkan istilah dalam bentuk normal, untuk membuktikan istilah sama atau setara, atau untuk berfungsi sebagai prosedur pengambilan keputusan untuk beberapa relasi monoton transitif. Penerapannya mewakili metode penalaran matematika yang ampuh, karena terkadang dapat mengatasi ledakan kombinatorial yang disebabkan oleh prosedur pembuktian lainnya, misalnya, resolusi. Berikut ini, kita akan secara singkat memperkenalkan beberapa kasus khusus dari sistem penulisan ulang.

Sistem Penulisan Ulang untuk Persamaan Dasar

Persamaan dasar adalah sepasang istilah dasar, yaitu, tidak ada variabel yang muncul dalam persamaan. Penyelesaian Knuth-Bendix dapat sangat disederhanakan pada persamaan dasar karena hal-hal berikut:

- Terdapat urutan penyederhanaan total untuk istilah dasar. Ini dapat dicapai dengan mendefinisikan prioritas total atas simbol fungsi dan kemudian menerapkan salah satu urutan penyederhanaan yang dikenal, seperti urutan Knuth-Bendix (KBO), pada istilah-istilah tersebut.

- Perhitungan pasangan kritis tidak diperlukan jika sistem penulisan ulang saling direduksi. Artinya, jika terdapat pasangan kritis antara dua aturan penulisan ulang, maka satu aturan dapat menulis ulang sisi kiri aturan lainnya.

Contoh 7.2.21 Misalkan f^3a dan f^5a masing-masing menyatakan $f(f(f(a)))$ dan $f(f(f(f(f(a)))))$. Dengan memasukkan $E = \{f^5a \doteq a, f^3a \doteq a\}$ ke dalam penyelesaian Knuth-Bendix, aturan penulisan ulang yang dibuat dari E seharusnya adalah $\{(1)f^5a \rightarrow a, (2)f^3a \rightarrow a\}$. Karena $f^5a = f^2f^3a$, pasangan kritis dari (1) dan (2) adalah $f^2a \doteq a$, dan ini sama dengan kita menulis ulang (1) menjadi $f^2a \doteq a$ dengan (2). Bahkan, penulisan ulang terjadi terlebih dahulu (baris 11) dalam prosedur Knuth-Bendix. Dari $f^2a \doteq a$, kita memiliki (3) $f^2a \rightarrow a$, yang menulis ulang (2) menjadi $f^1a \doteq a$. Penyelesaian Knuth-Bendix berhenti pada $\{f^1a \rightarrow a\}$.

Teorema 7.2.22 Untuk setiap himpunan E persamaan dasar, terdapat sistem penulisan ulang kanonik R yang berfungsi sebagai prosedur pengambilan keputusan dari E .

Kami serahkan pembuktian teorema ini sebagai latihan.

Sistem Penulisan Ulang String

Dalam ilmu komputer teoretis dan logika matematika, sistem penulisan ulang string (SRS), yang secara historis disebut sistem semi-Thue, adalah sistem penulisan ulang atas string dari suatu alfabet.

Contoh 7.2.23 $R = \{(1) ba \rightarrow ab, (2) ca \rightarrow ac, (3) cb \rightarrow bc\}$ adalah SRS atas alfabet $\Sigma = \{a, b, c\}$. Kita dapat menulis ulang cba oleh R menjadi abc dengan urutan penulisan ulang berikut:

$$cba \Rightarrow (1) cab \Rightarrow (2) acb \Rightarrow (3) abc$$

Kita juga dapat menerapkan (3) terlebih dahulu ke cba dan mencapai hasil yang sama.

Secara formal, SRS didefinisikan sebagai $P = (\Sigma, R)$, di mana Σ adalah alfabet dan R adalah himpunan bagian terbatas dari $\Sigma^* \times \Sigma^*$. Setiap anggota (s, t) dari R disebut aturan penulisan ulang dan ditulis sebagai $s \rightarrow t$.

Sistem penulisan ulang istilah (TRS) lebih ekspresif daripada SRS karena SRS mudah dikonversi menjadi TRS, di mana setiap simbol s dalam SRS digantikan oleh fungsi uner $s/1$ dalam TRS. Misalnya, aturan penulisan ulang $ba \rightarrow ab$ dalam R dari contoh di atas digantikan oleh $b(a(x)) \rightarrow a(b(x))$. Semua konsep penting TRS, seperti terminasi dan konfluensi, dapat dibawa ke SRS. Misalnya, mudah untuk menunjukkan bahwa R dalam contoh di atas adalah sistem penulisan ulang kanonik.

Untuk TRS, sistem penulisan ulang mendefinisikan relasi biner transitif, monotonik, dan stabil atas istilah. Untuk SRS, sistem penulisan ulang mendefinisikan relasi biner transitif dan monotonik atas Σ^* . Artinya, diberikan himpunan R aturan penulisan ulang, yang merupakan relasi biner $\rightarrow$ antara string tetap di atas Σ^* SRS memperluas relasi penulisan ulang ke semua string di mana sisi kiri dan kanan aturan muncul sebagai substring, yaitu, $usv \Rightarrow utv$ dari $s \rightarrow t$, di mana s, t, u , dan v adalah string dari Σ^* . Jika kita mempertimbangkan penutupan simetris dari $\Rightarrow$, maka R disebut sistem *Thue*, yang namanya berasal dari matematikawan Norwegia Axel Thue, yang memperkenalkan perlakuan sistematis SRS pada tahun 1914.

Penulisan Ulang dengan Persamaan Polinomial

Setiap pembaca buku ini telah melihat persamaan polinomial dalam aljabar sekolah menengah. Misalnya, berikut ini adalah sistem persamaan polinomial:

$$(1) x^3 + 4x^2 - 4xz - 8 = 0$$

$$(2) x^2y + 4xy - 4yz + 4 = 0$$

Kita akan menggunakan contoh ini untuk mengilustrasikan semua konsep tentang penulisan ulang dengan polinomial.

Dalam aljabar abstrak, suatu medan adalah himpunan elemen yang tertutup terhadap operasi aritmatika $+$, $-$, $*$, dan $/$. Himpunan bilangan rasional adalah bidang yang paling dikenal. Diberikan himpunan n variabel, misalnya $x_1, \dots, x_n$, sebuah monomial adalah hasil perkalian $cx_1^{a_1}x_2^{a_2} \cdots x_n^{a_n}$, dimana $a_i \in \mathbb{N}$ dan c , yang disebut koefisien, adalah elemen dari medan yang diberikan. Monomial ini dapat direpresentasikan oleh daftar $[c, a_1, a_2, \dots, a_n]$. Polinomial adalah jumlah dari satu atau lebih monomial. Sistem persamaan polinomial, atau hanya sistem polinomial, adalah himpunan persamaan simultan $f_1 = 0, \dots, f_m = 0$, di mana setiap $f_i, 1 \leq i \leq m$, adalah polinomial. Sistem polinomial biasanya diklasifikasikan berdasarkan jumlah variabel dan jumlah maksimum variabel dalam setiap monomial. Misalnya, pemrograman linier hanya mempertimbangkan sistem polinomial dengan paling banyak satu variabel dalam setiap monomial.

Diberikan dua monomial yang diwakili oleh $M = [c_1, a_1, a_2, \dots, a_n]$ dan $N = [c_2, b_1, b_2, \dots, b_n]$, perkalian M dan N adalah $MN = [c_1c_2, a_1 + b_1, a_2 + b_2, \dots, a_n + b_n]$. Dua monomial yang hanya berbeda koefisiennya dapat digabungkan menjadi satu dengan menambahkan koefisiennya.

Jika kita mendefinisikan orde total $>$ pada variabel yang muncul dalam sistem polinomial, orde ini dapat diperluas dengan beberapa cara ke orde total atas monomial. Tanpa mengurangi keumuman, mari kita asumsikan $x_1 > x_2 > \cdots > x_n$. Misalkan $M = [c_1, a_1, a_2, \dots, a_n]$ dan $N = [c_2, b_1, b_2, \dots, b_n]$, di mana $c_i \neq 0$. Definisikan $M > N$ dengan urutan leksikografis pada daftar bilangan asli: $M > N$ jika dan hanya jika $[a_1, a_2, \dots, a_n] >^{lex} [b_1, b_2, \dots, b_n]$. Mudah untuk menunjukkan bahwa $>$ adalah urutan yang baik atas monomial (Proposisi 6.3.5). Misalkan $x > y > z$ dalam contoh di atas, maka monomial dalam setiap polinomial diurutkan oleh $>$ dari kiri ke kanan.

Dengan adanya pengurutan total atas monomial, persamaan polinomial dapat dilihat sebagai aturan penulisan ulang yang sisi kirinya adalah monomial maksimal dan sisi kanannya berisi negasi dari monomial lainnya. Kita akan menampilkan aturan penulisan ulang seperti sebelumnya dan dua aturan penulisan ulang dalam contoh kita terlihat seperti berikut:

$$\begin{aligned}(1) \quad x^3 &\rightarrow -4x^2 + 4xz + 8 \\(2) \quad x^2y &\rightarrow -4xy + 4yz - 4\end{aligned}$$

Di sini, variabel diperlakukan sebagai konstanta logika orde pertama dan koefisien sisi kiri dari setiap aturan penulisan ulang selalu disederhanakan menjadi satu. Kita dapat menggunakan aturan penulisan ulang ini untuk menyederhanakan polinomial dan menghitung pasangan kritis dengan superposisi.

Diberikan dua monomial $M = [1, a_1, a_2, \dots, a_n]$ dan $N = [1, b_1, b_2, \dots, b_n]$, misalkan kelipatan persekutuan terkecil (KPK) dari M dan N adalah

$$lcm(M, N) = [1, \max(a_1, b_1), \max(a_2, b_2), \dots, \max(a_n, b_n)].$$

Untuk menghitung pasangan kritis, kita mempertimbangkan dua aturan penulisan ulang yang sisi kirinya memiliki variabel yang sama dan menulis ulang KPK dari sisi kiri dengan kedua aturan tersebut. Untuk contoh kita, KPK dari (1) dan (2) adalah $KPK(x_3, x_2y) = x_3y$:

$$x^3y \Rightarrow \begin{cases} (1) \quad (-4x^2 + 4xz + 8)y = -4x^2y + 4xyz + 8y \\ (2) \quad x(-4xy + 4yz - 4) = -4x^2y + 4xyz - 4x \end{cases}$$

Dengan demikian, pasangan kritis dari (1) dan (2) adalah persamaan:

$$-4x^2y + 4xyz + 8y = -4x^2y + 4xyz - 4x$$

Setelah disederhanakan, kita peroleh $x + 2y = 0$, yang menghasilkan aturan baru:

$$(3) \quad x \rightarrow -2y$$

Secara formal, misalkan $M \rightarrow P$ dan $N \rightarrow Q$ menyatakan dua aturan penulisan ulang polinomial, M dan N berbagi variabel, dan $lcm(M, N) = MM' = N'N$ untuk beberapa monomial M' dan N' , pasangan kritis dari kedua aturan ini adalah persamaan $PM' = N'Q$. Seperti sebelumnya, proses ini disebut superposisi. Untuk contoh di atas, $M = x_3, N = x_2y, M' = y$, dan $N' = x$.

Sekarang, kita menggunakan (3) untuk menyederhanakan (1) dan memperoleh aturan baru:

$$(4) \quad y^3 \rightarrow 2y^2 + yz - 1$$

Dengan menggunakan (3) dan (4), (2) disederhanakan menjadi benar, dan kita memperoleh himpunan aturan penulisan ulang berikut:

$$\{ (3) x \rightarrow -2y, (4) y^3 \rightarrow 2y^2 + yz - 1 \}$$

Ini adalah sistem penulisan ulang kanonik untuk sistem polinomial asli.

Deskripsi di atas mengilustrasikan perluasan penyelesaian Knuth-Bendix (Prosedur 7.2.15) untuk persamaan polinomial. Secara teori, karena variabel polinomial dianggap sebagai konstanta oleh prosedur penyelesaian, maka akan berakhir dengan sistem penulisan ulang kanonik seperti pada kasus persamaan dasar. Dalam praktiknya, prosedur penyelesaian mungkin kehabisan memori sebelum berakhir.

Dalam literatur, sistem penulisan ulang kanonik untuk sistem polinomial disebut basis Gröbner dan prosedur untuk memperolehnya disebut algoritma Buchberger, yang diperkenalkan oleh Bruno Buchberger pada tahun 1965. Algoritma Buchberger dapat dilihat sebagai generalisasi dari eliminasi Gaussian untuk pemrograman linier dan algoritma Euclid untuk menghitung pembagi persekutuan terbesar polinomial. Penulisan ulang serta superposisi adalah generalisasi dari reduksi baris yang terjadi dalam eliminasi Gaussian dan pembagian Euclidean dari polinomial univariat. Selama bertahun-tahun, basis Gröbner telah berhasil digunakan di banyak bidang ilmu pengetahuan dan teknik, termasuk geometri aljabar, teori pengkodean, kriptografi, teori invarian, pemrograman bilangan bulat, teori graf, dan aljabar komputer.

Metode Wenjun Wu untuk sistem polinomial didasarkan pada konsep himpunan karakteristik, yang diperkenalkan pada akhir tahun 1940-an oleh Joseph Ritt. Sepenuhnya independen dari karya Buchberger, metode Wu menggunakan pembagian untuk mengubah sistem polinomial menjadi bentuk segitiga, yang disebut himpunan karakteristik Ritt-Wu. Metode Wu mudah digambarkan sebagai perluasan dari penyelesaian Knuth-Bendix seperti yang diilustrasikan di atas, karena pembagian dapat diimplementasikan dengan penulisan ulang dan superposisi. Sistem penulisan ulang kanonik dari persamaan polinomial dapat berfungsi sebagai himpunan karakteristik Ritt-Wu untuk banyak masalah dalam geometri. Selain pembuktian otomatis dalam geometri, metode Wu telah berhasil diterapkan di berbagai bidang ilmiah, seperti biologi, visi komputer, dan kinematika robot.

7.3 PEMBUKTIAN TEOREMA INDUKTIF

Pada Contoh 7.2.3, kita mengubah himpunan persamaan E menjadi aturan penulisan ulang, di mana add didefinisikan menggunakan dua persamaan: $E = \{add(0, y) \doteq y, add(s(x), y) \doteq s(add(x, y))\}$. Mudah dibuktikan dengan induksi pada i bahwa $add(s^i(0), s^j(0)) = s^{i+j}(0) = add(s^j(0), s^i(0))$. Karena $add(s^i(0), s^j(0)) = add(s^j(0), s^i(0))$, wajar untuk menyatakan bahwa $add(x, y) \doteq add(y, x)$ berlaku untuk semua istilah dasar x dan y . Jelas, $add(x, y) \doteq add(y, x)$ bukanlah konsekuensi logis dari E dan E_{ax} (aksioma kesamaan), karena $E \cup E_{ax}$ mungkin memiliki model di mana $add(x, y) \doteq$

$add(y, x)$ tidak berlaku. Diketahui bahwa beberapa sifat dari rumus orde pertama tidak dapat dibuktikan dengan deduksi atau kontradiksi; sifat-sifat tersebut hanya dapat dibuktikan dengan induksi. Itulah sebabnya kita menyebut sifat-sifat ini sebagai teorema induktif, dan salah satunya adalah $add(x, y) \doteq add(y, x)$. Induksi matematika adalah aturan inferensi yang mendasar dan ampuh. Pembuktian teorema induktif memiliki aplikasi penting dalam verifikasi formal dan telah digunakan untuk membuktikan kebenaran arsitektur atau algoritma komputer. Seperti bab-bab sebelumnya, kita sangat tertarik pada metode pembuktian induktif yang dapat diotomatisasi dalam komputer.

Teorema Induktif

Definisi 7.3.1 Diberikan bahasa orde pertama $L = (P, F, X, Op)$, suatu rumus bebas kuantifikasi B dikatakan sebagai teorema induktif dari A , jika untuk semua substitusi $\sigma: X \rightarrow T(F), A \models B\sigma$.

Dengan kata lain, suatu rumus merupakan teorema induktif dari A jika semua contoh dasarnya merupakan konsekuensi logis dari A . Kita dapat menunjukkan bahwa $add(x, y) = add(y, x)$ dan $add(add(x, y), z) = add(x, add(y, z))$ adalah teorema induktif dari $E = \{add(0 \dots$. Biasanya, A bebas kuantifikasi, dengan pemahaman bahwa semua variabel bebas bertindak sebagai variabel yang dikuantifikasi secara universal.

Untuk membuktikan teorema induktif, kita membutuhkan aturan inferensi untuk induksi, yang memiliki banyak versi berbeda. Salah satu aturan induksi mengharuskan simbol fungsi dibagi menjadi dua bagian: konstruktor dan fungsi yang didefinisikan.

Definisi 7.3.2 Diberikan himpunan E persamaan atas himpunan F simbol fungsi, himpunan bagian C dari F disebut konstruktor jika C mengandung setidaknya satu konstanta dan $D = F - C$ disebut fungsi yang didefinisikan.

Suatu fungsi $f \in D$ dengan aritas k dikatakan C -lengkap jika untuk $t_1, \dots, t_k \in T(C)$, terdapat suatu suku $t \in T(C), f(t_1, \dots, t_k) =_E t$. D dikatakan C -lengkap jika setiap $f \in D$ adalah C -lengkap. E dikatakan C -konsisten jika $t_1 \not\equiv_E t_2$ untuk $t_1, t_2 \in T(C)$ dan $t_1 \neq t_2$.

Jika $f \in D$ adalah C -lengkap, artinya f didefinisikan secara lengkap untuk suku-suku dalam $T(C)$. E akan disebut definisi persamaan D atas C . Jika $T(C)$ mewakili suatu domain, maka f mewakili fungsi total atas domain tersebut. Dalam literatur, konsep " C -lengkap" juga disebut "cukup lengkap".

Contoh 7.3.3 Dalam Contoh 7.2.3, kita mengubah himpunan persamaan E menjadi sistem penulisan ulang sehingga perhitungan dapat dilakukan dengan menulis ulang:

- (1) $pre(0) \rightarrow 0$
- (2) $pre(s(x)) \rightarrow x$
- (3) $add(0, y) \rightarrow y$
- (4) $add(s(x), y) \rightarrow s(add(x, y))$

- (5) $sub(x, 0) \rightarrow x$
- (6) $sub(x, s(y)) \rightarrow sub(pre(x), y)$
- (7) $mul(0, y) \rightarrow 0$
- (8) $mul(s(x), y) \rightarrow add(mul(x, y), y)$
- (9) $exp(x, 0) \rightarrow s(0)$
- (10) $exp(x, s(y)) \rightarrow mul(x, exp(x, y))$

Dalam contoh ini, simbol fungsi adalah $F = \{0, s, pre, add, sub, mul\}$, yang dapat dibagi menjadi dua bagian: $C = \{0, s\}$ dan $D = \{pre, add, sub, mul\}$. $T(C) = \{0, s(0), s^2(0), \dots\}$ adalah representasi dari bilangan asli. D mewakili himpunan fungsi atas bilangan asli.

Untuk setiap fungsi f dalam D , f adalah C -lengkap:

- $pre(0) = 0$ dan $pre(s^i(0)) = s^{i-1}(0)$ untuk $i > 0$
- $add(s^i(0), s^j(0)) = s^{i+j}(0)$ untuk setiap $i, j \geq 0$
- $sub(s^i(0), s^j(0)) = 0$ jika $i \leq j$ dan $pre(s^i(0), s^j(0)) = s^{i+j}(0)$ untuk $i > j$
- $mul(s^i(0), s^j(0)) = s^{i*j}(0)$ untuk setiap $i, j \geq 0$

Mudah untuk memeriksa bahwa $T(F)/=E = \{[0], [s(0)], \dots, [s^i(0)], \dots\}$, yang merupakan domain dari model Herbrand dengan kesamaan dan memiliki bijeksi h dari $T(C)$ ke $T(F)/=E$ dengan $h(t) = [t]$.

Jika $C = \{0, 1\}$, maka konsep konsistensi C bertepatan dengan konsistensi logika Boolean. Konsistensi C dari E tidak memungkinkan persamaan atas C . Untuk beberapa aplikasi, kita mungkin memerlukan persamaan semacam itu. Misalnya, untuk memodelkan himpunan bilangan bulat, kita dapat menggunakan tiga konstruktor: $0, s$ (ditambah satu), dan p (dikurangi satu), sehingga bilangan bulat negatif $-i$ direpresentasikan oleh $p^i(0)$. Dalam hal ini, kita memerlukan $E_C = \{p(s(x)) \doteq x, s(p(x)) = x\}$, dan definisi konsistensi C dapat dimodifikasi dengan mengganti $T(C)$ dengan $T(C)/=_{E_C}$.

Kami meninggalkan pembuktian lemma berikut sebagai latihan.

Lemma 7.3.4 Misalkan E adalah definisi persamaan dari $D = F - C$. D adalah C -lengkap jika dan hanya jika untuk setiap istilah dasar $s \in T(F)$, terdapat $t \in T(C)$ sedemikian sehingga $s =_E t$.

Proposisi 7.3.5 Misalkan E adalah himpunan persamaan atas F , $C \subset F$ adalah himpunan konstruktor dan $D = F - C$. Misalkan $h : T(C) \rightarrow T(F)/=E$ didefinisikan oleh $h(t) = [t]$. Pernyataan-pernyataan berikut ini benar:

- (a) h bersifat injektif jika dan hanya jika E adalah C -konsisten.
- (b) h bersifat surjektif jika dan hanya jika D adalah C -lengkap.
- (c) h bersifat bijektif jika dan hanya jika E adalah C -konsisten dan D adalah C -lengkap.

Bukti

- (a) $h: T(C) \rightarrow T(F)/=_E$ bersifat injektif jika dan hanya jika $h(t_1) \neq h(t_2)$ setiap kali $t_1 \neq t_2$ untuk $t_1, t_2 \in T(C)$. Dengan demikian, $h(t_1) = h(t_2)$ jika dan hanya jika $[t_1] = h(t_1) = h(t_2) = [t_2]$, jika dan hanya jika $t_1 =_E t_2$, jika dan hanya jika E tidak konsisten dengan C .
- (b) $h: T(C) \rightarrow T(F)/=_E$ bersifat surjektif jika dan hanya jika $\forall s \in T(F) \exists t \in T(C) h(t) = [t] = [s]$, jika dan hanya jika $\forall s \in T(F) \exists t \in T(C) s =_E t$, jika dan hanya jika D lengkap dengan C (Lemma 7.3.4).
- (c) Berasal dari (a) dan (b).

Tidak selalu mudah untuk menunjukkan bahwa setiap fungsi yang didefinisikan adalah C -lengkap atau E adalah C -konsisten. Proposisi di atas tidak memberikan metode yang efektif untuk tugas ini tetapi dapat membantu kita memahami konsep-konsep ini.

Untuk membuktikan teorema induktif dengan adanya konstruktor dan fungsi C -lengkap, kita hanya perlu mempertimbangkan istilah dasar dalam $T(C)$, bukan $T(F)$, sehingga membuat bukti induktif lebih sederhana. Secara umum, teorema induktif adalah sifat-sifat fungsi yang didefinisikan.

Induksi Struktural

Salah satu aturan induksi disebut aturan induksi struktural, yang didasarkan pada struktur istilah yang dibangun oleh konstruktor. Dalam contoh saat ini, konstruktornya adalah $C = \{0, s\}$.

Aturan Induksi Struktural untuk $C = \{0, s\}$

$$\frac{B(0) \quad B(y) \rightarrow B(s(y))}{B(x)}$$

di mana $B(x)$ merupakan teorema induktif yang akan dibuktikan dengan induksi pada x .

Aturan di atas menyatakan bahwa untuk membuktikan teorema induktif $B(x)$, kita perlu membuktikan dua rumus: $B(0)$ dan $B(y) \rightarrow B(s(y))$. Kebenarannya didasarkan pada kondisi bahwa semua fungsi dalam $F - C$ adalah C -lengkap. Kelengkapan C memastikan bahwa setiap suku dalam $T(F)$ setara dengan suku dalam $T(C)$ (Lemma 7.3.4). Karena setiap suku $x \in T(C)$ dapat diwakili oleh 0 atau $s(y)$, jika $B(0)$ dan $B(s(y))$ benar, maka $B(x)$ juga benar. $B(0)$ adalah kasus dasar induksi. Untuk membuktikan $B(s(y))$, kita menggunakan $B(y)$ sebagai "hipotesis induksi" karena y lebih kecil dari $s(y)$ dalam hal struktur suku $T(C)$.

Contoh 7.3.6 Misalkan $B(x)$ adalah $\text{add}(x, 0) \doteq x$. Dengan menerapkan induksi struktural pada persamaan ini, kita perlu membuktikan (a) $\text{add}(0, 0) \doteq 0$ dan (b) $\text{add}(x, 0) \doteq x \rightarrow \text{add}(s(x), 0 \doteq s(x))$. (a) mudah dibuktikan; (b) disederhanakan dengan aturan (4), definisi add , menjadi: $\text{add}(x, 0) \doteq x \rightarrow s(\text{add}(x, 0)) \doteq s(x)$, yang benar dengan persilangan kesamaan, seperti yang dijelaskan di bawah ini.

Persilangan Kesamaan

$$\frac{(t_1 \neq t_2 \mid A(t_1))}{(t_1 \neq t_2 \mid A(t_2))}$$

Karena $(t_1 \neq t_2 \mid A(t_1))$ setara dengan $(t_1 \doteq t_2) \rightarrow A(t_1)$, maka hasil $(t_1 \neq t_2 \mid A(t_2))$ dapat dianggap sebagai “menulis ulang $A(t_1)$ menjadi $A(t_2)$ dengan konteks $t_1 \doteq t_2$ ”. Misalnya, $add(x, 0) \doteq x \rightarrow s(add(x, 0)) \doteq s(x)$ ditulis menjadi $add(x, 0) \doteq x \rightarrow s(x) \doteq s(x)$ dengan persilangan kesamaan, di mana $t_1 = add(x, 0)$ dan $t_2 = x$. Dengan kata lain, kita menggunakan hipotesis induksi $add(x, 0) \doteq x$ untuk menunjukkan bahwa $s(add(x, 0) \doteq s(x))$ adalah benar.

Penyeberangan kesamaan, yang disebut fertilisasi silang oleh Boyer dan Moore, adalah aturan penyederhanaan dengan adanya aksioma kesamaan, karena hasil berikut.

Proposisi 7.3.7 (Keabsahan Penyeberangan Kesamaan) $E_{ax} \models (t_1 \neq t_2 \mid A(t_1)) \leftrightarrow (t_1 \neq t_2 \mid A(t_2))$, di mana E_{ax} adalah aksioma kesamaan.

Bukti Dari aksioma kesamaan monoton, yaitu, $(x_i = y_i) \wedge p(\dots, x_i, \dots) \rightarrow p(\dots, y_i, \dots)$, kita dapat memperoleh $1.2.1.2$, yang bentuk klausanya adalah

$$(t_1 \neq t_2 \mid \overline{A(t_1)} \mid A(t_2))$$

Resolven dari klausa ini dan $(t_1 \neq t_2 \mid A(t_1))$ pada $A(t_1)$ adalah $(t_1 \neq t_2 \mid A(t_2))$. Dengan menukar t_1 dan t_2 , implikasi lainnya juga berlaku. Perbedaan mendasar antara menggunakan aturan inferensi seperti resolusi dan menggunakan aturan induksi bukanlah bahwa urutan yang berdasar, misalnya, $s(x) > x$, diperlukan saat menerapkan induksi. Saat menggunakan resolusi untuk pembuktian teorema dengan penyangkalan, klausa kosong adalah tujuannya, dan aturan resolusi digunakan untuk menurunkan tujuan ini dari klausa masukan. Saat menggunakan aturan induksi untuk pembuktian teorema, tujuannya adalah rumus yang akan dibuktikan. Namun, kita tidak pernah mencoba (dan hampir tidak mungkin) untuk menurunkan tujuan ini secara langsung dari rumus masukan dengan aturan induksi. Sebaliknya, kita mencoba untuk mereduksi tujuan menjadi sub-tujuan dengan menggunakan aturan induksi secara terbalik. Artinya, untuk membuktikan $B(x)$ dengan induksi, kita menghasilkan dua sub-tujuan $B(0)$ dan $B(y) \rightarrow B(s(y))$ dari $B(x)$ dan mencoba untuk menetapkan validitas sub-tujuan ini. Ingat bahwa ketika setiap fungsi yang didefinisikan adalah C -lengkap, kondisi ini memastikan kebenaran aturan induksi.

Contoh 7.3.8 Melanjutkan dari Contoh 7.3.3, mari kita buktikan bahwa $sub(add(x, y), y) = x$. Jika kita melakukan induksi pada x , pembuktian tidak akan berhasil. Di sisi lain, kita dapat melakukan induksi pada y . Dengan

memeriksa definisi *add* dan *sub*, kita melihat bahwa x adalah posisi induksi untuk $add(x, y)$ dan y adalah posisi induksi untuk $sub(x, y)$.

Kasus dasarnya adalah $sub(add(x, 0), 0) \doteq x$, yang disederhanakan menjadi benar dengan aturan (3) dan (5). Kasus induktifnya adalah

$$sub(add(x, y), y) \doteq x \rightarrow sub(add(x, s(y)), s(y)) \doteq x$$

Untuk menyederhanakan $add(x, s(y))$, kita membutuhkan $add(x, s(y)) \doteq s(add(x, y))$, yang juga merupakan teorema induktif, dan pembuktiannya diserahkan sebagai latihan. Misalkan teorema ini tersedia bagi kita, maka

$$\begin{aligned} & sub(add(x, s(y)), s(y)) \\ = & sub(s(add(x, y)), s(y)) && \text{by } add(x, s(y)) \doteq s(add(x, y)), \\ = & sub(pre(s(add(x, y))), y) && \text{by rule (6),} \\ = & sub(add(x, y), y) && \text{by rule (2),} \\ = & x && \text{by equality crossing} \end{aligned}$$

Persilangan kesamaan dapat digunakan sebagai aturan penyederhanaan dalam pembuktian resolusi dengan kesamaan.

Induksi pada Dua Variabel

Contoh sebelumnya menunjukkan bahwa kita perlu memilih variabel yang tepat untuk melakukan induksi. Terkadang, kita perlu melakukan induksi pada beberapa variabel.

Contoh 7.3.9 Menambahkan dua predikat berikut ke Contoh 7.3.3:

- (11) $y < 0 \rightarrow \perp$
- (12) $0 < s(x) \rightarrow \top$
- (13) $s(y) < s(x) \rightarrow y < x$
- (14) $0 \leq y \rightarrow \top$
- (15) $s(x) \leq 0 \rightarrow \perp$
- (16) $s(x) \leq s(y) \rightarrow x \leq y$

Untuk membuktikan $(x \leq y) \doteq \neg(y < x)$ dengan induksi struktural, baik hanya pada x saja maupun hanya pada y saja tidak akan berhasil. Kita membutuhkan induksi pada dua variabel.

Aturan Induksi pada Dua Variabel untuk $C = \{0, s\}$

$$\frac{B(0, y) \quad B(s(x), 0) \quad B(x, y) \rightarrow B(s(x), s(y))}{B(x, y)}$$

di mana $B(x, y)$ merupakan teorema induktif yang akan diinduksikan pada x dan y .

Ini bukan satu-satunya aturan induksi untuk kedua variabel tersebut. Mengapa kita harus menggunakan aturan ini untuk $(x \leq y) \doteq \neg(y < x)$? Petunjuknya ada pada definisi $\leq$

dan $<$: untuk $(x \leq y)$, kita mendefinisikan $\leq$ pada $(0, y)$, $(s(x), 0)$, dan $(s(x), s(y))$ dengan panggilan rekursif pada (x, y) . Untuk $(y < x)$, kita juga mendefinisikan $<$ pada $(0, y)$, $(s(x), 0)$, dan $(s(x), s(y))$ dengan panggilan rekursif pada (x, y) (posisi x dan y ditukar di sini). Dengan kata lain, persamaan definisi dari kedua predikat ini memberi tahu kita kasus apa yang perlu dipertimbangkan: kasus dasar berasal dari persamaan definisi tanpa rekursi; Kasus-kasus induktif berasal dari persamaan definisi dengan rekursi. Ini adalah heuristik induksi utama dalam logika komputasi Boyer dan Moore, di mana fungsi lisp digunakan sebagai pengganti definisi persamaan. Aturan inferensi induksi selalu diperoleh secara otomatis dari definisi fungsi yang muncul dalam teorema yang akan dibuktikan.

Sekarang, kembali ke pembuktian perlunya membuktikan tiga sub-tujuan: $(x \leq y) \doteq \neg(y < x)$, dengan aturan induksi di atas, kita

1. $(0 \leq y) \doteq \neg(y < 0)$, yang dapat dengan mudah dibuktikan dengan menunjukkan $0 \leq y = T$ dengan aturan (14) dan $\neg(y < 0) = \neg(\perp) = T$ dengan aturan 11.
2. $(s(x) \leq 0) \doteq \neg(0 < s(x))$, yang dapat dibuktikan dengan menunjukkan $(s(x) \leq 0) = \perp$ dengan aturan 15 dan $\neg(0 < s(x)) = \neg(T) = \perp$ dengan aturan 12.
3. $(x \leq y) \doteq \neg(y < x) \rightarrow (s(x) \leq s(y)) \doteq \neg(s(y) < s(x))$. Bukti dari poin 3 juga mudah: $s(x) \leq s(y)$ disederhanakan menjadi $x \leq y$ dengan aturan 16 dan $\neg(s(y) < s(x))$ menjadi $\neg(y < x)$ dengan aturan 13, maka poin 3 menjadi implikasi yang sepele: $(x \leq y) \doteq \neg(y < x) \rightarrow (x \leq y) \doteq \neg(y < x)$.

Spesifikasi Aljabar Multi-Sort

Sejauh ini, kita telah berfokus pada fungsi bilangan asli berdasarkan konstruktor 0 dan s . Dalam banyak aplikasi, kita membutuhkan struktur data lain, seperti daftar, pohon, dll., yang memiliki konstruktornya sendiri. Pada Bagian 5.2, kita telah membahas secara singkat logika banyak-sort, yang merupakan perluasan kecil dari logika orde pertama. Suatu tipe atau sort sesuai dengan predikat unary (atau monadik) $p(x)$ dan mendefinisikan himpunan $S = \{x \mid p(x)\}$. Spesifikasi aljabar banyak-sort adalah bahasa di mana kita mendefinisikan fungsi dengan persamaan dalam logika orde pertama dengan banyak tipe. Spesifikasi aljabar banyak-sort akan menyediakan alat yang nyaman untuk aplikasi dengan banyak tipe. Misalnya, untuk bekerja pada daftar bilangan asli, kita membutuhkan dua tipe: bilangan asli dan daftar, dan domainnya dapat ditentukan oleh BNF sebagai berikut:

$$\begin{aligned}\langle nat \rangle &::= 0 \mid s(\langle nat \rangle) \\ \langle list \rangle &::= nil \mid cons(\langle nat \rangle, \langle list \rangle)\end{aligned}$$

di mana konstruktor untuk nat adalah $0: nat$ dan $s: nat \rightarrow nat$ dan konstruktor untuk $list$ adalah $nil: list$ dan $cons: nat, list \rightarrow list$. Kita telah melihat fungsi-fungsi pada nat ; fungsi-fungsi pada $list$ juga dapat didefinisikan menggunakan persamaan. Untuk membuktikan teorema induksi yang melibatkan nil dan $cons$, kita membutuhkan aturan berikut:

Aturan Induksi Struktural untuk $C = \{nil, cons\}$

$$\frac{B(nil) \quad B(y) \rightarrow B(cons(x, y))}{B(y)}$$

di mana $B(y)$ merupakan teorema induktif yang akan diinduksikan pada y .

Contoh 7.3.10 Operasi penambahan dan pembalikan pada daftar dapat didefinisikan sebagai berikut:

$$\begin{aligned} & app : list, list \rightarrow list \\ (1) \quad & app(nil, y) \rightarrow y \\ (2) \quad & app(cons(x, y), z) \rightarrow cons(x, app(y, z)) \\ & rev : list \rightarrow list \\ (3) \quad & rev(nil) \rightarrow nil \\ (4) \quad & rev(cons(x, y)) \rightarrow app(rev(y), cons(x, nil)) \end{aligned}$$

Untuk membuktikan $rev(app(y, z)) \doteq app(rev(z), rev(y))$ untuk semua daftar y dan z , aturan induksi struktural mereduksi $rev(app(y, z)) \doteq app(rev(z), rev(y))$ menjadi dua sub-tujuan:

$$\begin{aligned} G_1: \quad & rev(app(nil, z)) \doteq app(rev(z), rev(nil)) \\ G_2: \quad & rev(app(y, z)) \doteq app(rev(z), rev(y)) \rightarrow rev(app(cons(x, y), z)) \\ & \doteq app(rev(z), rev(cons(x, y))) \end{aligned}$$

Kita dapat menyederhanakan G_1 dengan aturan 1 dan 3 menjadi $rev(z) = app(rev(z), nil)$, yang menunjukkan sifat yang lebih umum: $app(y, nil) \doteq y$. Heuristik ini disebut generalisasi dan dapat diimplementasikan secara otomatis. Memang, $app(y, nil) \doteq y$ adalah teorema induktif lain, dan pembuktiannya diserahkan sebagai latihan. Dengan menggunakan lemma ini, $rev(z) \doteq app(rev(z), nil)$ menjadi benar.

Untuk G_2 , kita dapat menyederhanakannya

$$\begin{aligned} & rev(app(cons(x, y), z)) \doteq app(rev(z), rev(cons(x, y))) \\ & \text{by rule 2 to } rev(cons(x, app(y, z))) \doteq app(rev(z), rev(cons(x, y))), \text{ then to} \\ & app(rev(app(y, z)), cons(x, nil)) \doteq app(rev(z), app(rev(y), cons(x, nil))) \end{aligned}$$

berdasarkan aturan 4. Dengan menggunakan persilangan kesamaan, $rev(app(y, z))$ ditulis ulang menjadi $app(rev(z), rev(y))$, dan kita dihadapkan pada

$$app(app(rev(z), rev(y)), cons(x, nil)) \doteq app(rev(z), app(rev(y), cons(x, nil)))$$

Generalisasi dari rumus di atas adalah asosiatifitas aplikasi:

$$app(app(x', y'), z') = app(x', app(y', z'))$$

di mana x' berasal dari $rev(z)$, y' dari $rev(y)$, dan z' dari $cons(x, nil)$. Asosiativitas app dapat ditunjukkan sebagai teorema induktif secara terpisah. Setelah ini dilakukan, bukti G_2 selesai. Generalisasi adalah teknik heuristik yang awalnya digunakan dalam pembuktian teorema Boyer-Moore.

Contoh di atas mengilustrasikan fitur menarik dari pembuktian teorema induktif: untuk membuktikan $rev(app(y, z)) = app(rev(z), rev(y))$, kita perlu membuktikan lemma tambahan: $app(y, nil) \doteq y$ dan $app(app(x, y), z) \doteq app(x, app(y, z))$. Lemma-lemma ini terkadang dapat dihasilkan secara otomatis dan terkadang perlu diberikan oleh pengguna. Demi ketelitian, lemma-lemma ini perlu dibuktikan sebelum digunakan dalam pembuktian.

Contoh 7.3.11 Mari kita buktikan teorema induktif lain, $rev(rev(y)) = y$ dengan aturan induksi yang sama. Aturan induksi mereduksi $rev(rev(y)) = y$ menjadi dua sub-tujuan: $G_3 : rev(rev(nil)) = nil$, yang mudah dibuktikan dengan aturan 2, dan

$$G_4 : rev(rev(y)) \doteq y \rightarrow rev(rev(cons(x, y))) \doteq cons(x, y).$$

Dengan menerapkan aturan 4, kita menulis ulang $rev(rev(cons(x, y)))$ menjadi $rev(app(rev(y), cons(x, nil)))$, yang kemudian disederhanakan menjadi $app(rev(cons(x, nil)), rev(rev(y)))$. Karena $rev(cons(x, nil))$ ditulis ulang menjadi x oleh aturan 4, 3, dan 1, maka $app(rev(cons(x, nil)), rev(rev(y)))$ ditulis ulang menjadi $cons(x, rev(rev(y)))$ oleh aturan 2 dan 1. Dengan demikian, G_4 menjadi

$$G_5 : rev(rev(y)) \doteq y \rightarrow cons(x, rev(rev(y))) \doteq cons(x, y)$$

Dengan menerapkan persilangan kesamaan, $rev(rev(y))$ diganti dengan y dan G_5 menjadi benar. Kami telah menunjukkan cara menggunakan bahasa spesifikasi aljabar dari tipe data abstrak untuk menulis aksioma (definisi fungsi dan predikat), teorema atau lemma (inferensi perantara), dll. untuk penalaran induktif. Secara khusus, kami mensyaratkan bahwa aksioma serta teorema dalam spesifikasi aljabar harus dinyatakan sebagai himpunan persamaan (bersyarat). Dengan kata lain, kami tertarik pada metode otomatis yang dapat membuktikan persamaan dari persamaan dengan induksi matematika.

Contoh 7.3.12 Kita dapat mendefinisikan algoritma pengurutan penyisipan menggunakan persamaan:

$$\begin{aligned} & isort : list \rightarrow list \\ (5) \quad & isort(nil) \rightarrow nil \\ (6) \quad & isort(cons(x, y)) \rightarrow insert(x, isort(y)) \\ & insert : nat, list \rightarrow list \end{aligned}$$

- (7) $insert(x, nil) \rightarrow cons(x, nil)$
- (8) $insert(x, cons(y, z)) \rightarrow cons(x, cons(y, z))$ if $x \leq y$
- (9) $insert(x, cons(y, z)) \rightarrow cons(y, insert(x, z))$ if $\neg(x \leq y)$
- $sorted : list \rightarrow boolean$
- (10) $sorted(nil) \rightarrow \top$
- (11) $sorted(cons(x, nil)) \rightarrow \top$
- (12) $sorted(cons(x, cons(y, z))) \rightarrow (x \leq y) \wedge sorted(y, z)$

Perhatikan bahwa $\leq$ yang digunakan di sini didefinisikan dalam Contoh 7.3.9.

Untuk kebenaran $isort(x)$, kita dapat membuktikan bahwa $sorted(isort(x))$ adalah teorema induktif (soal latihan). Jika kita mendefinisikan $perm(x, y)$ sebagai benar jika dan hanya jika x adalah permutasi dari y , maka kita juga dapat membuktikan bahwa $perm(x, isort(x))$ adalah teorema induktif. Kedua teorema tersebut memastikan kebenaran algoritma pengurutan penyisipan.

Teorema induktif yang ditunjukkan dalam contoh di atas dapat dengan mudah dibuktikan oleh pembukti teorema induktif saat ini. Banyak algoritma dapat ditentukan oleh persamaan dengan cara ini dan dapat diverifikasi secara formal secara otomatis menggunakan pendekatan ini.

Keberhasilan yang patut diperhatikan dari pembuktian teorema induktif berasal dari bidang pemrograman fungsional (lihat Bagian 9.1). Pembuktian teorema Boyer-Moore NQTHM, penerusnya ACL2, Coq, dan Isabelle, adalah perangkat lunak terkenal yang menggunakan bahasa fungsional untuk menentukan sirkuit digital, sistem operasi, kompiler, algoritma, dll., dan membuktikan bahwa spesifikasi ini memiliki sifat-sifat yang diinginkan melalui pembuktian induksi otomatis dan teknik penalaran otomatis lainnya. Pemahaman tentang pembuktian induksi dalam konteks sistem penulisan ulang akan membantu kita memahami perangkat lunak ini karena mereka memiliki prinsip dan teknik yang sama.

Rekursi, Induksi, dan Referensi Diri

Rekursi, induksi, dan referensi diri terkadang diperlakukan sebagai sinonim. Misalnya, kalimat-kalimat berikut dapat digunakan untuk menjelaskan satu sama lain:

- Suatu fungsi didefinisikan secara rekursif.
- Suatu fungsi didefinisikan secara induktif.
- Suatu fungsi didefinisikan dengan referensi diri.

Induksi sebagai aturan inferensi tidak dapat digantikan oleh rekursi atau referensi diri. Keabsahan aturan induksi didasarkan pada urutan yang beralasan seperti yang telah kita lihat di bagian ini. Urutan ini tersirat ketika kita mendefinisikan suatu fungsi (atau sekumpulan objek) dengan rekursi. Tanpa urutan ini, kita mungkin memiliki nilai fungsi yang tidak terdefinisi atau inkonsistensi.

Contoh 7.3.13 Konjektur Collatz yang belum terpecahkan menanyakan apakah fungsi rekursif $f(x)$ berikut ini terdefinisi total, yaitu $f(x) = 1$, untuk setiap bilangan bulat positif x :

$$f(x) = \text{if } (x = 1) \text{ then } 1 \text{ else if } \text{even}(x) \text{ then } f(x/2) \text{ else } f((3x + 1)/2)$$

Jika kita dapat menemukan bilangan bulat ganjil x sedemikian sehingga $f(x) = f((3x + 1)/2) = \dots = f(x)$, maka x akan menjadi contoh tandingan terhadap konjektur Collatz dan $f(x)$ tidak terdefinisi. Prosedur untuk menghitung $f(x)$ akan berulang selamanya pada x ini.

Dalam kasus penggunaan rekursi yang salah, kita sering menggunakan referensi diri. Kesalahan penalaran melingkar adalah kesalahan khas dari referensi diri, di mana dua pernyataan (atau konsep) dari fakta yang sama digunakan untuk saling mendukung. Jika makna variabel proposisional didefinisikan oleh referensi diri, hal itu dapat menyebabkan kontradiksi. Misalnya, pernyataan “Saya berbohong” menyebabkan kontradiksi jika kita berasumsi bahwa “jika seseorang berbohong, maka apa yang dia katakan adalah salah.”

Jika pernyataan “Saya berbohong” benar, karena diucapkan oleh seseorang yang berbohong, maka pernyataan tersebut salah. Jika pernyataan itu salah, maka “Saya tidak berbohong” adalah benar, jadi apa yang saya katakan (yaitu, “Saya berbohong”) seharusnya benar. Kontradiksi di atas disebut “paradoks pembohong” dan juga dapat ditemukan dalam logika orde pertama. Misalnya, paradoks Epimenides, “Semua orang Kreta adalah pembohong” ketika diucapkan oleh orang Kreta Yunani kuno, adalah salah satu contoh pertama yang tercatat.

Dengan menginterpretasikan “berbohong” sebagai “salah,” paradoks pembohong dapat diungkapkan dalam logika sebagai rujukan diri: misalkan p menyatakan “ p adalah salah,” memberikan nilai kebenaran apa pun kepada p akan menghasilkan kontradiksi. Paradoks ini dapat diperluas ke beberapa kalimat:

- p : Pernyataan berikutnya, yaitu q , adalah salah.
- q : Pernyataan berikutnya, yaitu r , adalah salah.
- r : Pernyataan pertama, yaitu p , adalah salah.

Dari $p \equiv \neg q$, $q \equiv \neg r$, dan $r \equiv \neg p$, mudah untuk menyimpulkan $p \equiv \neg p$.

Arthur Prior mengklaim bahwa tidak ada yang paradoks tentang paradoks pembohong. Argumennya adalah bahwa setiap pernyataan mencakup pernyataan implisit tentang kebenarannya sendiri. Misalnya, pernyataan “Memang benar bahwa “ $2 + 2 = 4$ ” tidak mengandung informasi lebih dari pernyataan “ $2 + 2 = 4$,” karena frasa “memang benar bahwa...” selalu ada secara implisit. Artinya, frasa “memang benar bahwa...” setara dengan “seluruh pernyataan ini benar dan...”. Dengan demikian, dalam paradoks pembohong, dua pernyataan berikut ini adalah:

- Pernyataan ini salah.
- Pernyataan ini benar dan pernyataan ini salah.

Yang terakhir adalah kontradiksi sederhana dalam bentuk “ p dan bukan p ,” dan karenanya, tidak ada paradoks.

Paradoks Curry, yang dinamai menurut ahli logika Haskell Curry, adalah paradoks referensi diri lainnya dalam logika. Paradoks ini dapat dijelaskan sebagai berikut: misalkan p menyatakan “jika p maka q ,” atau setara dengan $\neg p \vee q$. Jika p benar, maka q benar dari $\neg p \vee q$. Karena q sembarang dan dapat berupa $\perp$, kita memiliki kontradiksi. Jika p salah, maka negasi dari $\neg p \vee q$ adalah benar. Karena $\neg(\neg p \vee q) \equiv p \wedge \neg q$ adalah salah ketika p salah, kita juga memiliki kontradiksi. Jika kita menggunakan argumen Prior dalam paradoks Curry, maka p akan

menunjukkan “ p dan jika p maka q .” Ekspresi ini setara dengan $p \wedge q$, dan tidak ada paradoks yang tampak. Paradoks Curry dapat diekspresikan dalam bahasa alami dan dalam berbagai logika, termasuk teori himpunan naif tertentu, kalkulus lambda, dan logika kombinatorial.

Referensi diri disalahkan sebagai sumber paradoks Russell dalam teori himpunan naif. Namun, referensi diri sangat diperlukan dalam matematika dan ilmu komputer. Seperti yang terlihat di Bab 1, rekursi adalah alat yang ampuh untuk membangun objek, fungsi, dan prosedur. Di Bab 11, kita akan melihat bahwa rekursi adalah landasan teori komputabilitas.

7.4 RESOLUSI DENGAN KESETARAAN

Sampai saat ini dalam bab ini, fokus kita adalah pada persamaan, yang merupakan klausa unit positif dengan “ $\doteq$ ” sebagai satu-satunya predikat. Persamaan cukup ekspresif untuk menggambarkan semua komputasi. Misalnya, semua mesin Turing dapat diekspresikan sebagai himpunan persamaan. Di sisi lain, lebih mudah untuk mengekspresikan aksioma dan teorema dalam klausa umum. Penyelesaian Knuth-Bendix menunjukkan bahwa kita tidak memerlukan aksioma kesamaan secara eksplisit jika kita memiliki inferensi seperti superposisi yang menghitung pasangan kritis dari dua aturan penulisan ulang. Hal ini juga berlaku dalam pembuktian berbasis resolusi seperti yang disarankan oleh Alan Robinson dan Larry Wos mengenai aturan paramodulasi, tak lama setelah penemuan aturan resolusi.

Paramodulasi

Definisi 7.4.1 (*Paramodulasi*) Misalkan klausa c_1 adalah $(s \doteq t \mid \alpha)$ dan c_2 adalah $(A \mid \beta)$, di mana A adalah literal dan p adalah posisi non-variabel dari A , α dan β adalah literal sisa dalam c_1 dan c_2 , masing-masing, aturan paramodulasi didefinisikan sebagai berikut:

$$\frac{(s \doteq t \mid \alpha) \quad (A \mid \beta)}{(A[p \sim t] \mid \alpha \mid \beta)\sigma}$$

di mana σ adalah mgu dari s dan A/p . Klausa $(A[p \sim t] \mid \alpha \mid \beta)\sigma$ yang dihasilkan oleh aturan paramodulasi disebut paramodulan dari paramodulasi; c_1 dan c_2 adalah induk dari paramodulan. Kita menggunakan $paramod(c_1, c_2)$ untuk menunjukkan himpunan semua klausa seperti $(A[p \sim t] \mid \alpha \mid \beta)\sigma$ dari c_1 dan c_2 .

Superposisi adalah kasus khusus dari paramodulasi di mana α dan β kosong dan A juga merupakan persamaan.

Contoh 7.4.2 Misalkan c_1 adalah $(f(g(y)) \doteq a \mid r(y))$ dan c_2 adalah $(p(g(f(x))) \mid q(x))$, maka $f(g(y))$ dan $p(g(f(x)))/1.1 = f(x)$ dapat disatukan dengan $\sigma = [x \sim g(y)]$. Paramodulannya adalah $(p(g(a)) \mid q(g(y)) \mid r(y))$.

Proposisi 7.4.3 (*Keabsahan Paramodulasi*) Untuk setiap klausa c_1 dan c_2 , $E_{ax} \cup \{c_1, c_2\} \models paramod(c_1, c_2)$, di mana E_{ax} adalah aksioma kesamaan.

Bukti Jika $paramod(c_1, c_2)$ kosong, maka $paramod(c_1, c_2) \equiv T$ berdasarkan asumsi kita dan proposisi tersebut terbukti benar secara trivial. Jika $paramod(c_1, c_2)$ tidak kosong, misalkan σ adalah mgu yang digunakan dalam paramodulasi, $c_1\sigma$ adalah $(s' \doteq t' \mid \alpha')$ dan $c_2\sigma$ adalah $(A'[s'] \mid \beta')$, maka $paramod(c_1, c_2)$ mengandung $c_3: (A'[t'] \mid \alpha' \mid \beta')$.

Dari aksioma monotonik di E_{ax} , kita dapat menyimpulkan $(s' \doteq t') \wedge A'(s') \rightarrow A'(t')$ yang bentuk klausanya adalah $c_4: (s' \neq t' \mid \overline{A'(s')} \mid A'(t'))$. Terapkan resolusi pada $(s' \doteq t')$ dari $c_1\sigma$ dan c_4 , kita mendapatkan $c_5: (\overline{A'(s')} \mid A'(t') \mid \alpha')$. Akhirnya, terapkan resolusi pada $A'(s')$ dari $c_2\sigma$ dan c_5 , kita mendapatkan $c_6: (A'(t') \mid \alpha' \mid \beta')$, yang sama dengan c_3 . Karena resolusi tersebut sah, c_3 merupakan konsekuensi logis dari c_1, c_2 , dan E_{ax} .

Bukti di atas menunjukkan bahwa satu paramodulasi setara dengan dua resolusi dengan satu contoh aksioma monotonik. Dengan menggunakan paramodulasi, kita tidak memerlukan sebagian besar aksioma kesamaan. Dapat ditunjukkan bahwa setiap konsekuensi dari aksioma kesamaan dapat disimpulkan dari paramodulasi, dengan satu-satunya pengecualian aksioma refleksivitas: $x \doteq x$. ($x \doteq x$) sangat diperlukan dalam menunjukkan bahwa $A = \{f(x) \neq f(y)\}$ tidak dapat dipenuhi. Kita dapat memperkenalkan aturan inferensi berikut yang membuat aksioma $(x \neq x)$ menjadi tidak diperlukan.

Definisi 7.4.4 (Refleksi) Misalkan klausa C adalah $(s \neq t \mid \alpha)$, refleksi adalah aturan inferensi yang didefinisikan oleh skema berikut:

$$\frac{(s \neq t \mid \alpha)}{(\alpha)\sigma}$$

di mana σ adalah mgu dari s dan t .

Refleksi bekerja dengan cara yang sama seperti pemfaktoran: pemfaktoran menghilangkan satu literal dengan menyatukan dua literal; refleksi menghilangkan $(s \neq t)$ dengan menyatukan s dan t .

Paramodulasi yang dikombinasikan dengan resolusi, pemfaktoran, dan refleksi bersifat lengkap secara refutasional untuk pembuktian teorema dengan kesamaan. Artinya, jika $A \cup E_{ax}$ tidak dapat dipenuhi, di mana E_{ax} adalah aksioma kesamaan, maka klausa kosong dapat diturunkan dari A dengan paramodulasi, resolusi, pemfaktoran, dan refleksi. Ia tetap lengkap dengan strategi resolusi lengkap apa pun, seperti resolusi positif atau resolusi terurut. Kita dapat menambahkan batasan pada paramodulasi seperti kita menambahkan batasan pada resolusi, tanpa mengorbankan kelengkapan refutasional.

Definisi 7.4.5 (Paramodulasi Terurut) Diberikan urutan penyederhanaan $>$, misalkan $s \doteq t$ maksimal dalam klausa $c_1 (s \doteq t \mid \alpha)$, $s \not\prec t$ dan literal A maksimal dalam $c_2 (A \mid \beta)$, p adalah posisi non-variabel dari A , α dan β adalah literal sisa dalam c_1 dan c_2 , masing-masing. Aturan paramodulasi terurut akan

menghasilkan paramodulan $(A[p \sim t] \mid \alpha \mid \beta)\sigma$, di mana σ adalah mgu dari s dan A/p .

Paramodulasi dalam Contoh 7.4.2 adalah paramodulasi terurut jika $f(g(y)) \doteq a$ maksimal dalam c_1 dan $p(g(f(x)))$ maksimal dalam c_2 .

Dengan menggunakan urutan penyederhanaan yang sama, paramodulasi terurut dapat dikombinasikan dengan resolusi terurut tanpa kehilangan kelengkapan refutasional. Ketika sekumpulan klausa hanyalah sekumpulan persamaan (dan dianggap sebagai sekumpulan aturan penulisan ulang), paramodulasi terurut direduksi menjadi superposisi. Karya Robinson dan Wos tentang paramodulasi mendahului, tetapi tampaknya tidak memengaruhi, karya Knuth dan Bendix, yang secara independen merumuskan konsep superposisi dan penulisan ulang pada tahun 1970. Sebagai perbandingan, superposisi adalah kasus khusus dari aturan inferensi paramodulasi Robinson dan Wos, dan penulisan ulang sama dengan aturan penyederhanaan demodulasi Robinson dan Wos.

Aturan Penyederhanaan

Untuk pembuktian teorema berdasarkan resolusi dan paramodulasi, sejumlah besar klausa baru dihasilkan, dan kita perlu mengendalikan redundansi dalam ruang pencarian yang besar. Strategi terbatas untuk menggunakan resolusi dan paramodulasi efektif untuk mengurangi klausa yang berlebihan, menghapus klausa yang tidak perlu dalam pencarian klausa kosong juga penting untuk mengendalikan ruang pencarian. Seperti yang ditunjukkan sebelumnya, kita dapat menerapkan strategi penghapusan berikut dalam pembuktian teorema berbasis resolusi:

- Penghapusan literal murni: Klausa yang berisi literal murni dibuang.
- Penghapusan tautologi: Klausa tautologi dibuang.
- Penghapusan subsumsi: Klausa yang disubsumsi dibuang.
- Penghapusan unit: Literal yang merupakan negasi dari contoh beberapa klausa unit dibuang dari klausa.

Strategi penyederhanaan memungkinkan kita untuk mempertahankan klausa dalam bentuk yang disederhanakan sehingga tidak diperlukan banyak salinan klausa yang setara. Misalnya, jika sebuah klausa berisi istilah s , katakanlah $(p[s] \mid q)$, dan kita memiliki aturan penulisan ulang $s \rightarrow t$ atau hanya $s \doteq t$, maka klausa ini dapat ditulis ulang menjadi $(p[t] \mid q)$ dengan $s \doteq t$. Penulisan ulang bukanlah satu-satunya cara untuk mendapatkan $(p[t] \mid q)$, karena paramodulasi dapat menghasilkannya dari $(p[s] \mid q)$ dan $s \doteq t$. Namun, tidak perlu menyimpan $(p[s] \mid q)$ dan $(p[t] \mid q)$, karena keduanya secara logis setara di bawah kesamaan. Dengan kata lain, ketika $(p[s] \mid q)$ ditulis ulang menjadi $(p[t] \mid q)$, kita menyimpan yang terakhir dan membuang yang pertama. Itulah mengapa kita menyebut strategi semacam ini sebagai strategi penyederhanaan.

Untuk bekerja dengan resolusi terurut dan paramodulasi, kita dapat memberlakukan pembatasan pada persilangan kesamaan sebagai aturan penyederhanaan:

Persilangan Kesamaan Terurut

$$\frac{(s \neq t) \mid (A(s))}{(s \neq t) \mid (A(t))}$$

jika $s > t$ untuk urutan penyederhanaan $>$ digunakan dalam paramodulasi terurut.

Kebenarannya mengikuti dari Proposisi 7.3.7. Bentuk yang lebih umum dari persilangan kesamaan adalah aturan penyederhanaan berikut:

Definisi 7.4.6 (Penulisan Ulang Kontekstual) Diberikan klausa $c_1: (s \doteq t \mid \alpha)$, di mana $s > t$, klausa $c_2: (A \mid \beta)$ dapat ditulis ulang oleh c_1 jika ada substitusi σ sedemikian sehingga $A/p = s\sigma$ untuk posisi p dari A dan $\alpha\sigma \subseteq \beta$. Hasil dari penulisan ulang ini adalah $c'_2: (A[p \sim t\sigma] \mid \beta)$.

Perhatikan bahwa jika α kosong, maka c_1 adalah klausa unit positif dan penulisan ulang kontekstual sama dengan penulisan ulang biasa. $\neg\alpha$ disebut kondisi penulisan ulang (karena c_1 setara dengan $\neg\alpha \rightarrow s \doteq t$) dan $\neg\beta$ disebut konteks penulisan ulang (yang menyiratkan bahwa kondisi $\neg\alpha$ benar).

Contoh 7.4.7 Misalkan c_1 adalah $(f(g(y)) \doteq a) \mid q(y)$ dan c_2 adalah $(p(f(g(b))) \mid q(b))$ maka c_1 dapat menulis ulang c_2 menjadi $(p(a) \mid q(b))$ karena $p(f(g(b)))/1 = f(g(b)) = f(g(y))\sigma$, di mana $\sigma = [y \sim b]$ dan $q(b) = q(y)\sigma$ muncul di c_2 .

Proposisi 7.4.8 (Keabsahan Penulisan Ulang Kontekstual) Jika c_1 menulis ulang c_2 menjadi c'_2 dengan penulisan ulang kontekstual, maka $E_{ax} \cup \{c_1\} \models c_2 \leftrightarrow c'_2$, di mana E_{ax} adalah aksioma kesamaan.

Bukti Misalkan c_1 adalah $(s \doteq t \mid \alpha)$ dan σ adalah substitusi yang digunakan dalam penulisan ulang kontekstual, misalkan $s' = s\sigma$ dan $t' = t\sigma$, maka c_2 adalah $(A[s'] \mid \beta)$, dan c' adalah $(A[t'] \mid \beta)$.

Dengan menggunakan paramodulasi, kita dapat memperoleh c' dari $c_1\sigma$ dan c_2 serta c_2 dari $c_1\sigma$ dan c_r (dengan menukar peran s dan t). Keabsahan paramodulasi memastikan kesetaraan c_2 dan c_r di bawah c_1 dan E_{ax} .

Ada dua kemungkinan perluasan untuk penulisan ulang kontekstual. Perluasan pertama adalah mengganti kondisi $\alpha\sigma \subseteq \beta$ dengan $Ax \cup \{\neg\beta\} \models \neg\alpha\sigma$, di mana Ax adalah himpunan aksioma yang digunakan dalam masalah saat ini. Perhatikan bahwa $\alpha\sigma \subseteq \beta$ menyiratkan $\neg\beta \models \neg\alpha\sigma$.

Ekstensi kedua dari penulisan ulang kontekstual adalah memperlakukan literal non-kesetaraan sebagai aturan penulisan ulang. Artinya, jika c_1 adalah $(A \mid \alpha)$, di mana A adalah

literal non-kesetaraan maksimal dalam c_1 , maka kita menganggap c_1 sebagai aturan penulisan ulang “ $A \text{ T jika } \neg\alpha$ ” jika A positif; atau “ $B = \perp$ jika $\neg\alpha$ ” jika A adalah $\neg B$. Kemudian, c_1 dapat menulis ulang atom dalam klausa lain menjadi T atau $\perp$. Jika literal dalam klausa lain ditulis ulang menjadi T, ini setara dengan subsumsi. Jika ditulis ulang menjadi *false*, klausa lain akan memiliki satu literal lebih sedikit. Secara khusus, jika α kosong, ini setara dengan penghapusan unit (Definisi 6.2.10).

Kebenaran penulisan ulang kontekstual tetap berlaku dengan kedua ekstensi tersebut; Oleh karena itu, penulisan ulang kontekstual dapat berfungsi sebagai aturan penyederhanaan yang baik yang mencakup penulisan ulang biasa, subsumsi, dan penghapusan unit.

Teorema 7.4.9 Paramodulasi terurut yang dikombinasikan dengan resolusi terurut, pemfaktoran, refleksi, persilangan kesamaan terurut, dan penulisan ulang kontekstual terurut adalah lengkap secara refutasional untuk logika orde pertama dengan kesamaan.

Bukti teorema ini mengikuti alur bukti yang sama dengan Teorema 3.3.13: jika klausa kosong dihasilkan, maka klausa masukan tidak dapat dipenuhi karena semua aturan inferensi dan penyederhanaan adalah sah. Jika klausa kosong tidak pernah dapat dihasilkan dan himpunan klausa jenuh oleh aturan inferensi, maka model Herbrand dapat dibangun menggunakan urutan penyederhanaan pada $T(F)$.

Kesamaan dalam Prover9

Semua aturan inferensi dan penyederhanaan yang dibahas dalam bagian ini telah diimplementasikan dalam Prover9, kecuali penulisan ulang kontekstual. Penulisan ulang menggunakan persamaan, yang disebut *demodulasi*, didukung di Prover9. Bahkan, sebagian besar contoh yang diberikan dalam distribusi Prover9 mengandung kesamaan.

Contoh 7.4.10 Dalam Contoh 7.1.6, kita diberikan tiga persamaan yang menentukan grup bebas $(S,*)$. Dapatkah kita mengurangi jumlah persamaan menjadi satu? Berapa ukuran minimal dari persamaan tunggal tersebut? Ini adalah contoh pertanyaan yang menarik bagi para matematikawan. Berikut adalah jawaban untuk pertanyaan pertama:

$$y * i \left(z * \left(\left((u * i(u)) * i(x * z) \right) * y \right) \right) \doteq x$$

yang melibatkan empat variabel.

File input Prover9 untuk masalah ini hanyalah satu aksioma dan tujuannya adalah tiga persamaan dalam Contoh 7.1.6, dengan satu-satunya pengecualian bahwa kita menggunakan $y * i(y)$ untuk identitas e .

formulas(sos).

$y * i(z * (((u * i(u)) * i(x * z)) * y)) = x \# \text{label(oneAxiom)}.$


```

end_of_list.

formulas(goals).
(x * y) * z = x * (y * z)      # label(associativity).
x * i(x) = y * i(y)           # label(inverse).
x * (y * i(y)) = x             # label(identity).
end_of_list.

```

Prover9 membutuhkan waktu kurang dari satu detik untuk menemukan bukti untuk ketiga persamaan tersebut setelah menghasilkan 1587 klausa. Pembuktian hukum asosiatif adalah yang tersulit, yang terdiri dari 48 langkah setelah pembuktian persamaan lainnya.

Simbol kesamaan “=” sudah terintegrasi; begitu pula “! =”, yang merupakan negasi dari kesamaan. Opsi utama yang melibatkan paramodulasi adalah sebagai berikut:

```

clear(ordered_para). % default set
set(ordered_par).

assign(para_lit_limit, n). % default n=-1, range [-1 .. INT_MAX]

```

Jika $n \neq -1$, setiap induk dalam paramodulasi dapat memiliki paling banyak n literal. Opsi ini dapat menyebabkan ketidaklengkapan sistem inferensi.

```

set(para_units_only).
clear(para_units_only). % default clear

```

Bendera ini menyatakan bahwa kedua induk untuk paramodulasi harus berupa klausa unit. Satu-satunya efek dari bendera ini sama dengan menetapkan 1 ke parameter `para_lit_limit`. Opsi utama yang melibatkan penulisan ulang adalah sebagai berikut:

```

assign(demod_step_limit, n). % default n=1000, range [-1 .. INT_MAX]

```

Parameter ini membatasi jumlah langkah penulisan ulang yang diterapkan pada klausa selama demodulasi. Jika $n = -1$, tidak ada batasan.

```

assign(demod_size_limit, n). % default n=1000, range [-1 .. INT_MAX]

```

Parameter ini membatasi ukuran (diukur sebagai jumlah simbol) dari istilah saat didemodulasi. Jika ada istilah yang didemodulasi memiliki lebih dari n simbol, demodulasi klausa berhenti. Jika $n = -1$, tidak ada batasan.

```

set(back_demod).
clear(back_demod). % default clear

```

Jika flag ini diatur, aturan penulisan ulang baru akan mencoba menulis ulang persamaan dalam daftar usable dan sos. Prover9 juga menyediakan opsi bagi persamaan non-orientable untuk ditulis ulang sesuai aturan dalam kondisi tertentu, dan penulisan ulang menggunakan aturan ini dibatasi untuk memastikan penghentian.

7.5 PENCARIAN MODEL TERBATAS DALAM LOGIKA ORDE PERTAMA

Secara teori, masalah pencarian model dengan ukuran tetap dalam logika orde pertama dapat dirumuskan sebagai masalah SAT. Dalam praktiknya, konversi ke dalam instance SAT tidak selalu merupakan pendekatan yang paling efektif meskipun kita memiliki pemecah SAT yang canggih. Ada banyak pencari model terbatas yang diusulkan dalam literatur. Berikut ini, kita akan memperkenalkan Mace4 karena tersedia dan memiliki bahasa spesifikasi yang sama dengan Prover9.

Mace4: Alat Pencarian Model Terbatas

Mace4 adalah alat perangkat lunak untuk pencarian model terbatas dalam bahasa orde pertama. Mace4 didistribusikan bersama dengan Prover9 dan dibuat oleh William McCune. Prover9 dan Mace4 berbagi bahasa orde pertama dan ada antarmuka GUI untuk keduanya. Mace4 mencari model terbatas yang memenuhi rumus dalam bahasa orde pertama. Untuk ukuran domain tertentu, semua instance rumus di atas domain dibangun. Hasilnya adalah himpunan klausa dasar dengan kesamaan.

Kemudian, prosedur pengambilan keputusan berdasarkan penulisan ulang persamaan dasar diterapkan. Jika kepuasan terdeteksi, satu atau lebih model dicetak. Jika rumusnya adalah penolakan terhadap beberapa dugaan, setiap model yang ditemukan oleh Mace4 adalah contoh tandingan terhadap dugaan tersebut. Mace4 merupakan pelengkap yang berharga untuk Prover9, mencari contoh tandingan sebelum (atau bersamaan dengan) Prover9 digunakan untuk mencari bukti. Ini juga dapat digunakan untuk membantu men-debug klausa input dan rumus untuk Prover9.

Sebagian besar, Mace4 menerima file input yang sama dengan Prover9. Jika file input berisi perintah yang tidak dipahami oleh Mace4, maka argumen "-c" harus diberikan untuk memberi tahu Mace4 agar mengabaikan perintah tersebut. Misalnya, kita dapat menjalankan dua pekerjaan berikut secara paralel pada mesin Linux, dengan Prover9 mencari bukti dan Mace4 mencari contoh tandingan.

```
prover9 -f x2.in > x2.prover9.out
mace4 -c -f x2.in > x2.mace4.out
```

Sebagian besar opsi yang diterima oleh Mace4 dapat diberikan baik melalui baris perintah maupun dalam file input. Perintah berikut mencantumkan opsi baris perintah yang diterima oleh Mace4.

```
mace4 -help
```

Mace4 hanya mencari model hingga tak terurut. Artinya, sebuah model memiliki satu himpunan hingga yang mendasarinya, yang disebut domain (atau semesta), dan anggotanya selalu $0, 1, \dots, n - 1$ untuk himpunan berukuran n . Model-model tersebut adalah struktur yang mendefinisikan fungsi dan relasi di atas domain, sebagai interpretasi terhadap simbol fungsi dan predikat dalam rumus (menggunakan simbol yang sama). Secara default, Mace4 mulai mencari struktur dengan ukuran domain 2, dan kemudian, ia menambah ukurannya hingga berhasil atau mencapai batas tertentu. Ukuran domain awal atau ukuran tambahan dapat ditentukan oleh pengguna.

Jika sebuah rumus berisi konstanta yang merupakan bilangan asli, $\{0, 1, \dots\}$, Mace4 menganggapnya sebagai anggota domain dari suatu struktur, yaitu, mereka adalah objek yang berbeda; pada intinya, Mace4 beroperasi di bawah asumsi $0 \neq 1, 0 \neq 2$, dan seterusnya. Bagi Prover9, bilangan asli hanyalah konstanta biasa. Ini adalah perbedaan halus antara Prover9 dan Mace4. Karena Mace4 mengasumsikan bahwa bilangan asli adalah anggota domain, jika sebuah formula berisi bilangan asli yang berada di luar jangkauan ($\geq n$, ketika mencari struktur berukuran n), Mace4 akan berhenti dengan kesalahan fatal.

Mace4 dan Prover9 memiliki batasan yang sama pada formula tujuan yang diterimanya. Mace4 meniadakan tujuan dan menerjemahkannya ke dalam klausa dengan cara yang sama seperti Prover9. Istilah "tujuan" tidak terlalu intuitif bagi pengguna Mace4, karena Mace4 tidak membuktikan sesuatu. Namun, hal itu lebih masuk akal jika seseorang menganggap Mace4 sebagai pencari contoh tandingan untuk tujuan tersebut. Mace4 menggunakan perintah berikut untuk menentukan ukuran domain awal, ukuran domain maksimal, dan peningkatan ukuran domain.

```
assign(domain_size, n).      % default n=2, range [2 .. 200]
                             % command-line -n n
assign(iterate_up_to, n).    % default n=10, range [-1 .. 200]
                             % command-line -N n
assign(increment, n).        % default n=1, range [1 .. 200]
                             % command-line -i n
```

Ketiga parameter ini bekerja bersama untuk menentukan ukuran domain yang akan dicari. Pencarian dimulai untuk struktur dengan ukuran domain_size; jika pencarian itu gagal, ukurannya ditambah, dan pencarian lain dimulai. Ini berlanjut hingga nilai iterate_up_to (atau sampai batas lain mengakhiri proses).

Contoh 7.5.1 Pada Bagian 4.3.3, kami menunjukkan cara menemukan persegi Latin dengan batasan khusus oleh pemecah SAT. Kami menggunakan variabel proposisional $p_{x,y,z}$ untuk mewakili arti bahwa sel pada baris x dan kolom y berisi nilai z . Hubungan yang sama antara x, y , dan z dapat ditentukan dengan cara yang sama oleh predikat $p_{(x,y,z)}$. Dengan demikian, kelima himpunan klausa proposisional dapat ditentukan oleh lima rumus orde pertama sebagai berikut:

1. $\forall x \forall y \exists z p(x, y, z).$
2. $\forall x \forall y \forall z \forall w ((z \doteq w) \mid \overline{p(x, y, z)} \mid \overline{p(x, y, w)}).$
3. $\forall x \forall y \forall z \forall w ((y \doteq w) \mid \overline{p(x, y, z)} \mid \overline{p(x, w, z)}).$
4. $\forall x \forall y \forall z \forall w ((x \doteq w) \mid \overline{p(x, y, z)} \mid \overline{p(w, y, z)}).$
5. $\forall x \forall y \forall z \forall w (\overline{p(y, x, z)} \mid \overline{p(z, y, w)} \mid p(w, y, x)).$

Rumus-rumus ini dapat dengan mudah diubah menjadi klausa dan dapat ditentukan dalam Mace4 sebagai berikut:

```
assign(domain_size, 5).

formulas(assumptions).
  p(x, y, f(x, y)).
  (z = w) | -p(x, y, z) | -p(x, y, w).
  (y = w) | -p(x, y, z) | -p(x, w, z).
  (x = w) | -p(x, y, z) | -p(w, y, z).
  -p(y, x, z) | -p(z, y, w) | p(w, y, x).
end_of_list.
```

Perhatikan bahwa fungsi f pada klausa pertama adalah fungsi Skolem dari $\exists z$. Mace4 akan menemukan model berukuran 5 untuk input ini, dan menampilkan nilai $p(x, y, z)$ sebagai matriks Boolean dengan 25 baris dan 5 kolom, format yang kurang mudah dibaca untuk persegi Latin. Dari arti $p(x, y, z)$, kita dapat merekonstruksi persegi Latin sebagai berikut:

```
0 2 1 4 3
4 1 3 2 0
3 4 2 0 1
1 0 4 3 2
2 3 0 1 4
```

Jika kita menentukan persegi Latin menggunakan $x * y = z$ alih-alih $p(x, y, z)$, maka input ke Mace4 adalah sebagai berikut:

```
assign(domain_size, 5).
formulas(assumptions).
  x * z != y * z | x = y. % "!=" means "not equal"
  x * y != x * z | y = z.
  x * x = x. % idempotent law
  ((y * x) * y) * y = x. % special constraint
end_of_list.
```

Perhatikan bahwa " $x * z \neq y * z \mid x = y$ " menentukan bahwa jika baris x dan y berisi nilai yang sama pada kolom z , maka $x = y$. Demikian pula, " $x * y \neq x * z \mid y = z$ " menentukan bahwa jika kolom y dan z berisi nilai yang sama pada baris x , maka $y = z$. Spesifikasi

alternatif akan menggunakan $\backslash$ dan $/$ sebagai fungsi Skolem untuk u dan v , masing-masing, dalam

$$\forall x, y \exists! u, v (x * u \doteq y) \wedge (v * y \doteq x)$$

di mana $\exists!$ berarti “ada secara unik.” Sekarang, input Mace4 akan terlihat seperti

```
assign(domain_size, 5).

formulas(assumptions).
  % quasigroup axioms (equational)
  x * (x \ y) = y.
  x \ (x * y) = y.
  (x / y) * y = x.
  (x * y) / y = x.
  ((y * x) * y) * y = x. % special constraint
end_of_list.
```

Mace4 akan menemukan model berikut:

```
interpretation( 5, [number=1, seconds=0], [
  function(*( , ), [ function(/(_,_),[ function($_,_), [
    0, 2, 1, 4, 3, 0, 4, 3, 1, 2, 0, 2, 1, 4, 3,
    3, 1, 4, 0, 2, 4, 1, 0, 2, 3, 3, 1, 4, 0, 2,
    4, 3, 2, 1, 0, 3, 0, 2, 4, 1, 4, 3, 2, 1, 0,
    2, 4, 0, 3, 1, 1, 2, 4, 3, 0, 2, 4, 0, 3, 1,
    1, 0, 3, 2, 4 ]), 2, 3, 1, 0, 4 ]), 1, 0, 3, 2, 4 ]
  ]).
```

Persegi Latin yang didefinisikan oleh $*$ berbeda dari yang diperoleh oleh $p(x, y, z)$. Jika kita menggunakan perintah `assign(max_models, -1)`, Mace4 akan menemukan keenam model tersebut ketika ukuran domainnya adalah 5; ia akan menemukan 120 model ketika ukuran domainnya adalah 7 (Anda perlu mematikan perintah `auto`).

Pencarian Model Terbatas dengan Pemecah SAT

Versi Mace sebelumnya (sebelum Mace3) didasarkan pada fakta bahwa himpunan C klausa orde pertama memiliki model terbatas jika rumus proposisional yang diterjemahkan dari C dapat dipenuhi. Misalkan $D = \{0, 1, \dots, n - 1\}$ adalah domain dari model terbatas I , kita akan melakukan hal berikut untuk mendapatkan himpunan klausa proposisional dari C :

- Untuk setiap simbol predikat p/k , definisikan himpunan variabel proposisional $q_p^{x_1, \dots, x_k} \mid x_i \in D$, sedemikian sehingga $q_{x_1, \dots, x_k}^p$ bernilai benar jika dan hanya jika $p^I(x_1, \dots, x_k)$ bernilai benar.

- Untuk setiap simbol fungsi f/k , definisikan himpunan variabel proposisional $q_{x_1, \dots, x_k, y}^f$, $x_1, y \in D$, sedemikian sehingga $q_{x_1, \dots, x_k}^f$ bernilai benar jika dan hanya jika $f^I(x_1, \dots, x_k) = y$, dan tambahkan klausa berikut ke dalam C :

$$(f(x_1, \dots, x_k) \neq y_1 \mid f(x_1, \dots, x_k) \neq y_2 \mid y_1 \doteq y_2)$$

di mana $s \neq t$ berarti $\neg(s \doteq t)$

- Untuk setiap klausa A dari C , jika $f(t_1, \dots, t_k)$ muncul di A dan bukan dalam format $f(x_1, \dots, x_k) \doteq y$ (atau $y \doteq f(x_1, \dots, x_k)$), maka ganti A dengan $A[f(t_1, \dots, t_k) \sim z] \mid f(t_1, \dots, t_k) \neq z$, di mana z adalah variabel baru. Langkah ini disebut perataan. Setelah proses ini berhenti, setiap atom dalam C akan menjadi $p(x_1, \dots, x_k)$ atau $f(x_1, \dots, x_k) \doteq y$, di mana $x_1, \dots, x_k, y$ adalah variabel bebas.
- Untuk setiap klausa yang diratakan A dari C , untuk setiap variabel x dari A , untuk setiap nilai d dari D , buat semua contoh dasar dari A . Misalkan himpunan semua klausa dasar adalah G .
- Terakhir, untuk setiap klausa dasar dalam G , ganti $p(d_1, \dots, d_k)$ dengan $q_{d_1, \dots, d_k}^p$ dan $f(d_1, \dots, d_k) \doteq d$ dengan $q_{d_1, \dots, d_k, d}^f$. Misalkan himpunan semua klausa proposisional adalah $\text{propo}(C)$.

Contoh 7.5.2 Untuk spesifikasi persamaan dari masalah persegi Latin, kita menggunakan tiga simbol fungsi: $*$, $\backslash$, dan $/$. Variabel proposisional akan menjadi $p_{x,y,z}$ untuk $x * y \doteq z$, $q_{x,y,z}$ untuk $x \backslash y \doteq z$, dan $r_{x,y,z}$ untuk $x / y \doteq z$; jika $|D| = 5$, maka akan ada 375 variabel proposisional. Klausa yang diratakan dari aksioma ditambah kendala fungsional adalah

$(x \backslash y \neq z \mid x * z \doteq y)$	// $x * (x \backslash y) \doteq y$.
$(x * y \neq z \mid x \backslash z \doteq y)$	// $x \backslash (x * y) \doteq y$.
$(x / y \neq z \mid z * y \doteq x)$	// $(x / y) * y \doteq x$.
$(x * y \neq z \mid z / y \doteq x)$	// $(x * y) / y \doteq x$.
$(y * z \neq z \mid z * y \neq w \mid w * y \doteq x)$	// $((y * x) * y) * y \doteq x$.
$(x * y \neq z) \mid x * y \neq w \mid z \doteq w$	// " $*$ " is a function
$(x \backslash y \neq z) \mid x \backslash y \neq w \mid z \doteq w$	// " $\backslash$ " is a function
$(x / y \neq z) \mid x / y \neq w \mid z \doteq w$	// " $/$ " is a function

Jika $|D| = 5$, sebuah klausa dengan tiga variabel bebas akan menghasilkan $5^3 = 125$ klausa dasar; sebuah klausa dengan empat variabel bebas akan menghasilkan $5^4 = 625$ klausa dasar. Instans dasar dari klausa-klausa di atas mudah dikonversi ke klausa proposisional. Untuk literal $(z = w)$, instans dasarnya menjadi T jika kita memiliki $z\sigma = w\sigma = d$ dan $\perp$ jika $z\sigma = d_1 \neq d_2 = w\sigma$.

Meskipun konversi dari rumus orde pertama ke himpunan klausa proposisional menggunakan kesamaan "=", aksioma kesamaan (diperkenalkan di bagian pertama bab ini)

tidak diperlukan untuk konversi karena hanya menghasilkan klausa proposisional trivial. Hanya aksioma fungsi yang diperlukan dalam konversi untuk setiap fungsi f :

$$(f(x_1, \dots, x_k) = y_1 \mid f(x_1, \dots, x_k) = y_2 \mid y_1 \doteq y_2)$$

Proposisi 7.5.3 Suatu formula orde pertama C dalam CNF memiliki model hingga jika dan hanya jika formula proposisional $propo(C)$ dapat dipenuhi.

Bukti Misalkan C adalah formula CNF dari (P, F, X, Op) dan $D = \{0, 1, \dots, n-1\}$ adalah domain hingga.

Jika $propo(C)$ dapat dipenuhi, maka $propo(C)$ memiliki model σ dan σ mendefinisikan relasi R^p atas D^k untuk setiap simbol predikat p/k :

$$R^p = \{\langle d_1, \dots, d_k \rangle \mid \sigma(q_{d_1, \dots, d_k}^p) = 1\}$$

dan sebuah fungsi $f^\sigma: D^k \rightarrow D$ untuk setiap fungsi f/k

$$f^\sigma(d_1, \dots, d_k) \doteq d \text{ iff } \sigma(q_{d_1, \dots, d_k, d}^f) = 1$$

Kemudian, siap untuk memeriksa apakah $I = (D, \{R^p \mid p \in P\}, \{f^m \mid f \in F\})$ adalah model dari C .

Di sisi lain, jika C memiliki model terbatas $I = (D, R, G)$, di mana D terbatas, maka kita dapat membuat interpretasi σI untuk $propo(C)$ sebagai berikut:

$$\sigma_I(q_{d_1, \dots, d_k}^p) = 1 \text{ iff } p^I(d_1, \dots, d_k) = 1$$

Dan

$$\sigma_I(q_{d_1, \dots, d_k, d}^f) = 1 \text{ iff } f^I(d_1, \dots, d_k) = d$$

Ia siap untuk memeriksa bahwa σ adalah model dari $propo(C)$.

Untuk menggunakan pembukti SAT untuk menyelesaikan masalah dengan kesulitan sedang, kita perlu menulis *encoder* yang menghasilkan klausa proposisional dan kemudian memasukkan klausa-klausa ini ke dalam pemecah SAT. Kita juga membutuhkan *decoder* yang menerjemahkan solusi pemecah SAT menjadi solusi untuk masalah tersebut. Mace4 mengurangi beban penulisan encoder/decoder dengan secara otomatis mengkonversi klausa orde pertama menjadi klausa proposisional dengan variabel bebas yang digantikan oleh semua nilai dalam domain terbatas dan kemudian memanggil pemecah SAT untuk menemukan model; kemudian mengkonversi model proposisional menjadi model orde pertama.

Versi Mace yang lebih baru, termasuk Mace4, mempertahankan serangkaian klausa dasar dengan kesamaan dan bekerja dengan cara yang sama seperti SEM, generator model awal yang dikembangkan oleh penulis buku ini sebelum Mace dibuat. Hal ini karena metode yang didasarkan pada metode DPLL bekerja dengan baik dalam banyak kasus, tetapi jika masalah aslinya memiliki klausa dengan banyak variabel atau istilah yang bersarang dalam, proses transformasi ke klausa SAT akan gagal, dan hanya model yang sangat kecil yang dapat dicari. Pembaca yang tertarik dapat merujuk pada manual pengguna Mace4 atau pengantar SEM untuk detailnya.

7.6 LATIHAN SOAL

1. Tunjukkan bahwa $(f(x_1, \dots, x_k) \neq y_1 \mid f(x_1, \dots, x_k) \neq y_2 \mid y_1 \doteq y_2)$ dari aksioma kesamaan.
2. Tuliskan dalam logika orde pertama pernyataan bahwa terdapat setidaknya, paling banyak, dan tepat tiga elemen x sedemikian sehingga $A(x)$ berlaku.
3. Berapa banyak pasangan kritis yang dapat dihitung antara aturan penulisan ulang dalam himpunan akhir Contoh 7.2.14? Hitunglah berdasarkan pasangan aturan.
4. Dengan menambahkan persamaan $a * b \doteq b * a$ ke sistem penulisan ulang Contoh 7.2.14, gunakan penyelesaian Knuth-Bendix untuk mendapatkan sistem penulisan ulang kanonik dengan LPO dan prioritas $a > b > i > * > e$.
5. Diberikan himpunan persamaan dasar berikut

$$E = \{I \doteq J, K \doteq L, A[I] \doteq B[K], J \doteq A[J], M \doteq B[L]\},$$

Jalankan penyelesaian Knuth-Bendix pada E dengan KBO, dengan asumsi $B > A > M > L > K > J > I$, dan tunjukkan bagaimana setiap aturan penulisan ulang dihasilkan oleh prosedur tersebut.

6. Misalkan $W = \{aaa \doteq \epsilon, bbb \doteq \epsilon, ababab \doteq \epsilon\}$ adalah persamaan atas $\{a, b\}^*$. Ubah W menjadi persamaan suku dan jalankan penyelesaian Knuth-Bendix padanya dengan urutan penyederhanaan LPO (atau RPO) dan prioritas $b > a$.
7. Definisikan secara formal konsep pasangan kritis untuk sistem penulisan ulang string dan ilustrasikan konsep tersebut dengan $R = \{ab \rightarrow ba, bc \rightarrow cb\}$. Dengan asumsi $a > b > c$ dan string dibandingkan secara leksikografis. Dapatkah Anda memperoleh sistem penulisan ulang kanonik dari R ? Mengapa?
8. Dengan asumsi $x > y > z$, jalankan penyelesaian Knuth-Bendix pada sistem polinomial berikut:

$$\begin{array}{ll} (a) & (1) \quad x^7y^2 + x^3y^2 - y + 1 \doteq 0 \\ & (2) \quad xy^2 - x \doteq 0 \end{array}$$

$$\begin{array}{ll} (b) & (1) \quad x^3 - x^2y - x^2z + x \doteq 0 \\ & (2) \quad x^2y - z \doteq 0 \end{array}$$

$$(c) \quad (1) \quad x^4y^2 - z \doteq 0$$

$$(2) \quad x^3 y^3 - 1 \doteq 0$$

$$(3) \quad x^2 y^4 - 2z \doteq 0$$

9. Buktikan Lemma 7.3.4.

10. Berdasarkan definisi penjumlahan dan perkalian pada Contoh 7.3.3, buktikan dengan induksi struktural teorema induktif berikut:

- (a) $add(x, s(y)) \doteq s(add(x, y))$
- (b) $add(x, y) \doteq add(y, x)$
- (c) $mul(x, y) \doteq mul(y, x)$
- (d) $add(add(x, y), z) \doteq add(x, add(y, z))$
- (e) $mul(mul(x, y), z) \doteq mul(x, mul(y, z))$

11. Buktikan bahwa aturan-aturan berikut dari Contoh 7.3.3 berakhir dengan urutan penyederhanaan:

- (a) $exp(x, 0) \rightarrow s(0)$
- (b) $exp(x, s(y)) \rightarrow mul(x, exp(x, y))$

di mana $exp(x, y)$ menghitung x^y , fungsi eksponensial. Buktikan dengan induksi struktural bahwa $exp(x, add(y, z)) \doteq mul(exp(x, y), exp(x, z))$

12. Buktikan bahwa aturan-aturan berikut ini berakhir dengan urutan penyederhanaan:

- (a) $fib(0) \rightarrow 0$
- (b) $fib(s(0)) \rightarrow s(0)$
- (c) $fib(s(s(x))) \rightarrow add(fib(s(x)), fib(x))$
- (d) $sum(0) \rightarrow 0$
- (e) $sum(s(x)) \rightarrow add(sum(x), fib(s(x)))$

di mana $fib(x)$ adalah bilangan Fibonacci dan $sum(x)$ adalah penjumlahan dari x bilangan Fibonacci pertama. Tambahkan aturan-aturan ini ke Contoh 7.3.3 dan buktikan dengan induksi struktural bahwa $sum(x) \doteq pre(fib(s(s(x))))$ adalah benar.

13. Buktikan bahwa aturan-aturan berikut ini berakhir dengan urutan penyederhanaan:

- (a) $A(0, y) \rightarrow s(y)$
- (b) $A(s(x), 0) \rightarrow A(x, s(0))$
- (c) $A(s(x), s(y)) \rightarrow A(x, A(s(x), y))$

di mana $A(x, y)$ adalah fungsi Ackermann. Tambahkan aturan-aturan ini ke Contoh 7.3.3 dan buktikan dengan induksi struktural bahwa $A(s(0), y) \doteq s(s(y))$ dan $A(s(s(0)), y) \doteq s(s(s(add(y, y))))$.

14. Misalkan $C = \{0, s, p\}$ adalah konstruktor untuk memodelkan himpunan bilangan bulat dengan $E_C = \{p(s(x)) \doteq x, s(p(x)) \doteq x\}$:

- (a) Berikan definisi C -lengkap untuk add (penjumlahan bilangan bulat) dan sub (pengurangan bilangan bulat).
- (b) Buktikan bahwa add bersifat komutatif dan asosiatif.
- (c) Buktikan $sub(add(x, y), y) \doteq x$ dan $add(sub(x, y), y) \doteq x$.

15. Diberikan definisi app dan rev pada Contoh 7.3.10, buktikan bahwa

- (a) $app(x, nil) \doteq x$
- (b) $app(x, app(y, z)) \doteq app(app(x, y), z)$
- (c) $rev(app(x, y)) \doteq app(rev(y), rev(x))$

adalah teorema induktif dengan induksi struktural.

16. Diberikan definisi *isort* dalam Contoh 7.3.12, buktikan bahwa $sorted(isort(y)) = T$ adalah teorema induktif dengan induksi struktural.
17. Gunakan Prover9 untuk menjalankan penyelesaian Knuth-Bendix pada hal berikut:
 - (a) Tiga persamaan dalam Contoh 7.2.8
 - (b) Persamaan $((x \uparrow y) \uparrow z) \uparrow (x \uparrow ((x \uparrow z) \uparrow x)) \cdot z$ dalam Proposisi 2.4.5.
18. Gunakan Mace4 untuk menentukan apakah ada persegi Latin berukuran antara 4 dan 9 untuk masing-masing kendala berikut (disebut “identitas ortogonal konjugasi pendek”), secara berturut-turut.

Code	Constraint			Name
QG3	$(x * y) * (y * x)$	$\doteq$	x	<i>Schröder quasigroup</i>
QG4	$(y * x) * (x * y)$	$\doteq$	x	<i>Stein's third law</i>
QG5	$((y * x) * y) * y$	$\doteq$	x	
QG6	$(x * y) * y$	$\doteq$	$x * (x * y)$	<i>Schröder's first law</i>
QG7	$(y * x) * y$	$\doteq$	$x * (y * x)$	<i>Stein's second law</i>
QG8	$x * (x * y)$	$\doteq$	$y * x$	<i>Stein's first law</i>
QG9	$((x * y) * y) * y$	$\doteq$	x	<i>C3 – quasigroup</i>

19. Ulangi masalah sebelumnya ketika kendala tambahan $x * x \doteq x$, dimasukkan ke dalam himpunan aksioma.

BAB 8

PROLOG PEMROGRAMAN DALAM LOGIKA

Prolog adalah bahasa pemrograman yang namanya berasal dari pemrograman dalam logika. Tidak seperti banyak bahasa pemrograman lainnya, Prolog berasal dari logika orde pertama dan dimaksudkan terutama sebagai bahasa pemrograman deklaratif: program diekspresikan dalam bentuk relasi, yang direpresentasikan sebagai fakta dan aturan Prolog. Komputasi dimulai dengan menjalankan kueri atas relasi-relasi ini.

Berdasarkan prinsip resolusi Robinson, Prolog pertama kali dikembangkan pada tahun 1972 oleh Alain Colmerauer bersama Philippe Roussel, yang disebut "Marseille Prolog." Proyek Komputer Generasi Kelima Jepang, yang diumumkan pada tahun 1981, mengadopsi Prolog sebagai bahasa default komputer dan dengan demikian memfokuskan perhatian yang cukup besar pada bahasa dan kemampuannya. Prolog adalah contoh bahasa pemrograman generasi keempat yang mendukung paradigma pemrograman deklaratif dan secara luas dianggap sebagai bahasa yang sangat baik untuk "pemrograman eksploratif" dan "pemrograman prototipe".

Saat ini tersedia beberapa implementasi Prolog gratis, seperti gprolog GNU dan SWI-prolog, dan keduanya merupakan turunan dari Marseille Prolog. Bahasa ini telah digunakan untuk pembuktian teorema, sistem pakar, perencanaan otomatis, dan pemrosesan bahasa alami, serta merupakan alat populer dalam kecerdasan buatan dan linguistik komputasional. Prolog sangat cocok untuk tugas-tugas spesifik yang mendapat manfaat dari kueri logika berbasis aturan seperti basis data dan sistem kontrol suara.

8.1 PRINSIP KERJA PROLOG

Dalam pemrograman logika, logika program diekspresikan dalam bentuk sekumpulan fakta dan aturan, yang mendefinisikan relasi. Komputasi dimulai dengan menjalankan kueri atas relasi-relasi ini.

Klausa Horn dalam Prolog

Dalam pemrograman logika, relasi dan kueri biasanya dibangun dengan klausa Horn. Pada Bagian 3.4, kita telah melihat bahwa klausa Horn adalah klausa dengan paling banyak satu literal positif. Dalam Prolog, klausa Horn dibagi menjadi tiga jenis:

- **fakta:** klausa positif tunggal
- **aturan:** klausa dengan satu literal positif dan satu atau lebih literal negatif
- **kueri:** klausa negatif

Program Prolog terdiri dari fakta dan aturan, yang berisi tepat satu literal positif dan secara kolektif disebut klausa definit. Klausa aturan adalah $(A \mid \overline{B_1} \mid \dots \mid \overline{B_n})$, di mana $A, B_1, \dots, B_n, n \geq 1$, adalah rumus atomik (atom). Klausa aturan ditulis dalam Prolog sebagai

$$A : -B_1, \dots, B_n.$$

Format ini sangat mirip dengan rumus berikut:

$$B_1 \wedge \dots \wedge B_n \rightarrow A.$$

Dengan kata lain, simbol $:$ adalah kebalikan dari $\rightarrow$ dan koma adalah $\wedge$ dalam Prolog. Untuk aturan di atas, A disebut kepala dan " $B_1, \dots, B_n$ " disebut badan. Aturan ini dibaca sebagai " A benar jika " $B_1, \dots, B_n$ " semuanya benar." Fakta Prolog adalah klausa dengan badan kosong dan tidak memerlukan simbol $:$. Secara ekuivalen, kita dapat menulis fakta dalam Prolog sebagai " $A : - \text{benar.}$ "

Prolog menggunakan klausa negatif $(\overline{B_1} | \dots | \overline{B_n})$ untuk kueri. Setiap kueri Prolog ditulis sebagai $?- B_1, \dots, B_n$, yang menyerupai rumus ekuivalen $\neg(B_1 \wedge \dots \wedge B_n)$.

Contoh 8.1.1 Misalkan kita ingin membangun pohon keluarga dari keluarga besar. Kita dapat menyatakan hubungan ayah-anak dan ibu-anak dengan fakta dan mendefinisikan hubungan lain dengan aturan. Berikut adalah program Prolog untuk pohon keluarga.

```

mom(rose, john).      % Rose is the mother of John
mom(rose, bob).
mom(rose, ted).
papa(joe, john).      % Joe is the father of John
papa(joe, bob).
papa(joe, ted).
papa(ted, patrick).
papa(john, caroline).
papa(bob, kathleen).
papa(bob, joseph).
papa(bob, mary).
parent(X, Y) :- papa(X, Y).      % X is parent of Y if X is papa of Y.
parent(X, Y) :- mom(X, Y).      % X is parent of Y if X is mom of Y.
grandpa(X, Y) :- papa(X, Z), parent(Z, Y).
grandmo(X, Y) :- mom(X, Z), parent(Z, Y).

```

Dari program ini, kita melihat bahwa variabel dalam Prolog adalah pengidentifikasi yang karakter pertamanya adalah huruf kapital. Setelah program dimuat ke dalam Prolog, kita dapat mengeluarkan kueri setelah simbol $?-$.

Untuk menanyakan siapa orang tua dari ted, setelah simbol $?-$, kita ketik `parent(X, ted)`. Setelah kita menekan "enter", Prolog akan menjawab dengan `X = joe`. Jika Anda mengetik `;`, yang berarti untuk "solusi lebih lanjut", Prolog akan memberi Anda jawaban lain: `X = rose`. Jika Anda menekan `;` lagi, Prolog akan berhenti dengan "tidak", karena kehabisan jawaban. Untuk menanyakan relasi saudara kandung, kita mendefinisikan relasi saudara kandung sebagai sebuah aturan.

```

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
% X and Y share parent Z.

```

di mana $\neq$ berarti “tidak sama dengan.” Sekarang, hasilnya dapat diperoleh dengan kueri tersebut.

?- sibling(X, Y).

Di sisi lain, kita dapat mengeluarkan permintaan tersebut.

?- parent(Z, X), parent(Z, Y), X $\neq$ Y.

Kueri ini mencetak induk umum Z dari X dan Y . Misalnya, jawaban pertama yang ditemukan oleh Prolog adalah

X = john, Y = bob, Z = joe.

Bukti Resolusi dalam Prolog

Dalam Prolog, terkait dengan setiap kueri yang berhasil, terdapat bukti resolusi. Untuk program Prolog pada Contoh 8.1.1, kita memiliki jawaban X = joe untuk kueri “?- parent(X, ted).” Bukti resolusi yang sesuai diberikan di bawah ini, di mana p mewakili parent dan f mewakili papa.

<i>Prolog format</i>	<i>clausal format</i>	
1 papa(joe, ted).	$(f(joe, ted))$	// input
2 parent(X, Y) :- papa(X, Y).	$(p(X, Y) \mid \overline{f(X, Y)})$	// input
3 ?- parent(X, ted).	$(\overline{p(X, ted)})$	// query
4 ?- papa(X, ted).	$(\overline{f(X, ted)})$	// resolvent of 2 and 3
5 $\perp$	$\perp$	// resolvent of 1 and 4

Dalam resolusi terakhir, mgu adalah $\{X \mapsto joe\}$, yang merupakan jawaban atas kueri tersebut. Kueri tersebut berasal dari negasi rumus $\exists X p(X, ted)$, yang merupakan konsekuensi logis dari klausa masukan.

Sebagai contoh lain, kami mencantumkan di bawah ini bukti resolusi yang sesuai untuk kueri “parent(Z, X), parent(Z, Y), X $\neq$ Y.” Perhatikan lagi bahwa “?- A, B, C” menunjukkan klausa $(\neg A \mid \neg B \mid \neg C)$.

1 papa(joe, john).	// input
2 papa(joe, bob).	// input
3 parent(X, Y) :- papa(X, Y).	// input
4 ?- parent(Z, X), parent(Z, Y), X $\neq$ Y.	// query
5 ?- papa(Z, X), parent(Z, Y), X $\neq$ Y.	// resolvent of 3 and 4
6 ?- parent(joe, Y), john $\neq$ Y.	// resolvent of 1 and 5
7 ?- papa(joe, Y), john $\neq$ Y.	// resolvent of 3 and 6
8 ?- john $\neq$ bob.	// resolvent of 2 and 7
9 $\perp$	// $\perp$ denotes the empty clause,

Mgus yang digunakan dalam resolusi memberikan solusi dari kueri: resolusi antara 1 dan 5 memberikan $X = \text{john}$, $Z = \text{joe}$ dan resolusi antara 2 dan 7 memberikan $Y = \text{bob}$.

Kedua bukti resolusi di atas adalah bukti resolusi negatif, masukan, dan linier. Artinya, setiap resolusi mencakup klausa negatif dan klausa masukan sebagai induk, dan resolven terbaru (yang negatif) digunakan dalam resolusi berikutnya. Ini berlaku untuk setiap bukti resolusi yang ditemukan oleh Prolog. Korolari 3.4.11 menyatakan bahwa, untuk klausa Horn proposisional, jika hanya ada satu klausa negatif, maka resolusi negatif, masukan, dan linier lengkap untuk klausa Horn. Hasil ini dapat diperluas ke klausa orde pertama. Dengan kata lain, strategi resolusi negatif, masukan, dan linier lengkap untuk klausa Horn orde pertama. Namun, strategi pencarian mendalam yang digunakan oleh Prolog bukanlah strategi yang adil. Dengan demikian, meskipun ada bukti resolusi negatif, masukan, dan linier untuk sekumpulan klausa Horn, Prolog mungkin gagal menemukan bukti. Kita akan membahas masalah ini ketika berbicara tentang rekursi di Prolog.

Singkatnya, diberikan sebuah kueri, mesin Prolog mencoba menemukan bukti resolusi dari program dan kueri tersebut. Jika klausa kosong dapat diturunkan, maka kueri tersebut, bersama dengan substitusi yang ditemukan yang diterapkan, merupakan konsekuensi logis dari program tersebut. Hal ini membuat Prolog sangat berguna untuk komputasi simbolik dan aplikasi penguraian bahasa.

Contoh 8.1.2 Misalkan kita diminta untuk menulis program untuk mengubah formula proposisional menjadi bentuk normal negasi. Ini akan menjadi latihan sehari-hari menggunakan bahasa pemrograman konvensional. Misalkan kita hanya menggunakan tiga variabel proposisional, katakanlah p, q, r , dan operator logikanya adalah “ n ” untuk $\neg$, “ a ” untuk $\wedge$, “ o ” untuk $\vee$, “ i ” untuk $\rightarrow$, dan “ e ” untuk $\leftrightarrow$, program Prolog diberikan di bawah ini, di mana $nnf(X, Y)$ bernilai benar jika dan hanya jika Y adalah NNF dari X .

```
pvar(p). % definition of propositional variables
pvar(q).
pvar(r).

literal(n(L)) :- pvar(L).      % definition of literals
literal(L) :- pvar(L).

nnf(n(n(X)), NNF) :- nnf(X, NNF). % double negation

nnf(a(A, B), a(N1, N2)) :- nnf(A, N1), nnf(B, N2). % and
nnf(n(a(A, B)), o(N1, N2)) :- nnf(n(A), N1), nnf(n(B), N2).

nnf(o(A, B), o(N1, N2)) :- nnf(A, N1), nnf(B, N2). % or
nnf(n(o(A, B)), a(N1, N2)) :- nnf(n(A), N1), nnf(n(B), N2).
```

```

nnf(i(A, B), o(N1, N2)) :- nnf(n(A), N1), nnf(B, N2). % implication
nnf(n(i(A, B)), a(N1, N2)) :- nnf(A, N1), nnf(n(B), N2).

nnf(e(A, B), o(a(NA, NB), a(NnA, NnB))) :- % equivalence
    nnf(A, NA), nnf(n(A), NnA), nnf(B, NB), nnf(n(B), NnB).
nnf(n(e(A, B)), o(a(NA, NnB), a(NnA, NB))) :-
    nnf(A, NA), nnf(n(A), NnA), nnf(B, NB), nnf(n(B), NnB).

nnf(X, X) :- literal(X).          % NNF of literal is itself

```

Berikut ini adalah hasil dari kueri ke program di atas:

```

?- nnf(n(o(n(a(p,q)),e(p,q))), X).
X = a(a(p,q),o(a(p,n(q)),a(n(p),q))).

```

Program Prolog di atas dapat diselesaikan dalam beberapa jam oleh seorang mahasiswa.

Prosedur Reduksi Tujuan

Mesin Prolog dapat dianggap sebagai proses reduksi tujuan: untuk setiap atom G dalam kueri asli, mesin akan mencari aturan dalam program Prolog yang kepalanya menyatu dengan G dan secara rekursif menerapkan mesin ke sub-tujuan, yang merupakan atom dalam badan aturan. Jika badan kosong, maka aturan ini adalah fakta. Ketika semua sub-tujuan G diselesaikan, atau ketika badan kosong, tujuan G dianggap telah diselesaikan dan substitusi untuk variabel G dikembalikan; jika tidak, label "gagal" dikembalikan.

Prosedur 8.1.3 Dengan asumsi P adalah program Prolog, prosedur engine_1 mengambil satu tujuan G sebagai input dan mengembalikan substitusi sebagai output jika satu aturan P dapat menyelesaikan G ; jika tidak, gagal dikembalikan. Prosedur engine mengambil daftar G_s tujuan sebagai input dan mengembalikan substitusi sebagai output jika setiap tujuan G_s diselesaikan oleh P .

```

proc engine1(G)
1   for (H : - B) ∈ P do
2       if (H and G are unifiable with mgu σ)
3           θ := engine(Bσ)
4           if θ ≠ fail return σθ
5   return fail

proc engine(Gs)
1   θ := []
2   for A ∈ Gs do
3       σ := engine1(Aθ)
4       if σ = fail break else θ := θσ
5   if σ = fail return fail else return θ

```

Berikut ini diasumsikan dalam Prolog: $H: \neg B$ adalah fakta jika B kosong. $(H: \neg B) \in P$ memilih $H: \neg B$ di P (dan $A \in Gs$ memilih A di Gs) dari pertama hingga terakhir. Panggilan pertama adalah $engine(Qs)$, di mana Qs adalah tujuan dalam sebuah kueri.

Jelas, $engine_1$ dan $engine$ adalah prosedur rekursif bersama. Jika kita memeriksa pohon rekursi $engine_1$ dan $engine$, tautan yang keluar dari node $engine_1$ diberi label oleh aturan P ; dan tautan yang keluar dari node $engine$ diberi label oleh tujuan di Gs . Node $engine_1$ adalah node V : jika salah satu anaknya berhasil, ia akan berhasil. Node $engine$ adalah node Λ : jika salah satu anaknya gagal, ia akan gagal. Dengan kata lain, pohon rekursi $engine$ dan $engine_1$ adalah pohon $\Lambda - V$.

Prosedur $engine_1$ seperti yang diberikan bertujuan untuk menemukan satu jawaban; Jika lebih banyak jawaban diinginkan, maka baris 4 dari $engine_1$ perlu dimodifikasi untuk mengingat aturan mana yang digunakan untuk mendapatkan σ dan akan menggunakan aturan berikutnya ketika lebih banyak solusi diinginkan. Dalam hal ini, node $engine_1$ mungkin memiliki lebih dari satu anak yang berhasil.

Node daun dalam pohon $\Lambda - V$ dikatakan baik jika (a) itu adalah node $engine$ dan Gs kosong atau (b) itu adalah node $engine_1$ dan G adalah atom bawaan yang dievaluasi menjadi benar. Node internal dalam pohon $\Lambda - V$ dikatakan baik jika (c) itu adalah node $engine$ dan setiap node anak baik atau (d) itu adalah node $engine_1$ dan salah satu anaknya baik.

Dalam hal ini, pohon $\Lambda - V$ memiliki solusi jika dan hanya jika node akarnya baik, dan itu mewakili semua solusi yang ditentukan oleh node V .

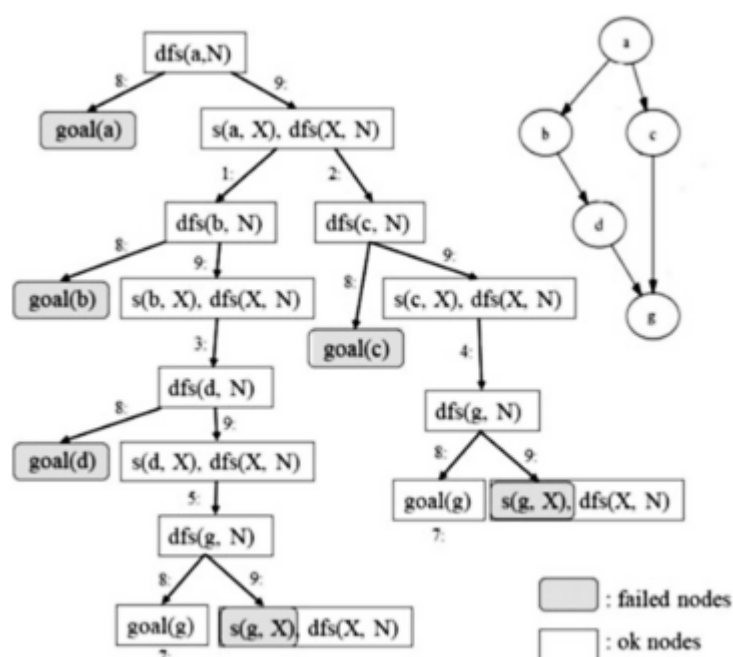
Contoh 8.1.4 Program berikut mengimplementasikan prosedur pencarian mendalam untuk grafik berarah:

```
% s(u, v): directed edge (u, v)
1: s(a,b).
2: s(a,c).
3: s(b,d).
4: s(c,g).
5: s(d,g).
6: goal(e). % goal(x): x is a goal
7: goal(g).
8: dfs(N, N) :- goal(N). % dfs(startNode, solution)
9: dfs(N, l(N, R)) :- s(N, X), dfs(X, R).
```

Ada dua solusi untuk kueri “?- dfs(a, N).” Yaitu, $N = l(a, l(b, l(d, g)))$ dan $N = l(a, l(c, g))$, di mana istilah seperti $l(a, l(c, g))$ mewakili jalur (a, c, g) dalam grafik. Kemudian, setelah memasukkan daftar, jalur yang sama dapat diwakili oleh $[a, c, g]$. Gambar 8.1 menunjukkan simpul-V dalam pohon pencarian untuk kueri “?- dfs(a, N).” Simpul- Λ bersifat implisit, simpul yang diarsir adalah simpul yang **gagal**, dan simpul lainnya adalah simpul yang berhasil.

Perhatikan bahwa dua prosedur rekursif *engine* dan *engine*₁ hanya mengilustrasikan proses reduksi tujuan; *engine* Prolog yang sebenarnya adalah prosedur interaktif dan menggunakan tumpukan untuk menyimpan semua informasi dalam pohon $\Lambda - V$.

Prosedur *engine*₁ menunjukkan bahwa urutan aturan dalam *P* itu penting. Prosedur *engine* menunjukkan bahwa urutan atom dalam badan aturan itu penting. Dengan kata lain, tautan dalam pohon $\Lambda - V$ diurutkan, dan ia mengimplementasikan kasus khusus dari strategi resolusi negatif, masukan, dan linier: literal dalam suatu aturan ditangani dari kiri ke kanan, dan koma, yang mewakili $\wedge$ dalam logika, tidak komutatif. Klausula masukan dicoba untuk disatukan dengan tujuan yang diberikan dari pertama hingga terakhir. Dari sudut pandang strategi pencarian, ini adalah pencarian mendalam (depth-first search) dalam pohon $\Lambda - V$.



Gambar 8.1 V-node dalam pohon Λ -V untuk kueri $?- \text{dfs}(a, N)$. Nomor klausa diberikan sebagai label tautan antar node.

Contoh 8.1.5 Menambahkan aturan berikut ke program Prolog pada Contoh 8.1.1:

```
descen(X, Y) :- parent(Y, X). % X is a descendant of Y
descen(X, Z) :- parent(Y, X), descen(Y, Z).
```

Aturan terakhir berbeda dari aturan sebelumnya karena *descen* didefinisikan secara rekursif. Jika kita bertanya, “siapa keturunan Joe?”, kuerinya adalah “ $?- \text{descen}(X, \text{joe})$ ”. Prolog akan menemukan bahwa $X = \text{john}, \text{bob}, \text{ted}, \text{patrick}, \text{caroline}, \text{kathleen}, \text{joseph}$, dan mary , dalam urutan yang diberikan. Untuk jawaban pertama, yaitu $X = \text{john}$, panggilan rekursif *engine* dan *engine*₁ berjalan sebagai berikut:

```

engine1(descen(X, joe))
  engine([parent(joe, X)])
    engine1(parent(joe, X))
      engine([papa(joe, X)])
        engine1(papa(joe, X)) return X = john
...
engine1([descen(X, joe)]) return X = john

```

Untuk jawaban terakhir, yaitu X=mary, panggilan rekursifnya adalah sebagai berikut. (panggilan ke *engine* diabaikan jika daftar hanya berisi satu tujuan):

```

engine1(descen(X, joe))
  engine([parent(Y, X), descen(Y, joe)])
    engine1(parent(Y, X))
      engine1(papa(Y, X)) return {X = mary, Y = bob}
    engine1(descen(bob, joe))
      engine1(parent(joe, bob))
        engine1(papa(joe, bob)) return ∅
...
  engine([parent(Y, X), descen(Y, joe)]) return {X = mary, Y = bob}
engine1(descen(X, joe)) return {X = mary, Y = bob}

```

Namun, jika kita mengubah aturan terakhir menjadi aturan yang secara logis setara.

```

descen(X, Z) :- descen(Y, Z), parent(Y, X)

```

Kueri yang sama, “?- descen(X, joe),” akan menghasilkan X = john, bob, ted, caroline, kathleen, joseph, mary, dan patrick, delapan jawaban yang sama dalam urutan yang berbeda, dan akan menyebabkan Prolog mengalami crash jika Anda meminta lebih banyak jawaban. Jika kita menukar urutan dua klausa terakhir, yaitu, dua klausa terakhir adalah

```

descen(X, Z) :- descen(Y, Z), parent(Y, X).
descen(X, Y) :- parent(Y, X).

```

Kueri yang sama akan menyebabkan Prolog mengalami crash tanpa menghasilkan jawaban apa pun. Logika orde pertama tidak dapat menjelaskan perbedaannya karena kedua klausa ini secara logis setara. Prosedur *engine* dan *engine₁* dapat menjelaskan perbedaannya dengan sempurna: Daftar tujuan awal adalah [descen(X, joe)] dalam ketiga kasus tersebut. Pada kasus terakhir, aturan descen(X, Z) :- descen(Y, Z), parent(Y, X) diterapkan terlebih dahulu, dan panggilan rekursif *engine* dan *engine₁* berjalan sebagai berikut:

```

engine1(descen(X, joe))
  engine([descen(Y, joe), parent(Y, X)])
    engine1(descen(Y, joe))
      engine([descen(Y1, joe), parent(Y1, Y)])
        engine1(descen(Y1, joe))
          engine([descen(Y2, joe), parent(Y2, Y1)])
...

```

Daftar tujuan yang belum terpecahkan berisi serangkaian tujuan yang panjang seperti...

```
..., parent(Y3, Y2), parent(Y2, Y1), parent(Y1, Y), parent(Y, X)
```

hingga Prolog mengalami crash ketika kehabisan memori. Di sini, kami mengubah nama variabel karena setiap klausa diasumsikan memiliki variabel yang berbeda dari yang lain.

Contoh di atas menunjukkan bahwa Prolog menggunakan strategi depth-first, yang bukan strategi yang adil, untuk mengimplementasikan resolusi, dan menjadi pembukti teorema yang tidak lengkap untuk klausa Horn. Sebagai programmer Prolog, seseorang perlu berhati-hati untuk menghindari jebakan loop tak terbatas yang terkait dengan strategi depth-first. Kita akan membahas masalah ini di bagian selanjutnya.

Contoh 8.1.6 Menara Hanoi, yang diperkenalkan dalam pengantar buku ini, meminta untuk memindahkan n cakram dengan ukuran berbeda dari pasak X ke pasak Y menggunakan pasak Z sebagai pasak penahan tambahan. Cakram yang lebih besar tidak boleh diletakkan di atas cakram yang lebih kecil. Kode Prolog berisi dua definisi predikat: `move` menuliskan informasi pemindahan; `hanoi(N,X,Y,Z)` memeriksa apakah $N = 1$, kemudian memindahkan cakram 1 dari X ke Y ; jika tidak, ia mencoba memindahkan $N-1$ cakram teratas dari X ke Z ; memindahkan cakram terbawah N dari X ke Y ; dan kemudian memindahkan $N-1$ cakram teratas dari Z ke Y .

```

move(N,X,Y) :-          % print a move info
  write('Move disk '), write(N), write(' from '),
  write(X), write(' to '), write(Y), nl. % 'nl' = newline

hanoi(1,X,Y,Z) :-       % disk 1 can move freely from X to Y move(1,X,Y).
hanoi(N,X,Y,Z) :-       % To move N disks from X to Y,
  N>1, M is N-1,        % where N > 1,
  hanoi(M,X,Z,Y),        % move the top N-1 disks from X to Z;
  move(N,X,Y),           % then move disk N, the largest, from X to Y;
  hanoi(M,Z,Y,X).        % then move the top N-1 disks from Z to Y.

?- hanoi(3, a, b, c).
Move disk 1 from a to b

```

```

Move disk 2 from a to c
Move disk 1 from b to c
Move disk 3 from a to b
Move disk 1 from c to a
Move disk 2 from c to b
Move disk 1 from a to b

```

Kode tersebut berasal langsung dari deskripsi solusi. Bagian dari aturan kedua hanoi berisi dua panggilan rekursif hanoi. Panggilan rekursif akan berakhir karena argumen pertama hanoi dikurangi satu pada setiap panggilan berikutnya.

8.2 TIPE DATA PROLOG

Sebagai bahasa pemrograman, Prolog memiliki beberapa tipe data bawaan, seperti angka (float atau integer), string, list, dan fungsi terkait. Tipe data ini merupakan subtype dari supertipe, yang disebut term. Dalam pandangan ini, term adalah satu-satunya tipe data dalam Prolog. Term dapat berupa atom, angka, variabel, atau term gabungan. Rumus atomik dalam Prolog disebut term yang dapat dipanggil.

Atom, Angka, dan Variabel

Kita telah menggunakan "atom" untuk "rumus atom"; di bagian ini, kita akan menggunakan "rumus atom" secara eksplisit karena "atom" memiliki arti khusus dalam Prolog: sebuah atom adalah pengidentifikasi selain variabel atau string karakter yang diapit tanda kutip tunggal. Contoh atom termasuk `x`, `merah`, `"Taco"`, dan `"some atom."` Atom adalah nama serbaguna tanpa makna inheren.

Prolog menyediakan predikat `atom/1`, di mana `atom(X)` bernilai benar jika dan hanya jika `X` adalah atom. Misalnya, `atom("some atom")` bernilai benar; `atom(123)` bernilai salah.

Angka dapat berupa bilangan pecahan atau bilangan bulat. Rentang angka bergantung pada jumlah bit (jumlah memori komputer) yang digunakan untuk merepresentasikan angka tersebut. Sebagian besar sistem Prolog utama mendukung bilangan bulat dengan panjang sembarang. Perlakuan terhadap bilangan pecahan dalam Prolog bervariasi dari implementasi ke implementasi; bilangan pecahan tidak terlalu banyak digunakan dalam program Prolog, karena penekanan dalam Prolog adalah pada manipulasi simbol. Prolog menyediakan semua predikat umum pada angka, seperti `>/2`, `</2`, `=</2`, `>=/2`, dll., dan operasi aritmatika, seperti `-/1`, `+/1`, `+/2`, `-/2`, `*/2`, `//2`, `mod/2`, `rem/2`, `gcd/2`, `lcm/2`, `abs/1`, `sign/1`, `max/2`, `min/2`, `floor/1`, `ceiling/1`, `truncate/1`, `sqrt/1`, `sin/1`, `cos/1`, dll. Anda menggunakan `==/2` untuk memeriksa apakah dua angka sama dan `=\=/2` untuk tidak sama. Simbol `=/2` dicadangkan untuk unifikasi: `s = t` mengembalikan `true` jika `s` dan `t` dapat diunifikasi.

Prolog juga menyediakan predikat `number/1` untuk memeriksa apakah suatu istilah adalah angka atau bukan; `integer/1` untuk bilangan bulat; dan `float/1` untuk bilangan float dengan `"`. Misalnya, `float(1.0)` mengembalikan `true`; `float(1)` mengembalikan `false`. Perhatikan bahwa `integer(1+1)` mengembalikan `false` karena `1+1` adalah suku majemuk. Jika

kita memeriksa nilai $1+1<3$, Prolog akan mengembalikan true karena $1 + 1$ akan dievaluasi menjadi 2 sebelum $<$ dipanggil pada (2, 3).

Variabel dilambangkan dengan pengidentifikasi yang terdiri dari huruf, angka, dan karakter garis bawah dan diawali dengan huruf besar atau garis bawah. Garis bawah saja, “_”, menunjukkan variabel yang berbeda tanpa nama. Variabel tanpa nama digunakan ketika kita tidak peduli dengan nilai variabel tersebut. Misalnya, jika kita mendefinisikan relasi perkalian $\text{mul}(X, Y, Z)$, di mana $Z = X * Y$, aturan pertama dapat berupa $\text{mul}(0, \_, 0)$, di mana posisi kedua adalah variabel tanpa nama.

Variabel Prolog sangat mirip dengan variabel dalam logika karena dapat diinstansiasi oleh istilah sembarang melalui unifikasi. Jika suatu variabel diinstansiasi, kita mengatakan “variabel tersebut dibatasi.” Beberapa operasi memerlukan variabel yang terlibat untuk dibatasi. Sebagai contoh, untuk menunjukkan

$\text{descen}(X, Z) \text{ :- parent}(Y, X), \text{descen}(Y, Z).$

adalah aturan terminasi dalam Contoh 8.1.5, kita perlu memeriksa ikatan X , Y , dan Z . Predikat $\text{parent}(Y, X)$ memastikan bahwa X dan Y akan terikat pada atom ketika kita mencapai $\text{descen}(Y, Z)$. Ketika variabel-variabel ini terikat oleh atom, atom yang terikat pada Y satu langkah lebih dekat daripada atom X ke Z dalam relasi turunan, artinya ini mendefinisikan urutan yang berdasar $>$ sedemikian rupa sehingga $X > Y$ ketika kita membandingkan $\text{descen}(X, Z)$ dan $\text{descen}(Y, Z)$.

Dalam Prolog, terdapat predikat infiks khusus yang disebut $\text{is}/2$, yang digunakan dalam contoh Hanoi. Predikat ini menyerupai pernyataan penugasan variabel dengan ekspresi numerik dalam bahasa pemrograman konvensional. Misalnya, nilai X adalah $1+1$ bernilai benar jika X diinisialisasi dengan 2. Artinya, Prolog akan mengevaluasi sisi kanan dari is , jika itu adalah angka, angka tersebut akan terikat pada variabel di sisi kiri is ; jika tidak, akan dihentikan dengan kesalahan. Sebaliknya, $X = 1+1$ juga akan mengembalikan nilai benar, tetapi akan mengikat variabel X ke istilah gabungan $1+1$. Sebagai contoh lain, “ $X = a$ ” akan mengembalikan nilai benar dan “ X adalah a ” akan dihentikan, karena a bukan angka.

Istilah Gabungan dan Daftar

Suatu istilah gabungan terdiri dari predikat atau simbol fungsi, yang disebut “functor” dan sejumlah “argumen,” yang merupakan istilah. Istilah majemuk biasanya ditulis sebagai sebuah funktor diikuti oleh daftar istilah argumen yang dipisahkan koma dan dikelilingi oleh tanda kurung. Jumlah argumen disebut aritas istilah tersebut.

Terdapat predikat bawaan yang disebut $\text{functor}/3$ yang memungkinkan kita untuk mengekstrak funktor dan aritasnya dari sebuah istilah majemuk:

?- functor(f(a, b), F, N).
F = f, N = 2

Istilah gabungan sangat penting untuk merepresentasikan berbagai struktur data dari bahasa konvensional. Misalnya, untuk merepresentasikan pohon biner di mana setiap simpul pohon

yang tidak kosong berisi kunci, kita dapat menggunakan istilah gabungan `node(key, leftChild, rightChild)`. Kita dapat dengan mudah mendefinisikan predikat `tree/1`, yang memeriksa apakah argumennya adalah pohon, dan predikat `dfs/2`, yang mencoba menemukan kunci dengan pencarian mendalam (depth-first search) dalam sebuah pohon.

```
tree(nil).                % nil denotes the empty tree
tree(node(_, Left, Right)) :- tree(Left), tree(Right).
dfs(node(K, _, _), K) :- write('found '), write(K).
dfs(node(_, L, _), K) :- dfs(L, K).          % go to Left branch
dfs(node(_, _, R), K) :- dfs(R, K).          % go to Right branch
```

Dalam definisi DFS, kita menggunakan “_” dua kali dalam setiap aturan; setiap “_” mewakili variabel unik.

Salah satu anggota khusus dari istilah majemuk adalah daftar, yang merupakan kumpulan istilah yang terurut. Ini dilambangkan dengan tanda kurung siku dengan istilah-istilah yang dipisahkan oleh koma atau dalam kasus daftar kosong, dengan `[]`. Berikut adalah beberapa daftar tiga elemen, `[1,2,3]`, `[red,green,blue]`, atau `[34,tom,[2,3]]`. Daftar adalah pengganti array dari bahasa pemrograman konvensional.

Untuk daftar yang tidak kosong, anggota pertama dari daftar dipisahkan dari anggota lainnya oleh simbol `|`. Misalnya, hasil dari `[X | Y] = [1,2,3]` adalah `X = 1` dan `Y = [2, 3]` dan hasil dari `[X | Y] = [1]` adalah `X = 1` dan `Y = []`. Dua daftar dapat disatukan jika panjangnya sama, dan semua elemennya dapat disatukan secara berpasangan.

<code>?-[a,b,c,d]=[Head Tail].</code>	<code>?- [(a+X),(Y+b)]=[(W+c),(d+b)].</code>
<code>Head = a,</code>	<code>W = a,</code>
<code>Tail = [b,c,d]?</code>	<code>X = c,</code>
<code>yes</code>	<code>Y = d?</code>
	<code>yes</code>
<code>?- [a,b,c,d]=[X,Y Z].</code>	<code>?- [(a+X),(Y+b)]=[(W+c),(X+Y)].</code>
<code>X = a,</code>	<code>no</code>
<code>Y = b,</code>	
<code>Z = [c,d]?</code>	
<code>yes</code>	

Predikat Populer pada Daftar

Sebagian besar implementasi Prolog menyediakan implementasi yang efisien untuk predikat-predikat berikut pada daftar. Kami juga menyediakan kode Prolog untuk predikat-predikat ini; kode-kode tersebut bukanlah implementasi sebenarnya.

- `list/1`: Memeriksa apakah argumennya adalah sebuah daftar.

```
list([]).
list(_ | X) :- list(X).
```

- `member/2`: Memeriksa apakah argumen pertama merupakan anggota dari argumen kedua.

```
member(X, [X | _]).
member(X, [_ | Y]) :- member(X, Y).
```

- `append/3`: Memeriksa apakah daftar ketiga merupakan gabungan dari dua daftar pertama.

```
append([], Y, Y).
append([F|X], Y, [F|Z]) :- append(X, Y, Z).
```

- `select/3`: `select(E, List1, List2)` bernilai benar jika `List2` adalah hasil penghapusan `E` dari `List1`.

```
select(E, [E|Y], Y).
select(E, [F|Y], [F|Z]) :- select(E, Y, Z)
```

Misalnya, `select(E, [a,b], Z)` memiliki dua solusi: (1) `E = a` dan `Z = [b]`; (2) `E = b` dan `Z = [a]`. Implementasi alternatif dari `select` adalah dengan menggunakan `append`:

```
select(E, L, R) :- append(X, [E|Y], L), append(X, Y, R).
```

- `delete/3`: `delete(E, List1, List2)` bernilai benar ketika `List2` adalah hasil penghapusan semua anggota `List1` yang menyatu dengan `E`.

```
delete(_, [], []).
delete(E, [E|Y], Z) :- delete(E, Y, Z).
delete(E, [F|Y], [F|Z]) :- E \= F, delete(E, Y, Z).
```

di mana `\=` berarti “tidak dapat disatukan.” Misalnya, `delete(E, [a,b], Z)` hanya memiliki satu solusi: `E = a`, `Z = [b]`; jawaban kedua, `E = b`, `Z=[a]`, tidak dapat dihasilkan oleh kode di atas, karena Prolog memperlakukan `E \= a` sebagai false (yaitu, `E = a` adalah true). Ini menunjukkan perbedaan lain antara `select` dan `delete`.

- `reverse/2`: `reverse(List1, List2)` bernilai true ketika elemen-elemen `List2` berada dalam urutan terbalik dibandingkan dengan `List1`.

```
reverse([], []).
reverse([F|X], Z) :- reverse(X, Y), append(Y, [F], Z).
```

- `permutasi/2`: `permutasi(List1, List2)` bernilai benar jika `List2` merupakan permutasi dari `List1`.

```
permutation([], []).
permutation([F|X], Y) :- permutation(X, Z), select(F, Y, Z).
```

- `length/2`: `length(List, N)` bernilai benar jika panjang `List` adalah `N`.

```
length([], 0).
length([_|X], N) :- length(X, M), N is M+1.
```

Salah satu fitur umum dari predikat-predikat ini adalah bahwa semuanya didefinisikan secara rekursif. Penghentian definisi ini mudah ditetapkan ketika salah satu argumen untuk panggilan rekursif menjadi lebih kecil, atau kita dapat menunjukkan bahwa kepala setiap aturan secara ketat lebih besar daripada badannya di bawah urutan penyederhanaan.

Karena sifat relasional dari banyak predikat bawaan, predikat tersebut dapat digunakan dalam beberapa arah. Misalnya, `append/3` dapat digunakan untuk menggabungkan dua daftar menjadi satu (dua argumen pertama adalah input dan argumen ketiga adalah output) atau untuk membagi daftar yang diberikan menjadi dua bagian (argumen ketiga adalah input, dan dua argumen pertama adalah output). Misalnya, kueri “`append(X, Y, [1,2,3])`.” akan menghasilkan empat jawaban berikut:

```
X = [], Y = [1,2,3]
X = [1], Y = [2,3]
X = [1,2], Y = [3]
X = [1,2,3], Y = []
```

Oleh karena itu, sejumlah predikat pustaka yang relatif kecil sudah cukup untuk banyak program Prolog. Perhatikan bahwa implementasi yang disarankan untuk `length` dapat menghitung jawaban `N = 3` dari `length([a,b,c], N)` tetapi tidak dapat menghasilkan daftar 3 placeholder, `L = [_1, _2, _3]`, dari `length(L, 3)`. Beberapa versi Prolog, misalnya, SWI Prolog, dapat melakukannya.

Untuk beberapa aplikasi, kita perlu mendapatkan himpunan $S = \{x \mid A(x)\}$, yaitu, S adalah himpunan elemen yang memenuhi kondisi $A(x)$. Tidak mudah untuk menulis program Prolog yang menghasilkan himpunan seperti itu. Untungnya, di sebagian besar implementasi Prolog, himpunan seperti itu dapat diperoleh dengan memanggil `bagof(X, Goal(X), S)`. Misalnya, dalam Contoh 8.1.1, kita dapat mendefinisikan $\text{children}(F) = \{C \mid \text{papa}(F, C)\}$. Sebuah kueri seperti berikut akan memberikan jawaban yang memuaskan:

```
?- bagof(C, papa(Father, C), Children).
Father = bob, Children = [kathleen,joseph,mary] ? ;
Father = joe, Children = [john,bob,ted] ? ;
Father = john, Children = [caroline] ? ;
```


Father = ted, Children = [patrick]

Beberapa implementasi Prolog juga menyediakan operasi himpunan, seperti intersection/3, union/3, subtract/3, subset/2, ketika sebuah daftar dipandang sebagai himpunan.

Dalam Prolog, string adalah urutan karakter yang dikelilingi oleh tanda kutip ganda dan setara dengan daftar kode karakter numerik. Salah satu kesalahan umum adalah menggunakan string dalam predikat write alih-alih urutan karakter yang diberi tanda kutip tunggal (sebuah atom).

Prolog juga menyediakan beberapa alat debugging. Misalnya, untuk melacak eksekusi, Anda dapat menggunakan trace untuk mengaktifkan pelacakan dan notrace untuk menonaktifkannya.

Algoritma Pengurutan dalam Prolog

Untuk mengilustrasikan bagaimana suatu algoritma diimplementasikan dalam Prolog, berikut adalah kode Prolog untuk pengurutan penyisipan (insertion sort) dan pengurutan cepat (quicksort).

Contoh 8.2.1 Program berikut mendefinisikan empat predikat untuk implementasi pengurutan penyisipan dan pengurutan cepat: isort/2, insert/3, split/4, dan qsort/2.

```
% isort(L, R): R is the output of L by insert sort
isort([], []).
isort([X|L], R) :-
    isort(L, T),           % sort L into T
    insert(X, T, R).       % insert X into T to obtain R.

% insert(X, L, R): insert X into sorted list L and R is the result.
insert(X, [], [X]).
insert(X, [Y|L], [X, Y|L]) :- X <= Y.
insert(X, [Y|L], [Y|R]) :- X > Y, insert(X, L, R).

% split(P, I, S, B): split the input list I into S and B by pivot P.
split(_, [], [], []).
split(P, [H|I], [H|S], B) :- H <= P, split(P, I, S, B).
split(P, [H|I], S, [H|B]) :- H > P, split(P, I, S, B).

% qsort(L, R): L is the input list and R is the output by quicksort
qsort([], []).
qsort([X|Y], Z) :- split(X, Y, S, B),
    qsort(S, S1), qsort(B, B1), append(S1, [X|B1], Z).
```

Algoritma pengurutan lainnya dapat didefinisikan dengan cara yang serupa.

Kode di atas jelas, rapi, dan mudah dipahami serta dipelihara. Ini adalah ciri khas bahasa pemrograman deklaratif. Tentu saja, eksekusinya tidak seefisien bahasa pemrograman konvensional. Itulah mengapa Prolog sebagian besar digunakan untuk pemrograman prototipe.

Untuk mengimbangi kelemahan ini, sebagian besar implementasi Prolog menyediakan implementasi bawaan yang efisien untuk fungsi-fungsi populer seperti operasi aritmatika atau pengurutan.

Fitur lain dari pemrograman deklaratif adalah kebenaran algoritma dapat diverifikasi. Untuk pengurutan, kebenarannya dipastikan oleh dua properti: (1) outputnya terurut; (2) outputnya merupakan permutasi dari daftar input. Dalam Prolog, permutasi adalah fungsi bawaan dan kita dapat mendefinisikan `sorted` dengan mudah.

Contoh 8.2.2 Program Prolog berikut mendefinisikan predikat `sorted(L)`, yang mengembalikan `true` jika dan hanya jika `L` adalah daftar yang terurut, dan beberapa predikat utilitas untuk tujuan verifikasi.

```
sorted([]).
sorted([_]).
sorted([X,Y|L]) :- X<Y, sorted([Y|L]).

verifyIsort(X, Y) :- isort(X, Y), permutation(X, Y), sorted(Y).
verifyQsort(X, Y) :- qsort(X, Y), permutation(X, Y), sorted(Y).
slowsort(X, Y) :- permutation(X, Y), sorted(Y).
```

Pada kode di atas, tiga klausa pertama mendefinisikan `sorted/1`. Dua klausa berikutnya mengilustrasikan verifikasi `isort` dan `qsort`. Klausa terakhir menunjukkan bagaimana persyaratannya dapat berupa kode itu sendiri. Kode `permutation/2` dapat ditemukan di Bagian sebelumnya.

Sebagian besar implementasi Prolog menyediakan dua versi predikat untuk pengurutan: `sort/2`, yang menghapus elemen duplikat, dan `msort/2`, yang tidak menghapus duplikat. Misalnya, kueri “`sort([b,a,c,a], X)`.” akan menghasilkan `X = [a,b,c]`, dan “`msort([b,a,c,a], X)`.” akan menghasilkan `X = [a,a,b,c]`. Predikat bawaan ini berjalan lebih cepat daripada algoritma pengurutan yang ditentukan pengguna.

8.3 REKURSI DALAM PROLOG

Dalam sebagian besar tugas pemrograman, kita sering perlu berulang kali melakukan beberapa operasi baik pada seluruh koleksi data atau hingga titik tertentu tercapai. Bahasa pemrograman konvensional menyediakan berbagai kontrol perulangan untuk melakukan tugas-tugas tersebut. Satu-satunya cara kita dapat melakukan ini di Prolog adalah dengan rekursi. Pada Bagian 8.2, kita telah melihat banyak program Prolog yang berisi definisi rekursif. Rekursi memungkinkan kita untuk menulis kode yang jelas dan elegan.

Data dalam rumus atau daftar tersebut sering didefinisikan secara rekursif dan paling baik diproses secara rekursif. Rekursi memungkinkan Prolog untuk melakukan pencarian kompleks dalam ruang masalah tanpa algoritma khusus. Tanpa rekursi, Prolog tidak akan bisa berbuat apa-apa. Untuk menguasai Prolog, kita harus belajar bagaimana menggunakan rekursi dengan benar dan efisien.

Setiap definisi rekursif, baik di Prolog atau bahasa lain, membutuhkan dua hal: kasus dasar dan kasus rekursif. Kasus dasar menentukan kondisi kapan rekursi berakhir. Tanpa ini, rekursi tidak akan pernah berhenti! Misalnya, kasus dasar dari `append(X, Y, Z)` adalah `append([], Y, Y)` ketika `X=[]`. Dalam program Prolog, klausa untuk kasus dasar hampir selalu mendahului klausa untuk kasus rekursif, yang meminta program untuk menangani masalah serupa dengan ukuran yang lebih kecil. Misalnya, `append([U|X], Y, [U|Z]) :- append(X, Y, Z).`

Penghentian Program

Bahaya rekursi adalah program dapat berulang tanpa henti dan tidak pernah berhenti. Masalah penghentian lebih rumit dalam Prolog daripada dalam program konvensional karena input ke program Prolog dapat berisi variabel, dan kita dapat meminta beberapa solusi dari satu kueri.

Definisi 8.3.1 Suatu program Prolog P dikatakan memiliki penghentian pertama pada input x jika P berhenti tanpa solusi atau solusi pertama pada x . P dikatakan memiliki penghentian terakhir pada x jika P berhenti setelah semua solusi x dihasilkan. Jika P memiliki penghentian pertama (terakhir) pada input dasar x mana pun, maka P dikatakan memiliki penghentian dasar pertama (terakhir).

Contoh 8.3.2 Diberikan program P berikut, definisi `ancestor(X, Y)` adalah “ X adalah leluhur dari Y .”

```
ancestor(tom, jack).
ancestor(X, Y) :- ancestor(X, Z), ancestor(Z, Y).
```

P memiliki terminasi pertama pada kueri “?- ancestor(A, B)” dengan solusi $A=\text{tom}$, $B=\text{jack}$, tetapi tidak pada ?- ancestor(jerry, tom), karena klausa kedua akan digunakan selamanya selama pencarian. Karena kueri kedua adalah ground, P tidak memiliki terminasi ground pertama. Karena terminasi terakhir menyiratkan terminasi pertama, P tidak memiliki terminasi ground terakhir.

Prolog adalah bahasa pemrograman relasional, dan relasi dapat berlaku antara beberapa entitas yang dilaporkan pada backtracking. Oleh karena itu, terminasi pertama tidak sepenuhnya mencirikan perilaku prosedural kueri Prolog, dan kita membutuhkan konsep terminasi terakhir. Kita dapat menggunakan terminasi kueri “?- Q, false.” untuk memeriksa apakah Q memiliki terminasi terakhir, karena semua solusi Q harus dihasilkan sebelum kueri berakhir.

Pertimbangkan program member/2 di Bagian 8.2, program tersebut memiliki terminasi ground ketika argumen kedua dari member adalah ground. Namun, kueri “?- member(X, Y), false.” akan gagal dengan kesalahan stack overflow, sehingga menunjukkan bahwa member(X, Y) tidak memiliki terminasi terakhir.

Contoh 8.3.3 Diberikan formula proposisional A dalam bentuk normal negasi (NNF, lihat Bagian 2.3), kita dapat menggunakan program berikut untuk memutuskan apakah A dapat dipenuhi, dengan asumsi formula tersebut menggunakan variabel Prolog untuk variabel proposisional, dan, atau, bukan untuk $\wedge, \vee, \neg$, masing-masing.

```
sat(true).                % positive literal
sat(not(false)).          % negative literal
sat(or(X, _)) :- sat(X).
sat(or(_, Y)) :- sat(Y).
sat(and(X, Y)) :- sat(X), sat(Y).
```

Dua klausa pertama adalah kasus dasar dari sat yang menangani literal, dan tiga klausa berikutnya adalah kasus rekursif yang menangani or dan and.

Mudah untuk melihat bahwa argumen untuk panggilan rekursif secara ketat lebih kecil daripada argumen di kepala setiap aturan. Dengan demikian, kita mengharapkan bahwa program akan berakhir dengan contoh uji. Sekarang, kita membuat tiga kueri untuk menguji program di atas:

```
test1(X, Y) :- sat(and(not(X), X)). % test examples
test2(X, Y) :- sat(and(X, not(Y))).
test3(X, Y) :- sat(and(X, or(not(X), Y)))
```

Kueri “?- test1(X, Y).” akan mengatakan tidak; dengan demikian, sat memiliki terminasi terakhir pada and(not(X), X). Untuk kueri kedua, “?- test2(X, Y).”, Prolog akan memberikan solusi pertama X=true, Y=false. Kueri ketiga “?- test3(X, Y).” akan menghasilkan solusi pertama X=true, Y=true. Dengan demikian, sat memiliki terminasi pertama pada dua kueri terakhir. Namun, jika kita mencari lebih banyak jawaban dari ?- test2(X, Y), kita akan melihat jawaban berikut:

```
X = not(false), Y = false
X = or(true,_), Y = false
X = or(not(false),_), Y = false
X = or(or(true,_),_), Y = false
X = or(or(not(false),_),_), Y = false
X = or(or(or(true,_),_),_), Y = false
X = or(or(or(not(false),_),_),_), Y = false
...
```

Jawaban-jawaban ini benar dalam Prolog, tetapi tidak benar untuk kepuasan proposisional (yang menanyakan model dari suatu formula). Jika Anda meminta lebih banyak jawaban dari `?- test3(X, Y)`, Anda juga akan melihat sejumlah jawaban yang tak terbatas. Dengan demikian, sat tidak memiliki terminasi terakhir pada `and(X, not(Y))` maupun pada `and(X, or(not(X), Y))`. Sangat mudah untuk memeriksa bahwa program tersebut memiliki terminasi dasar pertama dan terakhir dengan induksi pada struktur term dari input.

Masalah yang diilustrasikan oleh contoh di atas adalah bahwa ketika variabel yang muncul dalam argumen program rekursif tidak terbatas, variabel tersebut dapat digantikan oleh term yang berisi variabel tak terbatas lainnya, sehingga menciptakan sejumlah kemungkinan yang tak terbatas. Ini adalah alasan yang sama mengapa `member(X, Y)` tidak memiliki terminasi terakhir.

Rekursi Terfokus

Definisi predikat rekursif yang buruk dapat menyebabkan program berulang tanpa henti. Dalam Bab. 11, ditunjukkan bahwa memeriksa apakah program Prolog berakhir pada input tertentu adalah masalah yang tidak dapat diputuskan. Dengan demikian, orang-orang tertarik pada kondisi yang cukup untuk memastikan penghentian program Prolog. Rekursi terfokus adalah salah satu kondisi tersebut.

Definisi 8.3.4 Suatu predikat rekursif dikatakan terfokus pada input t_0 jika terdapat urutan yang berdasar $>$ sedemikian sehingga $t_1 > t_2 > \dots > t_n$, di mana t_i adalah nilai-nilai argumen yang sama yang muncul dalam panggilan rekursif berurutan dari predikat tersebut.

Argumen dalam definisi di atas disebut posisi terfokus dari predikat. Berdasarkan definisi `append/3` pada Bagian 8.2, predikat `append` terfokus pada argumen pertama ketika argumen pertama terikat pada daftar item (yang dapat berupa variabel) dengan panjang terbatas. Setiap panggilan rekursif mengurangi jumlah item dalam daftar sebanyak satu. Di sisi lain, jika argumen pertama terikat pada variabel, panggilan tersebut dapat berlangsung selamanya. Hal yang sama dapat dikatakan tentang `member/2` atau `sat` dalam Contoh 8.3.3, di mana satu rangkaian panggilan `sat` dalam kueri `"test2(X, Y)."` dapat berisi nilai-nilai berikut sebagai argumen dari `sat`:

`and(X, not(Y)), X, or(X1, _), X1, or(X2, _), X2, or(X3, _), X3, ...`

Contoh 8.3.5 Dalam Contoh 8.3.2, transitivitas relasi leluhur dinyatakan oleh aturan `ancestor(X, Y) :- ancestor(X, Z), ancestor(Z, Y)`. Aturan ini tidak dapat digunakan dalam Prolog karena menyebabkan loop tak terbatas. Untuk mendefinisikan `ancestor(X, Y)` sebagai " X adalah leluhur dari Y ," kita dapat menggunakan relasi `parent/2`, yang didefinisikan oleh sekumpulan fakta dasar, dan dapat direpresentasikan oleh grafik asiklik terarah (DAG) di antara individu. Kemudian, kita dapat menunjukkan bahwa

ancestor/2 yang didefinisikan oleh aturan berikut adalah rekursi terfokus pada argumen pertama.

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Misalkan $X_0, X_1, \dots, X_n$ adalah nilai argumen pertama dalam panggilan rekursif leluhur. Dengan asumsi bahwa induk mendefinisikan DAG di antara individu, kita dapat melihat bahwa jarak dari X_i ke Y dalam DAG berkurang satu ketika i bertambah satu. Dengan demikian, panjang jalur terpanjang dari DAG memberikan batas atas untuk jumlah panggilan rekursif leluhur. Mudah untuk melihat bahwa relasi leluhur adalah penutupan transitif dari relasi induk.

Proposisi 8.3.6 Jika predikat rekursif P difokuskan pada input dasar, maka P memiliki terminasi terakhir.

Pembuktiannya diserahkan sebagai latihan. Perhatikan bahwa argumen lain dari P dapat berisi variabel.

Rekursi Ekor

Dalam Prolog, karena kekuatan variabel logika, banyak predikat dapat ditulis secara alami dengan cara rekursif ekor, di mana rekursi terjadi pada posisi terakhir badan predikat. Misalnya, predikat `list`, `member`, `append`, `select`, dan `delete` adalah rekursif ekor seperti yang diberikan pada Bagian 8.2. Di sisi lain, `reverse`, `permutation`, dan `length` bukanlah rekursif ekor.

Dalam banyak kasus, rekursi ekor baik untuk kinerja. Untuk program konvensional, kompiler yang cerdas dapat mengubah rekursi ekor menjadi rekursi interaktif, sehingga menghemat waktu dan ruang penggunaan tumpukan untuk menyimpan lingkungan panggilan rekursif. Dalam Prolog, teknik yang sama dapat digunakan sehingga panggilan rekursif ekor berarti bahwa sistem Prolog dapat secara otomatis menggunakan kembali ruang yang dialokasikan dari lingkungan pada tumpukan lokal. Dalam kasus tipikal, ini secara terukur mengurangi konsumsi memori program Anda, dari $O(N)$ dalam jumlah panggilan rekursif menjadi $O(1)$. Karena penurunan konsumsi memori juga mengurangi tekanan pada alokasi memori dan pengumpulan sampah, menulis predikat rekursif ekor sering kali meningkatkan efisiensi ruang dan waktu program Prolog Anda.

Contoh 8.3.7 Predikat `reverse/2` yang didefinisikan di Bagian 8.2 disalin di sini:

```
reverse([], []).
reverse([F|X], Z) :- reverse(X, Y), append(Y, [F], Z).
```

Misalkan kuerinya adalah “?- reverse([a, b, c, d], L).” Daftar sub-tujuan melihat ?-reverse([], []), append([], [d], L3), append(L3, [c], L2), append(L2, [b],

L1), `append(L1, [a], L)`, sebelum menyusut. Untuk menghindari daftar sub-tujuan yang panjang ini, kita dapat memperkenalkan rekursi ekor `rev/3`:

```
reverse(X, Y) :- rev(X, [], Y).
rev([], R, R).
rev([U|X], Y, R) :- rev(X, [U|Y], R).
```

Sekarang, untuk kueri `?- reverse([a, b, c, d], L)`, daftar sub-tujuan tidak pernah berisi lebih dari satu item pada waktu tertentu:

```
?- reverse([a, b, c, d], L)
?- rev([a, b, c, d], [], L)
?- rev([b, c, d], [a], L)
?- rev([c, d], [b, a], L)
?- rev([d], [c, b, a], L)
?- rev([], [d, c, b, a], L)
L = [d, c, b, a].
```

Contoh ini mengilustrasikan bahwa rekursi ekor dapat menghemat ruang dan waktu. Namun, Anda tidak boleh terlalu menekannya, karena bagi pemula, lebih penting untuk memahami terminasi dan fokus pada deskripsi deklaratif yang jelas.

Program Prolog untuk Teka-Teki N-Ratu

Contoh 8.3.8 Di bawah ini adalah program Prolog sederhana untuk menghasilkan semua posisi yang sah untuk masalah N-ratu. Predikat `queen(N, R)` mengambil `N` sebagai jumlah ratu dan menghasilkan daftar `R` untuk nomor baris dari `N` ratu di `N` kolom.

```
% queen(N, R) generates at first a list R of N variables.
% Assuming one variable per column, then generate values for R,
% which gives the row number of the queen in each column.
queen(N, Res) :-
    length( Res, N),           % Res = N variables for columns
    gen_list( 1, N, L),        % L = [1, 2, ..., N]
    solution( N, Res, L).      % Res is a permutation of L

% gen_list( X, N, R) is true iff R = [X, X+1, X+2, ..., N].
gen_list( X, N, []) :- X>N.
gen_list( X, N, [X | Res]) :- Y is X+1, gen_list( Y, N, Res).

% solution( N, RowValues, List) is true if
% RowValues represent a list of N non-attacking queens
solution( 0, [], _) :- % run out of columns
solution( C, [R|Others], L) :- % 1st queen at position [R,C]
```

```

C1 is C-1,
select( R, L, L2),           % pick row R for column C
solution( C1, Others, L2),   % find a partial solution
noattack( R/C, C1, Others).  % 1st queen doesn't attack Others

% noattack(R/C, N, List) is true if the queen at R/C, where R is the row
% number, C is the column number, does not attack the other queens at
% R_i/Ci, where List = [R1, R2, ..., RN], Ci = i for 1<= i <= N.
% Since R is different from any Ri and C is different from any Ci,
% we check only if positions R/C and Ri/Ci share a diagonal.
noattack( _, _, []).        % Nothing to attack
noattack( R/C, C1, [R1 | Others]) :-
    C1-C =\= R1-R,           % R/C and R1/C1 on different diagonals
    C1-C =\= R-R1,           % R/C and R1/C1 on different diagonals
    C2 is C1-1,              % recursive on C2 = C1-1
    noattack( R/C, C2, Others). % no attacks for Others

```

Diberikan kueri “?- queen(4, R).”, jawaban pertama adalah $R = [2,4,1,3]$, yang menyatakan bahwa ratu ditempatkan di baris 2 pada kolom pertama, baris 4 pada kolom kedua, dan seterusnya, dan jawaban kedua adalah $R = [3,1,4,2]$. Dalam program di atas, setiap posisi dihasilkan dan diperiksa terhadap posisi yang dihasilkan sebelumnya, sehingga tidak ada dua posisi yang dapat saling menyerang dengan pergerakan ratu. Menggunakan pencarian mendalam (depth-first search) Prolog, program ini mudah ditulis tetapi sangat tidak efisien. Ketika $N > 12$, akan membutuhkan waktu cukup lama untuk menemukan satu solusi, jauh lebih lambat daripada pemecah SAT. Teka-teki Sudoku juga dapat dipecahkan oleh Prolog menggunakan ide yang sama. Namun, teka-teki Sudoku tidak dapat dipecahkan secara efisien menggunakan pendekatan generate-and-test ini, karena ruang pencariannya sangat besar. Pemecah SAT adalah alat yang jauh lebih baik untuk teka-teki Sudoku.

8.4 DI LUAR KLAUSA DAN LOGIKA

Sebagai bahasa tujuan umum, Prolog menyediakan berbagai predikat bawaan untuk melakukan aktivitas rutin seperti input/output, menggunakan grafik dan media lain untuk berkomunikasi dengan pengguna. Predikat-predikat ini tidak diberi makna relasional dan hanya berguna untuk efek samping yang ditunjukkannya pada sistem. Misalnya, predikat `write/1` menampilkan sebuah istilah dan mengembalikan nilai `true` ketika tampilan berhasil. Selain jenis ekstensi ini, berbagai jenis ekstensi diusulkan, beberapa dimaksudkan untuk meningkatkan daya ekspresif, dan beberapa mempercepat eksekusi.

Beberapa ekstensi ini mengikuti logika orde pertama; beberapa melampaui logika orde pertama. Misalnya, karena pemeriksaan kemunculan (`occurs-check`) dalam algoritma unifikasi itu mahal, sebagian besar implementasi Prolog memilih untuk mengabaikan pemeriksaan kemunculan tersebut. Ini tidak menimbulkan masalah di sebagian besar aplikasi. Jika Anda mengalami masalah ini dan membutuhkan algoritma unifikasi yang baik, Anda dapat memanggil fungsi bawaan `unify_with_occurs_check`.


```
?- X = f(Y,Y).
```

```
X = f(Y,Y)
```

```
yes
```

```
?- X = f(X,Y).
```

```
cannot display cyclic term for X
```

```
?- unify_with_occurs_check(X, f(X,Y)).
```

```
no
```

Operator Potong “!”

Operasi potong, dalam Prolog, adalah predikat dengan aritas 0, ditulis sebagai `!`, yang selalu berhasil tetapi tidak dapat di-backtrack. Operator ini paling baik digunakan untuk mencegah backtrack yang tidak diinginkan, termasuk penemuan solusi tambahan oleh Prolog dan untuk menghindari komputasi yang tidak perlu.

Contoh 8.4.1 Dalam contoh yang dibuat-buat ini, operator potong ditempatkan pada aturan pertama:

```
a :- b, !, c.           % 1
a :- c.                 % 2
b :- d.                 % 3
b :- e.                 % 4
c :- d.                 % 5
c :- e.                 % 6
d :- write('d').        % 7
e :- write('e').        % 8
```

Untuk kueri `:- a.`, Prolog akan mencoba aturan 1 terlebih dahulu. Daftar tujuan sekarang adalah `[b, !, c]`. Untuk menyelesaikan `b`, Prolog akan mencoba aturan 3, mereduksi `b` menjadi `d`. Prolog akan berhasil dengan aturan 7. Sekarang, operator `cut` dilewatkan, dan Prolog akan menyelesaikan `c` dengan aturan 5 dan 7 dengan keberhasilan pertama.

Jika kita mencari solusi berikutnya, Prolog akan berhasil untuk kedua kalinya dengan menyelesaikan `c` dari aturan 1 dengan aturan 6 dan 8. Jika kita meminta Prolog untuk solusi lebih lanjut, Prolog akan melakukan backtrack ke operator `cut`. Namun, arti dari operator `cut` adalah “tidak lagi melakukan backtrack.” Dengan demikian, Prolog akan berhenti hanya dengan dua keberhasilan. Tanpa operator `cut`, Prolog seharusnya menemukan empat solusi lagi: dua solusi dari `b` dari aturan 1 dengan aturan 4 dan 5–8 dan dua solusi dari aturan 2, 5–8.

Contoh di atas mengilustrasikan penggunaan operator `cut`: ketika `!` dicapai oleh *engine*₁, aturan alternatif untuk kepala aturan ini, seperti aturan 2 untuk `a`, dieliminasi; Aturan alternatif untuk semua saudara kandung kiri dari `!`, seperti aturan 4 untuk `b`, juga dihilangkan. Saudara kandung kanan dari `!` tidak terpengaruh oleh `!`.

Beberapa programmer menyebut cut sebagai fasilitas kontrol yang kontroversial karena ditambahkan hanya untuk efisiensi dan bukan merupakan fungsi logika.

Green cut adalah penggunaan cut yang hanya meningkatkan efisiensi. Green cut digunakan untuk membuat program lebih efisien tanpa mengubah output program. Misalnya, dalam definisi predikat `split` dari `qsort`, kita dapat memasukkan cut setelah `H =< P` dari aturan kedua, yaitu, ketiga klausa menjadi sebagai berikut:

```
split(_, [], [], []).
split(P, [H|I], [H|S], B) :- H =< P, !, split(P, I, S, B).
split(P, [H|I], S, [H|B]) :- H > P, split(P, I, S, B).
```

Jika tujuan saat ini `G` adalah `split(3, [2, 4, 1], L, R)`, maka `G` menyatu dengan kepala dari dua aturan terakhir. Setelah sub-tujuan `H =< P`, yaitu `2 =< 3`, berhasil, klausa terakhir tidak akan pernah dicoba karena operator cut, bahkan jika `G` menyatu dengan kepala klausa ini, dan kita ingin melihat lebih banyak jawaban. Cut ini adalah cut hijau, karena kita tahu sub-tujuan `H > P`, yaitu `2 > 3`, akan gagal dan klausa terakhir tidak dapat diterapkan. Operator cut menghemat waktu kita untuk mencoba menyatukan `G` dengan kepala klausa terakhir. Dengan atau tanpa cut, program berjalan dengan cara yang sama seperti yang diinginkan.

Cut merah adalah penggunaan cut yang mengubah arti program jika cut tidak ada. Untuk contoh `split` di atas, jika `H =< P` gagal, maka `H > P` akan berhasil. Untuk menghemat biaya `H > P`, kita dapat menulis program `split` sebagai berikut:

```
split(_, [], [], []).
split(P, [H|I], [H|S], B) :- H =< P, !, split(P, I, S, B).
split(P, [H|I], S, [H|B]) :- split(P, I, S, B).
```

Program ini akan memberikan hasil yang sama dengan program sebelumnya untuk input apa pun. Namun, program ini akan berkinerja berbeda dari program berikut, di mana pemotongan dihilangkan:

```
split(_, [], [], []).
split(P, [H|I], [H|S], B) :- H =< P, split(P, I, S, B).
split(P, [H|I], S, [H|B]) :- split(P, I, S, B).
```

Jadi, potongan di atas adalah potongan merah. Potongan merah berpotensi menjadi jebakan bagi bug. Misalnya, jika urutan kedua aturan dibalik, potongan hijau akan bekerja dengan benar dan potongan merah akan menghasilkan hasil yang salah. Jika penghematannya signifikan saat menggunakan potongan merah, dokumentasi terperinci harus diberikan bersama kode. Potongan harus digunakan dengan hemat. Penempatan operator potongan yang tepat dan urutan aturan diperlukan untuk menentukan makna logisnya.

Negasi sebagai Kegagalan

Klausul non-Horn seperti $A \vee B \vee \neg C$ dapat direpresentasikan sebagai $\neg B \wedge C \rightarrow A$. Bisakah kita menuliskannya dalam Prolog sebagai $A: \neg B, C$? Jawabannya terletak pada bagaimana menyelesaikan sub-tujuan $\neg B$ dengan proses reduksi tujuan. Salah satu pendekatannya disebut "negasi sebagai kegagalan": coba selesaikan B . Jika B berhasil, maka $\neg B$ gagal; jika B gagal, maka $\neg B$ berhasil.

Dengan menggunakan operator cut bersama dengan predikat bawaan fail, kita dapat mendefinisikan jenis negasi ini, sehingga properti yang tidak dapat ditentukan oleh klausa Horn dapat ditentukan. Sebagai contoh, untuk menyatakan "Mary menyukai semua hewan kecuali reptil" dalam Prolog

```
likes(mary, X) :- reptile(X), !, fail.
likes(mary, X) :- animal(X).
```

Kita dapat menggunakan ide yang sama dari "cut fail" untuk mendefinisikan predikat not, yang mengambil sebuah istilah sebagai argumen, dan menggunakan predikat bawaan "call." not(G) akan "memanggil" istilah G, mengevaluasi G seolah-olah G adalah sebuah tujuan. Jika G berhasil, maka call(G) juga berhasil, dan not(G) akan gagal. Jika tidak, not(G) berhasil.

```
not(G) :- call(G), !, fail.
not(_).
```

Sebagian besar sistem Prolog memiliki predikat bawaan seperti "not". Misalnya, SWI-Prolog menyebutnya \+. Ingat, "not" tidak sama dengan $\neg$, karena didasarkan pada keberhasilan/kegagalan tujuan. Namun, hal ini bisa bermanfaat:

```
likes(mary, X) :- not(reptile(X)).    % Mary does not like reptiles.
different(X, Y) :- not(X = Y).        % Not equal means "different"
```

Negasi sebagai kegagalan dapat menyesatkan. Misalkan basis data tersebut menyimpan nama-nama anggota masyarakat, ditandai dengan apakah mereka tidak bersalah atau bersalah atas suatu pelanggaran, dan dinyatakan dalam Prolog:

```
innocent(peter_pan).
innocent(winnie_the_pooh).
innocent(X) :- occupation(X, nun).
guilty(joe_bloggs).
guilty(X) :- occupation(X, thief).

?- innocent(einstein).
no.
```

Jika kita menambahkan satu aturan lagi ke dalam program di atas

```
guilty(X) :- not(innocent(X)).
```

```
?- guilty(einstein).
yes.
```

Menunjukkan bahwa “Einstein tidak dapat dibuktikan tidak bersalah” adalah satu hal; tetapi menunjukkan secara keliru bahwa dia bersalah adalah hal yang sangat berbeda. Untuk membenarkan perilaku “negasi sebagai kegagalan,” orang-orang menyarankan asumsi dunia tertutup: setiap rumus atom dasar diasumsikan salah kecuali dapat dibuktikan benar oleh program. Di bawah asumsi ini, karena kita tidak dapat menunjukkan bahwa `innocent(einstein)` benar dalam program, maka `innocent(einstein)` diasumsikan salah, sehingga membenarkan jawaban `guilty(einstein)`.

Beberapa perilaku yang mengganggu bahkan lebih halus daripada masalah `innocent/guilty` dan dapat menyebabkan beberapa kesalahan pemrograman yang sangat tidak jelas. Berikut adalah program Prolog tentang restoran:

```
good_standard(godels).
good_standard(hilberts).
expensive(godels).
reasonable(R) :- not(expensive(R)).
```

```
?- good_standard(X), reasonable(X).
X = hilberts
yes
```

```
?- reasonable(X), good_standard(X).
no.
```

Secara logis, kueri “`good_standard(X), reasonable(X)`.” setara dengan kueri “`reasonable(X), good_standard(X)`.” Mengapa kita mendapatkan jawaban yang berbeda untuk kueri yang tampaknya setara secara logis? Perbedaan antara kedua kueri tersebut adalah sebagai berikut. Pada kueri pertama, variabel `X` selalu diinstansiasi ketika `reasonable(X)` dieksekusi. Pada kueri kedua, `X` tidak diinstansiasi ketika `reasonable(X)` dieksekusi. Semantik `reasonable(X)` berbeda tergantung pada apakah argumennya diinstansiasi atau tidak. Untuk menghindari masalah semacam ini, beberapa implementasi Prolog meminta agar `G` didasarkan pada `not(G)`.

Karena masalah yang terkait dengan “negasi sebagai kegagalan,” diperlukan kehati-hatian ekstra saat menggunakannya.

Di Luar Klausa

Biasanya, badan aturan terdiri dari rumus atomik, dipisahkan oleh koma, dan menunjukkan konjungsi sub-tujuan. Jika koma diganti dengan titik koma (`;/2`), maka itu akan menunjukkan disjungsi sub-tujuan. Aturan seperti `H :- (C; D), E` bukan lagi sebuah klausa:

aturan ini setara dengan dua klausa: $H :- C, E$ dan $H :- D, E$, karena “;” dianggap sebagai v. Konjungsi “,” dan disjungsi “;” hanya dapat muncul di badan, bukan di kepala aturan.

Prolog juga mendukung relasi if-then dengan menggunakan simbol infiks “ $\rightarrow$ /2”. Aturan seperti $H :- C \rightarrow G$ memiliki efek yang sama dengan $H :- C, G$: jika C mengembalikan nilai true, G akan dipanggil; jika C mengembalikan nilai false, $C \rightarrow G$ mengembalikan nilai false. Ketika $\rightarrow$ dikombinasikan dengan “;”, kita kemudian memiliki relasi if-then-else yang diimplementasikan sebagai $H :- C \rightarrow G1; G2$: Jika C mengembalikan nilai true, G1 akan dipanggil; jika tidak, G2 akan dipanggil. Secara logis, $H :- C \rightarrow G1; G2$, yang bukan merupakan klausa, setara dengan dua klausa: $H :- C, G1$ dan $H :- \neg C, G2$ (yang bukan merupakan klausa Horn).

Setelah sebuah predikat dideklarasikan sebagai dinamis, Prolog memungkinkan klausa-klausa dari predikat tersebut untuk disisipkan atau dihapus selama eksekusi. Misalnya pada Contoh 8.1.1, jika kita memiliki file yang berisi pasangan nama ayah-anak, kita dapat mendeklarasikan papa sebagai dinamis: `dynamic(papa)`. Selama eksekusi, kita membaca nama ke dalam Father dan Child dan memanggil `assertz(papa(Father, Child))`, yang menambahkan klausa `papa(Father, Child)` di akhir semua klausa yang ada dari predikat papa. Jika kita ingin menambahkan klausa baru di awal klausa yang ada, perintahnya adalah `assertz/1`. Untuk menghapus klausa selama eksekusi, perintahnya adalah `extract/1`. Kumpulan perintah ini memungkinkan pengguna untuk secara dinamis mengubah program yang terjadi sebagai hasil dari eksekusi `assertz/1` atau `retract/1`, yang menghapus klausa yang ditambahkan oleh `assertz/1`. Perubahan tersebut tidak memengaruhi aktivasi apa pun yang sedang dieksekusi. Dengan demikian, basis data dibekukan selama eksekusi suatu tujuan, dan daftar klausa yang mendefinisikan predikat tetap pada saat eksekusinya.

Variasi dan Perluasan

Sejak lahirnya Prolog, paradigma lain untuk pemrograman logika juga telah diperkenalkan. Pada bagian ini, kita akan secara singkat menjelaskan beberapa di antaranya.

Datalog

Datalog adalah bahasa pemrograman deklaratif. Awalnya, bahasa ini terutama dirancang sebagai bahasa kueri untuk basis data deduktif; tetapi dapat digunakan di domain lain (seperti analisis program dan pemrosesan big data). Secara sintaksis, Datalog merupakan subset dari Prolog dan menggunakan klausa Horn (fakta dan aturan) sebagai kode programnya. Misalnya, ia tidak memiliki operator cut seperti Prolog. Tidak seperti strategi resolusi terbatas depth-first Prolog, Datalog biasanya mengadopsi strategi evaluasi bottom-up, yang dimulai dengan fakta dalam program dan berulang kali menerapkan aturan untuk mendapatkan fakta baru.

Pemrograman Logika Batasan

Dalam Prolog, sub-tujuan dari badan aturan Prolog diselesaikan dari kiri ke kanan. Pemrograman logika batasan (CLP) memperluas Prolog dengan melonggarkan urutan kiri-kekanan ini untuk sub-tujuan tertentu. Kendala adalah relasi yang telah ditentukan sebelumnya pada variabel, misalnya, $X + Y > 8$, yang juga muncul dalam program Prolog. Evaluasi kendala

mirip dengan Prolog. Tidak seperti Prolog, ketika interpreter CLP menemukan kendala yang tidak dapat dievaluasi, ia akan menempatkan kendala tersebut dalam penyimpanan kendala. Kemudian, jika penyimpanan kendala ditemukan tidak dapat dipenuhi, interpreter akan melakukan penelusuran balik. Jika kendala tersebut dapat dipenuhi, solusi akan diberikan.

Pemrograman Himpunan Jawaban

Pemrograman himpunan jawaban (ASP) adalah jenis paradigma pemrograman deklaratif lain untuk menyelesaikan masalah pemenuhan kendala. Ini didasarkan pada semantik model stabil (himpunan jawaban) dari pemrograman logika. ASP mengikuti pendekatan pembangkitan model daripada pendekatan pembuktian teorema. Pemecah himpunan jawaban mencoba menghasilkan model stabil menggunakan prosedur pencarian penelusuran balik (ditingkatkan dengan teknik seperti mekanisme penggerak konflik dalam pemecah SAT modern). Dan solusi diberikan oleh model spesifikasi masalah.

8.5 LATIHAN SOAL

1. Diberikan program Prolog berikut


```
append([], Y, Y).
append([H | X], Y, [H | Z]) :- append(X, Y, Z).
```

 Temukan empat solusi untuk kueri “?- append(X, Y, [a, b, c]).” Untuk setiap dari empat solusi tersebut, temukan bukti resolusi yang sesuai untuk setiap solusi.
2. Diberikan program pada Contoh 8.1.2, silakan jawab pertanyaan berikut:
 - a. Apa hasil program ketika mengkonversi $\neg(p \rightarrow (q \vee \neg p))$ ke NNF? Harap tampilkan NNF Anda menggunakan format Prolog dan format konvensional.
 - b. Apakah Anda mendapatkan hasil yang sama jika Anda mengubah urutan klausa? Harap jelaskan semua kemungkinan kasus.
 - c. Apa bukti resolusi yang terkait dengan hasil Anda?
3. Untuk program pada Contoh 8.3.3, tempatkan klausa untuk menangani dan sebelum klausa untuk atau, dan jalankan kueri “?- test2(X, Y).” untuk tiga jawaban. Harap berikan bukti resolusi untuk masing-masing dari tiga jawaban tersebut.
4. Implementasikan dalam Prolog algoritma pengurutan seleksi, `ssort(X, Y)`, di mana X adalah daftar bilangan bulat dan Y adalah hasil pengurutan X. Anda perlu mengimplementasikan `maxNum(L, A)`, yang mengembalikan nilai true ketika A adalah bilangan maksimal di antara bilangan-bilangan dalam daftar L. Pengurutan seleksi Anda akan memindahkan bilangan maksimal secara berulang dari input ke output.
5. Implementasikan dalam Prolog algoritma pengurutan gabungan, `msort(X, Y)`, di mana X adalah daftar bilangan bulat dan Y adalah hasil pengurutan X.
6. Predikat `slowsort` didefinisikan dalam Contoh 8.2.2. Susun bukti resolusi untuk kueri “?- slowsort([3,2,1], Y).”
7. Asumsikan `xor/2` adalah operasi logika $\oplus$ (eksklusif-atau). Tambahkan aturan untuk mengkonversi formula yang mengandung `xor` ke NNF dalam program Prolog di Contoh 8.1.2.

8. Tambahkan predikat `cnf/2` ke dalam program Prolog di Contoh 8.1.2, di mana `cnf(F, X)` mengambil formula proposisional F sebagai input dan mengkonversi F menjadi formula X yang setara dalam CNF.
9. Tulis program Prolog yang mengkonversi formula orde pertama A ke NNF (bentuk normal negasi), dengan asumsi berikut: simbol predikat adalah $p/2$, $q/1$, dan $r/0$. Kuantifikasinya adalah $all(X, A)$ dan $ex(X, A)$, di mana X mengambil nilai dalam $\{x, y, z\}$.
10. Tentukan dengan bukti apakah program Prolog berikut ini bersifat terminasi atau tidak. Apakah mereka memiliki terminasi pertama? terminasi terakhir? terminasi dasar? pada input apa? dalam kondisi apa?
 - (a) `goHome(no_12).`
`goHome(X) :- get_next_house(X,Y), goHome(Y).`
 - (b) `hasFlu(rebecca).`
`hasFlu(john).`
`hasFlu(X) :- kisses(X, Y), hasFlu(Y).`
`kisses(janet,john).`
 - (c) `search(end).`
`search(X) :- path(X,Y), search(Y).`
11. Perkenalkan predikat rekursif ekor untuk masing-masing predikat berikut: (a) `panjang/2` dan (b) `permutasi/2`, sehingga hasil yang sama dapat diperoleh melalui panggilan rekursif ekor.
12. Diberikan program Prolog berikut yang memberikan ulasan tentang restoran


```
not(P) :- call(P), !, fail.
not(_).
good_standard(godels).
good_standard(hilberts).
expensive(godels).
reasonable(R) :- not(expensive(R)).
```

 Apakah kedua kueri berikut memberikan hasil yang sama? Mengapa?


```
?- good_standard(X), reasonable(X).
?- reasonable(X), good_standard(X).
```

BAB 9

LOGIKA HOARE

Salah satu tujuan penelitian terpenting dalam logika adalah untuk memperoleh cara penalaran tentang sistem komputer (baik itu CPU, chip kecil, fungsi pustaka, seluruh program, atau sistem operasi). Membuktikan sifat-sifat sistem komputer sangat penting. Sebuah studi tahun 2003 yang ditugaskan oleh Institut Standar dan Teknologi Nasional Departemen Perdagangan menemukan bahwa bug perangkat lunak merugikan perekonomian AS sebesar Rp. 595 Triliun setiap tahunnya.

Dalam sistem yang kritis terhadap keselamatan, bahkan kesalahan kecil pun dapat memicu bencana. Karena jumlah data yang harus ditangani dalam verifikasi sistem komputer biasanya sangat besar, dukungan komputer sangat diperlukan. Banyak pendekatan telah diusulkan untuk memverifikasi secara formal sifat-sifat sistem perangkat lunak dan perangkat keras. Untuk verifikasi formal perangkat lunak, logika Hoare adalah salah satu pendekatan yang paling berpengaruh untuk membuktikan kebenaran desain perangkat lunak dalam bahasa pemrograman populer. Dengan memanfaatkan metode verifikasi formal dalam pengembangan perangkat lunak, seseorang memperoleh jaminan bahwa produk perangkat lunak akan berfungsi dengan benar.

9.1 VERIFIKASI FORMAL SISTEM KOMPUTER

Verifikasi formal adalah proses pembuktian kebenaran sistem komputer (perangkat keras/perangkat lunak/algoritma) menggunakan metode matematika. Pada dasarnya, verifikasi formal digunakan untuk memastikan bahwa sistem komputer bekerja sesuai harapan. Untuk melakukannya, kita merumuskan sifat-sifat yang diharapkan secara matematis dan membuktikan atau menyangkal teorema-teorema ini dengan alat penalaran. Dengan secara formal menentukan dan membuktikan perilaku sistem yang diinginkan, kita sangat mengurangi kemungkinan terjadinya kesalahan logika dalam sistem tersebut.

Secara umum, verifikasi formal memakan waktu dan rumit secara teknis. Untuk sistem yang sangat penting bagi kehidupan, biaya verifikasi formal jauh lebih kecil daripada potensi bencana. Selain itu, verifikasi formal akan mengurangi biaya debugging dan pengujian. Misalnya, ketika kita menulis program yang beroperasi pada bilangan asli (oleh karena itu seharusnya bekerja untuk setiap bilangan asli), kita tidak memiliki kemampuan untuk menguji hasil yang benar untuk setiap bilangan asli. Verifikasi formal dapat membuktikan sifat yang diinginkan untuk setiap bilangan asli dengan alat matematika.

Verifikasi Program Imperatif

Bahasa pemrograman imperatif seperti C, C++, Java, atau Python terkait erat dengan model komputasi *random-access stored-program* (RASP) dan merupakan contoh dari apa yang disebut arsitektur von Neumann. Algoritma yang ditulis dalam bahasa imperatif disebut program imperatif, di mana sebuah program menjelaskan langkah-langkah selanjutnya yang

mengubah memori komputer. Banyak teknik verifikasi formal yang digunakan bersama untuk program imperatif dan program fungsional.

Kita akan menggunakan kata "keadaan" untuk menunjukkan nilai semua variabel dalam program imperatif. Sebuah perintah dalam program dapat mengubah nilai variabel, sehingga menyebabkan transisi keadaan. Misalnya, misalkan x memiliki nilai 1 sebelum perintah " $x := x + 1$," nilai x akan menjadi 2 setelah eksekusi $x := x + 1$.

Suatu keadaan dapat dengan mudah ditentukan oleh rumus orde pertama, yang disebut pernyataan, seperti " $x \doteq 1$," " $A[0] \doteq 2$," dll.

Mengeksekusi program imperatif memiliki efek mengubah keadaan saat ini. Untuk melakukan penalaran tentang program semacam itu, pertama-tama kita menetapkan keadaan awal dengan menentukan nilai awal variabel. Kemudian kita menjalankan program tersebut, dan ini mengubah keadaan awal menjadi keadaan akhir melalui serangkaian transisi keadaan. Kemudian kita mengambil nilai variabel dalam keadaan akhir untuk mendapatkan hasil yang diinginkan. Misalnya, untuk menghitung " $z := \exp(x, y)$," keadaan awal menentukan nilai x dan y , dan keadaan akhir menentukan nilai z yang memenuhi " $z \doteq x^y$ " untuk menilai bahwa program $\exp(x, y)$ benar.

Logika Hoare (juga dikenal sebagai logika Floyd-Hoare atau aturan Hoare) adalah sistem formal dengan seperangkat aturan logis untuk melakukan penalaran secara ketat tentang kebenaran program imperatif. Logika ini diusulkan pada tahun 1969 oleh ilmuwan komputer dan ahli logika Inggris Tony Hoare dan kemudian disempurnakan oleh Hoare dan peneliti lainnya. Gagasan aslinya berakar pada karya Robert Floyd, yang telah menerbitkan sistem serupa untuk diagram alir.

Logika Hoare adalah pendekatan aksiomatik untuk verifikasi formal, yang terdiri dari (1) bahasa logika orde pertama untuk membuat pernyataan tentang program dan (2) aturan untuk menetapkan transisi keadaan, yaitu, pernyataan apa yang berlaku setelah transisi keadaan. Dari asumsi pada input, kita membuat prasyarat program, dan dari persyaratan pada output, kita membuat pascakondisi program. Baik prasyarat maupun pascakondisi adalah pernyataan, yaitu, rumus logika orde pertama. Kemudian kita menggunakan berbagai metode, termasuk pembuktian teorema, untuk menunjukkan bahwa pascakondisi adalah konsekuensi logis dari rumus yang diturunkan dari prasyarat dan kode program dengan aturan logika Hoare.

Verifikasi Program Fungsional

Pada bab sebelumnya, kita telah menyebutkan bahwa fungsi rekursif umum dan kalkulus Lambda merupakan dasar bagi bahasa pemrograman fungsional saat ini, seperti Lisp, ML, Clojure, atau Haskell. Dalam paradigma imperatif, kita menentukan secara tepat bagaimana suatu algoritma harus dijalankan; dalam paradigma fungsional, kita menentukan seperti apa hasil dari algoritma tersebut.

Contoh 9.1.1 Perhatikan program imperatif berikut yang menghitung jumlah semua angka dalam array bilangan bulat. Setelah eksekusi program, kita mengharapkan bahwa " $s \doteq \text{sum}(n, A)$ " adalah benar.

$$s := 0$$

$$\text{sum}(0, A) \doteq 0$$

```

for  $i := 0$  to  $n - 1$  do       $sum(i + 1, A) \triangleq sum(i, A) + A[i]$ 
       $s := s + A[i]$ 
od

```

Definisi penjumlahan sebenarnya adalah program fungsional, yang menentukan hasil yang diharapkan dari program imperatif.

Dalam program imperatif, variabel dapat dilewatkan melalui parameter ke dalam suatu fungsi, dan secara umum ada dua mode: "*pass by value*" dan "*pass by reference*." Dalam mode "*pass by value*," nilai variabel disalin ke parameter, dan kemudian, objek yang disalin digunakan dalam fungsi. Dalam mode "*pass by reference*," variabel itu sendiri dilewatkan melalui parameter ke dalam fungsi, dan setiap perubahan pada parameter akan memengaruhi nilai variabel (di luar fungsi).

Oleh karena itu, penugasan di dalam fungsi dapat memodifikasi keadaan global. Modifikasi ini disebut "efek samping." Efek samping seringkali menciptakan hasil yang tidak terduga, serta bug yang sulit dideteksi. Karena fitur yang tidak aman ini, beberapa bahasa pemrograman, seperti Java, melarang "*call by reference*."

Artinya, Java melewatkan semuanya berdasarkan nilai, termasuk referensi, meskipun orang dapat mensimulasikan "*call by reference*" dengan objek kontainer Java seperti array atau collection.

Dalam program fungsional, variabel tidak ada. Parameter suatu fungsi tampak seperti variabel dan berfungsi sebagai tempat penampung nilai. Kita tidak dapat membuat variabel global yang memengaruhi cara kerja fungsi kita. Pemrograman fungsional bertujuan untuk membuat semua fungsi murni, kecuali fungsi I/O yang dapat dimodelkan menggunakan konstruksi khusus seperti monad, yaitu struktur yang menggabungkan fragmen program dan membungkus nilai kembaliannya dalam suatu tipe dengan komputasi tambahan.

Program fungsional murni tidak memiliki beban untuk mempertahankan keadaan global dan menyerupai fungsi matematika. Dengan demikian, berbagai alat matematika dapat digunakan untuk menalar tentang sifat-sifatnya. Misalnya, dalam Contoh 7.3.12, kita mendefinisikan algoritma pengurutan sisipan, *isort*, dengan serangkaian persamaan, yang dapat dianggap sebagai program fungsional. Kebenaran *isort* dapat direduksi menjadi dua teorema: (a) $sorted(isort(x))$ (output dari *isort*(*x*) adalah daftar yang diurutkan), dan (b) $permu(x, isort(x))$ (*isort*(*x*) adalah permutasi dari *x*).

Pembuktian teorema Boyer-Moore (NQTHM), yang dimulai pada tahun 1971, menggunakan Lisp sebagai logika kerja untuk mendefinisikan fungsi rekursif total. Selama beberapa dekade, NQTHM berevolusi secara signifikan dan mencapai kemajuan penting dalam spesifikasi dan verifikasi formal:

- (1976) sebuah kompiler ekspresi untuk mesin tumpukan (Boyer dan Moore)
- (1983) keterbalikan algoritma enkripsi RSA (Boyer dan Moore)
- (1985) mikroprosesor FM8501 (Warren Hunt)
- (1988) kernel sistem operasi (William Bevier)
- (1989) bahasa assembly Piton pada FM9001 (Boyer dan Moore)

- (1990) hukum resiprositas kuadratik Gauss (David Russinoff)
- (1992) Jenderal Bizantium dan sinkronisasi jam (Bevier dan Young)
- (1992) sebuah kompiler untuk bahasa NQTHM (A. Falta)
- (1992) mikroprosesor FM9001 32-bit (Hunt dan Brock)
- (1993) protokol komunikasi asinkron tanda bi-fase (Boyer dan Moore)
- (1993) sebuah kompiler kecil (Debora Weber-Wulff)
- (1993) Motorola MC68020 dan Berkeley C String Library (Yuan Yu)

Penerus NQTHM yang luar biasa adalah ACL2 (singkatan dari “A Computational Logic for Applicative Common Lisp”), yang dikembangkan oleh Matt Kaufmann dan J.S. Moore. Mereka menerima Penghargaan Sistem Perangkat Lunak ACM 2005 karena membangun ACL2. Seperti NQTHM, ACL2 adalah bahasa logika dan pemrograman di mana Anda dapat memodelkan sistem komputer, bersama dengan alat untuk membantu Anda membuktikan properti dari model tersebut.

Pekerjaan terbaru menggunakan Coq dan Isabelle untuk verifikasi kernel sistem operasi dan kompiler juga menggembirakan dan menjanjikan. Bagi orang yang ragu tentang kinerja bahasa fungsional, Anda dapat menggunakannya setidaknya sebagai bahasa logika di mana desain lain dapat diformalkan dan diverifikasi dengan mudah. Misalnya, verifikasi sistem operasi seperti sel4 umumnya tentang program C imperatif, menggunakan program fungsional untuk spesifikasi. Kemudian Anda menggunakan kompiler terverifikasi untuk menghasilkan kode yang efisien dan aman untuk menambah keamanan dan kebenaran.

9.2 TRIPLE HOARE

Ide sentral logika Hoare adalah triple Hoare, notasi yang pertama kali diusulkan oleh Hoare. Suatu postcondition menentukan perilaku yang diinginkan dari program. Dengan adanya postcondition, triple Hoare menentukan dengan tepat precondition apa yang diminta program sebelum eksekusi. Artinya, sebuah triple Hoare menjelaskan bagaimana eksekusi suatu bagian kode mengubah keadaan komputasi, kita menganggap precondition dan postcondition sebagai deskripsi/batasan keadaan. Sebuah triple Hoare berbentuk:

$$\{P\} C \{Q\}$$

di mana P dan Q adalah pernyataan, yang merupakan rumus dalam bahasa orde pertama, dan C adalah kode program. P adalah *prasyarat* dan Q adalah *pascakondisi*: dengan asumsi P benar sebelum eksekusi C , C menyelesaikan eksekusi, dan Q benar setelah eksekusi C , kita katakan “ $\{P\} C \{Q\}$ ” adalah benar.

Misalnya, $\{x \doteq 1\} x := x + 1 \{x \doteq 2\}$ adalah benar, karena nilai x berubah dari 1 menjadi 2 setelah eksekusi $x := x + 1$. Di sisi lain, $\{x \doteq 1\} x := x + 1 \{x \doteq 3\}$ adalah salah. Karena sebuah rangkap tiga dapat benar atau salah, kita menyebutnya pernyataan dalam pengertian umum.

Contoh 9.2.1 Perhatikan program dari Contoh 9.1.1, yang menghitung jumlah semua angka dalam array bilangan bulat dengan prasyarat dan pascakondisi.

```

{  $P$  :  $A$  is an array of  $n$  integers, and  $n \geq 0$  }
 $s := 0$ 
for  $i := 0$  to  $n - 1$  do
     $s := s + A[i]$ 
{  $Q$ :  $s \doteq \sum_{i=0}^{n-1} A[i]$  }

```

Sekilas, P dan Q bukanlah rumus yang kita lihat di bab-bab sebelumnya. Sebuah array tidak berbeda dengan sebuah list dalam logika. Dalam Prolog, kita telah melihat bahwa sebuah list adalah term majemuk. Dengan demikian, tidak aneh untuk merepresentasikan sebuah array dengan sebuah term dalam logika orde pertama, dan kita dapat memperlakukan array sebagai sebuah term dalam logika tersebut. Demikian pula, kita juga dapat memperlakukan $\sum_{i=0}^{n-1} A[i]$ sebagai term lain. Bahkan, kita dapat mendefinisikan fungsi $sum: N, Array \rightarrow N$ secara rekursif sebagai berikut:

$$sum(0, A) \doteq 0$$

$$sum(i + 1, A) \doteq sum(i, A) + A[i]$$

Maka, $\sum_{i=0}^{n-1} A[i]$ setara dengan jumlah term $sum(n, A)$.

Sekarang, misalkan kita dapat menentukan prasyarat dan pascakondisi dalam logika orde pertama, bagaimana kita membuktikan bahwa pascakondisi adalah teorema dari prasyarat dan program? Ini ditangani oleh aturan Hoare untuk setiap jenis perintah dalam bahasa pemrograman kecil. Dalam bahasa ini, untuk kesederhanaan, kita hanya mempertimbangkan tiga jenis perintah:

1. Penugasan, yaitu $x := E$, di mana x adalah variabel dan E adalah ekspresi seperti yang dipahami dalam bahasa pemrograman
2. Kondisional, yaitu **if** B **Then** S_1 **else** S_2 **fi**, di mana B adalah ekspresi Boolean, S_1 dan S_2 adalah urutan perintah
3. Perulangan while, yaitu **while** B **do** S **od**, di mana B adalah ekspresi Boolean dan S adalah urutan perintah

Perintah dipisahkan oleh titik koma. Kita mengasumsikan bahwa komputasi semua ekspresi, seperti E dan B di atas, tidak memiliki efek samping. Artinya, E atau B tidak dapat berisi penugasan seperti dalam bahasa C , dan komputasi E atau B tidak mengubah nilai variabel apa pun.

Bahasa ini kecil tetapi mencakup bagian penting dari sebagian besar bahasa pemrograman imperatif. Ada banyak variasi konstruksi loop dalam bahasa pemrograman imperatif: bentuk "while", yang menggunakan kondisi kelanjutan; bentuk "do-until", yang selalu mengeksekusi badan loop setidaknya sekali, menguji kondisi di akhir daripada saat masuk; dan bentuk "for" atau "do", yang berulang pada interval bilangan bulat atau struktur

data. Semuanya dapat diekspresikan dengan cara yang mudah sebagai loop while, yang akan kita andalkan sepanjang bab ini. Misalnya, loop for dapat diekspresikan oleh loop while. Loop for dalam Contoh 9.2.1 dapat digantikan oleh loop while yang setara.

<pre> s := 0 for i := 0 to n - 1 do s := s + A[i] od </pre>	<pre> s := 0; i := 0 while (i < n) do s := s + A[i]; i := i + 1 od </pre>

Aturan Hoare

Aturan Hoare mencakup tiga perintah dasar dalam bahasa pemrograman kecil, ditambah satu aturan untuk komposisi perintah dan satu aturan untuk implikasi logis.

Aturan Penugasan

Aturan penugasan menyatakan bahwa, setelah penugasan, setiap pernyataan yang sebelumnya benar untuk sisi kanan penugasan sekarang berlaku untuk variabel tersebut. Secara formal, misalkan P adalah pernyataan di mana variabel x bebas, maka

$$\frac{T}{\{P [x \sim E]\} x := E \{P\}}$$

di mana $P[x \sim E]$ menunjukkan pernyataan yang diperoleh dari P di mana setiap kemunculan x digantikan oleh ekspresi E .

Aturan penugasan berarti bahwa kebenaran P setelah $x := E$ setara dengan kebenaran $P[x \sim E]$. Jadi, jika $P[x \sim E]$ benar sebelum penugasan, menurut aturan penugasan, P akan benar setelah penugasan karena nilai x adalah E . Semua prasyarat yang tidak dimodifikasi oleh penugasan dapat diteruskan ke pascakondisi.

Contoh-contoh tripel yang valid meliputi $\{x \doteq 4\} y := x + 1 \{y \doteq 5\}$ dan $\{x \leq N - 1\} x := x + 1 \{x \leq N\}$. Pada contoh pertama, untuk menunjukkan bahwa $y \doteq 5$ adalah benar, aturan penugasan menghasilkan prasyarat $x + 1 \doteq 5$, yang setara dengan $x \doteq 4$ dari $y \doteq 5$. Pada contoh kedua, aturan penugasan menghasilkan prasyarat $x + 1 \leq N$, yang setara dengan $x \leq N - 1$, dari $x \leq N$.

Karena kita belum memberikan definisi formal dari “keadaan,” kita tidak dapat membuktikan secara ketat kebenaran aturan penugasan. Namun, kita dapat berargumen secara informal. Untuk menunjukkan bahwa $\{P [x \sim E]\} x := E \{P\}$ adalah benar, misalkan s adalah keadaan sebelum $x := E$ dan s' adalah keadaan setelahnya. Jadi, $s' = s[x \rightarrow E]$ (dengan asumsi E tidak memiliki efek samping). $P[x \sim E]$ berlaku di s jika dan hanya jika P berlaku di $s' = s[x \rightarrow E]$, karena

1. Setiap variabel, kecuali x , memiliki nilai yang sama di s dan s' .
2. $P[x \sim E]$ memiliki setiap x di P yang digantikan oleh E .
3. P memiliki setiap x yang dievaluasi menjadi E di $s' = s[x \rightarrow E]$.

Aplikasi utama dari aturan penugasan adalah untuk menemukan prasyarat, yaitu $P[x \sim E]$, dari $x := E$ dari pascakondisi yang diinginkan P , bukan ke depan. Berhati-hatilah untuk tidak mencoba melakukan ini "ke depan" dengan mengikuti cara berpikir yang salah ini: $\{P\} x := E \{P[x \sim E]\}$; aturan ini mengarah ke contoh seperti aturan yang salah yang tampak menarik pada pandangan pertama adalah $\{x \div 4\} x := 5 \{5 \div 4\}$. Contoh lain; Hal ini mengarah pada contoh-contoh yang tidak logis seperti $\{x \div 5\} x := x + 1 \{x \div 5 \wedge x \div x + 1\}$.

Meskipun postcondition P yang diberikan secara unik menentukan precondition $P[x \sim E]$ berdasarkan aturan penugasan, kebalikannya tidak benar. Artinya, postcondition yang berbeda tersedia untuk precondition yang sama dan kode yang sama. Misalnya,

- $\{0 \leq y \wedge y \leq 4\} x := y \{0 \leq x \wedge x \leq 4\}$
- $\{0 \leq y \wedge y \leq 4\} x := y \{0 \leq y \wedge x \leq 4\}$
- $\{0 \leq y \wedge y \leq 4\} x := y \{0 \leq y \wedge y \leq 4\}$

merupakan contoh valid dari aturan penugasan.

Aturan penugasan yang diusulkan oleh Hoare dapat diterapkan pada beberapa penugasan timbal balik. Misalnya, untuk menukar dua elemen array, beberapa bahasa pemrograman mendukung perintah $A[i], A[j] := A[j], A[i]$. Aturan penugasan untuk jenis perintah ini adalah

$$\frac{T}{\{P[x \sim E_1, y \sim E_2]\} x, y := E_1, E_2 \{P\}}$$

Aturan Implikasi

Aturan implikasi, juga disebut aturan konsekuensi, murni logis karena tidak melibatkan perintah bahasa pemrograman apa pun, tetapi berkaitan dengan relasi implikasi dari pernyataan:

$$\frac{P_1 \rightarrow P_2 \quad \{P_2\} S \{Q_1\} \quad Q_1 \rightarrow Q_2}{\{P_1\} S \{Q_2\}}$$

Aturan ini dapat dibagi menjadi dua aturan:

$$\frac{P_1 \rightarrow P_2 \quad \{P_2\} S \{Q\}}{\{P_1\} S \{Q\}} \quad \frac{\{P\} S \{Q_1\} \quad Q_1 \rightarrow Q_2}{\{P\} S \{Q_2\}}$$

Kita katakan formula A lebih lemah daripada formula B jika $B \models A$, tetapi tidak jika $A \models B$. Jika kita menganggap relasi implikasi $\models$ sebagai urutan $>$ pada formula, maka $\perp$ adalah yang terkuat (atau terbesar) dan T adalah yang terlemah (atau terkecil).

Definisi 9.2.2 Kita katakan P adalah prasyarat terlemah untuk program S dan pascakondisi Q jika (a) $\{P\}S\{Q\}$ benar; dan (b) untuk setiap prasyarat P' , jika $\{P'\}S\{Q\}$ benar, maka $P' \models P$.

Aturan implikasi memungkinkan kita untuk memperkuat prasyarat P_2 menjadi P_1 , dengan mengasumsikan lebih dari yang kita butuhkan, dan untuk melemahkan pascakondisi Q_1 menjadi Q_2 , dengan menyimpulkan kurang dari yang diizinkan. Misalnya, jika kita telah menunjukkan bahwa $\{n > 0\} \text{ sort}(A) \{n > 0 \wedge \text{sorted}(A)\}$, dan pernyataan targetnya adalah $\text{sorted}(A)$, karena $n > 0 \wedge \text{sorted}(A) \rightarrow \text{sorted}(A)$, aturan implikasi memungkinkan kita untuk mendapatkan $\{n > 0\} \text{ sort}(A) \{\text{sorted}(A)\}$.

Aturan Kondisional

Aturan kondisional menyatakan bahwa kondisi pasca Q yang umum untuk kedua cabang pernyataan if-then-else juga merupakan kondisi pasca dari keseluruhan pernyataan if-then-else:

$$\frac{\{B \wedge P\} S \{Q\} \quad \{\neg B \wedge P\} T \{Q\}}{\{P\} \text{ if } B \text{ then } S \text{ else } T \text{ fi } \{Q\}}$$

Pada bagian “maka” dan bagian “lainnya”, B dan $\neg B$ ditambahkan ke prasyarat P , masing-masing.

Contoh 9.2.3 Untuk menunjukkan rangkap tiga berikut

$$\{P : T\} \text{ if } x < 10 \text{ then } x := x \text{ else } x := 10 \text{ fi } \{Q : x \leq 10\},$$

Agar pernyataan tersebut benar berdasarkan aturan kondisional, kita perlu membuktikan dua pernyataan berikut:

- (1) $\{P_1 : T \wedge x < 10\} x := x \{Q : x \leq 10\}$,
- (2) $\{P_2 : T \wedge x \geq 10\} x := 10 \{Q : x \leq 10\}$.

Untuk membuktikan (1) dan (2), kita peroleh hal berikut dengan aturan penugasan:

- (1') $\{x \leq 10\} x := x \{Q : x \leq 10\}$,
- (2') $\{10 \leq 10\} x := 10 \{Q : x \leq 10\}$.

Sangat mudah untuk menunjukkan bahwa (1) dan (2) benar berdasarkan aturan implikasi yang didasarkan pada (1') dan (2'). Jika bagian “else” hilang, aturan kondisional menjadi

$$\frac{\{B \wedge P\} S \{Q\} \quad \neg B \wedge P \rightarrow Q}{\{P\} \text{ if } B \text{ then } S \text{ fi } \{Q\}}$$

Contoh 9.2.4 Untuk menunjukkan pernyataan berikut

$$\{P : x \doteq a \wedge y \doteq b\} \text{ if } x > y \text{ then } y := x \text{ fi } \{Q : y \doteq \max(a, b)\}$$

Jika benar, berdasarkan aturan kondisional, kita perlu membuktikan dua pernyataan berikut:

- (1) $\{P_1 : x \doteq a \wedge y \doteq b \wedge x > y\} y := x \{Q : y \doteq \max(a, b)\};$
- (2) $(x \doteq a \wedge y \doteq b \wedge \neg(x > y)) \rightarrow y \doteq \max(a, b).$

Berdasarkan aturan penugasan, kita memiliki

$$\{x \doteq \max(a, b)\} y := x \{Q : y \doteq \max(a, b)\}$$

Untuk membuktikan (1), dengan aturan implikasi, kita perlu menunjukkan $P_1 \rightarrow x \doteq \max(a, b)$, yang dapat disederhanakan dengan persilangan kesamaan menjadi $a > b \rightarrow a \doteq \max(a, b)$ adalah benar. (2) juga dapat disederhanakan dengan persilangan kesamaan menjadi $\neg(a > b) \rightarrow b = \max(a, b)$, yang juga benar dalam aritmatika.

Kita dapat menambahkan perintah lewati, yang tidak melakukan apa pun, sehingga mudah untuk menggunakan perintah jika-maka-lain untuk perintah jika-maka: **if B then S fi** becomes **if B then S else skip fi**.

Proposisi 9.2.5 Jika $\{P_1\} S \{Q\}$ dan $\{P_2\} T \{Q\}$ adalah benar dan P adalah $(B \rightarrow P_1) \wedge (\neg B \rightarrow P_2)$, maka $\{P\} \text{if } B \text{ then } S \text{ else } T \text{ fi } \{Q\}$ is true.

Bukti Mudah untuk menunjukkan bahwa $(P \wedge B) \rightarrow P_1$ dan $(P \wedge \neg B) \rightarrow P_2$. Dari $(P \wedge B) \rightarrow P_1$ dan $\{P_1\} S \{Q\}$, kita memiliki $\{P \wedge B\} S \{Q\}$ berdasarkan aturan implikasi. Demikian pula, kita memiliki $\{P \wedge \neg B\} T \{Q\}$. Hasil akhirnya mengikuti dari aturan kondisional.

Proposisi di atas memungkinkan untuk merancang aturan kondisional baru:

$$\frac{\frac{\{P_1\} S \{Q\} \quad \{P_2\} T \{Q\}}{\{(B \rightarrow P_1) \wedge (\neg B \rightarrow P_2)\} \text{if } B \text{ then } S \text{ else } T \text{ fi } \{Q\}}}$$

Aturan ini berguna ketika kita ingin menyimpulkan prasyarat dari pascakondisi.

Aturan Iterasi

$$\frac{\{I \wedge B\} S \{I\}}{\{I\} \text{while } B \text{ do } S \text{ od } \{\neg B \wedge I\}}$$

Di sini, I adalah apa yang disebut invarian loop, yang bernilai benar di awal loop dan dipertahankan oleh badan loop S , yaitu, setelah loop selesai, I masih berlaku. Loop akan berakhir jika B menjadi salah. Pernyataan $\neg B$, yang disebut kondisi keluar loop, ditambahkan ke I sebagai kondisi pasca loop.

Contoh 9.2.6 Dengan asumsi x adalah bilangan bulat, untuk menunjukkan bahwa pernyataan berikut ini benar

$$\{I : x \leq 5\} \text{while } x < 5 \text{ do } x := x + 1 \text{ od } \{Q : x \doteq 5\}$$

kita menerapkan aturan iterasi dan menunjukkan bahwa

$$\{x \leq 5 \wedge x < 5\} x := x + 1 \{I : x \leq 5\}$$

Berdasarkan aturan penugasan, kita memiliki $\{x + 1 \leq 5\} x := x + 1 \{I : x \leq 5\}$, jadi kita hanya perlu membuktikan $(x \leq 5 \wedge x < 5) \rightarrow (x + 1 \leq 5)$ dengan aturan implikasi. Ketika x adalah bilangan bulat, $(x < 5) \rightarrow (x + 1 \leq 5)$ adalah benar. Jadi, $(x \leq 5 \wedge x < 5) \rightarrow (x + 1 \leq 5)$ adalah benar. Akhirnya, kita perlu menunjukkan $(I \wedge \neg B) \rightarrow Q$ dengan aturan implikasi. Karena $x \leq 5 \wedge \neg(x < 5) \rightarrow x = 5$ adalah benar untuk semua bilangan x , program tersebut benar ketika x adalah bilangan bulat.

Aturan iterasi dianggap sebagai yang paling sulit dalam logika Hoare karena kesulitan menemukan invarian loop yang baik. Kita akan membahas ini nanti di bab ini.

Aturan Urutan

Aturan penugasan, kondisional, dan iterasi mencakup tiga perintah dasar dari bahasa pemrograman kecil ini. Aturan urutan berlaku untuk program yang dieksekusi secara berurutan S dan T , di mana S dieksekusi sebelum T dan ditulis sebagai $S; T$:

$$\frac{\{P\} S \{Q\} \quad \{Q\} T \{R\}}{\{P\} S; T \{R\}}$$

di mana Q disebut pernyataan tengah. Misalnya, perhatikan dua contoh aksioma penugasan berikut: $\{x = 4\} y := x + 1 \{y = 5\}$ dan $\{y = 5\} z := y \{z = 5\}$. Dengan aturan urutan, dapat disimpulkan $\{x = 4\} y := x + 1; z := y \{z = 5\}$.

Contoh Verifikasi Formal

Sekarang, kita memiliki semua aturan yang dibutuhkan untuk menunjukkan kebenaran program apa pun dalam bahasa pemrograman kecil ini.

Contoh 9.2.7 Perhatikan program pada Contoh 9.1.1:

```

{ $n \geq 0$ }                                 $sum(0, A) = 0$ 
 $s := 0; i := 0$                              $sum(i + 1, A) = sum(i, A) + A[i]$ 
while ( $i < n$ ) do
     $s := s + A[i]; i := i + 1$ 
od
 $\{s = sum(n, A)\}$ 

```

Karena kode tersebut berisi perulangan while, langkah penting dalam menerapkan logika Hoare adalah menemukan invarian perulangan. Dengan memeriksa isi perulangan, kita melihat bahwa nilai s dan i diubah di dalam isi perulangan. Invarian perulangan yang baik harus mencerminkan perubahan ini. Karena variabel s mengakumulasi jumlah i elemen pertama dari A , tebakan pertama untuk invarian perulangan yang sesuai adalah $I: s = sum(i, A)$. Berdasarkan aturan urutan, kita perlu membuktikan pernyataan berikut:

$$(1) \{n > 0\} s := 0; i := 0 \{I\}$$

(2) $\{I\} \text{ while } i < n \text{ do } s := s + A[i]; i := i + 1 \text{ od } \{s \doteq \text{sum}(n, A)\}$

Pembuktian (1) mudah karena kita hanya perlu membuktikan $n > 0 \rightarrow I[i \sim 0, s \sim 0]$, yang berasal dari $\{I[i \sim 0, s \sim 0]\} s := 0 \{I[i \sim 0]\}$ dan $\{I[i \sim 0]\} i := 0 \{I\}$.

Karena $I[i \sim 0, s \sim 0]$ adalah $0 \doteq 0, A$, yang benar menurut definisi penjumlahan, maka (1) benar.

Pembuktian (2) direduksi menjadi dua pernyataan berikut dengan aturan iterasi dan aturan implikasi:

(2.1) $\{I \wedge i < n\} s := s + A[i]; i := i + 1 \{I\}$.

(2.2) $I \wedge \neg(i < n) \rightarrow \{s \doteq \text{sum}(n, A)\}$.

Cara membuktikan (2.1) sama dengan cara membuktikan (1), dan kita perlu membuktikan rumus berikut:

$$(2.1.1) (I \wedge i < n) \rightarrow I[i \sim i + 1, s \sim s + A[i]]$$

Membuka I di (2.1.1), kita memiliki

$$(s \doteq \text{sum}(i, A) \wedge i < n) \rightarrow (s + A[i] \doteq \text{sum}(i + 1, A))$$

Berdasarkan definisi jumlah (sum), $\text{sum}(i + 1, A) = \text{sum}(i, A) + A[i] = s + A[i]$, sehingga (2.1.1) benar; dengan demikian, (2.1) benar. Membuka I dalam (2.2), kita memiliki $(s \doteq \text{sum}(i, A) \wedge i \geq n) \rightarrow (s \doteq \text{sum}(n, A))$

Karena kondisi $i \geq n$ tidak dapat memberikan kita $i = n$, pembuktian tidak dapat dilanjutkan. Dari kegagalan tersebut, kita melihat bahwa kita perlu memperkuat invarian loop dengan menambahkan $i \leq n$ ke dalamnya: invarian loop baru I' seharusnya $s \doteq \text{sum}(i, A) \wedge i \leq n$. Dengan menggunakan I' , pembuktian (1) dan (2.1) akan berjalan seperti sebelumnya, dan pembuktian (2.2) menjadi mudah karena $(i \leq n \wedge i \geq n) \rightarrow i \doteq n$ adalah benar.

Contoh di atas menunjukkan bahwa invarian loop harus memuat semua informasi tentang cara kerja loop:

- Ini mencerminkan apa yang telah dilakukan sejauh ini bersama dengan apa yang masih harus dilakukan dan harus memuat semua variabel yang berubah.
- Ini benar baik di awal maupun di akhir setiap iterasi.
- Bersama dengan negasi dari kondisi loop, ini memberikan hasil yang diinginkan ketika loop berakhir.
- Jika suatu pembuktian tidak dapat berjalan dengan benar, kita mungkin perlu memperkuat invarian loop dengan menambahkan kondisi yang dibutuhkan dalam pembuktian.

Contoh 9.2.8 Perhatikan program berikut yang menemukan elemen maksimal dalam sebuah array:

$\{n > 0\}$	$arrMax(1, A) \doteq A[0]$
$m := A[0]; i := 1;$	$arrMax(i + 1, A) \doteq$
while $i < n$ do $A[i]$	$\quad max(arrMax(i, A),$
if $(m < A[i])$ then $m := A[i]$	$\quad max(x, y) \doteq x$ if $x > y$
$i := i + 1$	$max(x, y) \doteq y$ if $x \leq y$
od	
$\{m \doteq arrMax(n, A)\}$	

Dengan memeriksa isi loop, kita melihat bahwa nilai m dan i diubah di dalam loop. Sebuah invarian loop yang baik harus mencerminkan perubahan ini. Karena variabel m berisi nilai maksimal dari i elemen pertama dalam A , tebakan yang masuk akal untuk sebuah invarian loop adalah...

$$I : m \doteq arrMax(i, A) \wedge i \leq n.$$

Berdasarkan aturan barisan, kita perlu membuktikan pernyataan-pernyataan berikut:

- (1) $\{n > 0\} m := A[0]; i := 1 \{I\}$
- (2) $\{I\} \text{ while } i < n \text{ do } \dots \text{od } \{m \doteq arrMax(n, A)\}$

Pembuktian (1) mudah karena kita hanya perlu membuktikan

$$(1.1) : n > 0 \rightarrow I[i \sim 1, m \sim A[0]],$$

yang berasal dari $\{I[i \sim 1, m \sim A[0]]\} m := A[0] \{I[i \sim 1]\}$ dan $\{I[i \sim 1]\} i := 1 \{I\}$. Karena $I[i \sim 1, m \sim A[0]]$ adalah $A[0] \doteq arrMax(1, A) \wedge 1 \leq n$, yang benar menurut definisi $arrMax$ dan $n > 0$, maka (1) benar.

Pembuktian (2) direduksi menjadi dua pernyataan berikut dengan aturan iterasi dan aturan implikasi:

- (2.1) $\{I \wedge i < n\} \text{ if } (m < A[i]) \text{ then } m := A[i]; i := i + 1 \{I\}$
- (2.2) $I \wedge \neg(i < n) \rightarrow \{m \doteq arrMax(n, A)\}$

Pembuktian (2.1) difokuskan pada pembuktian pernyataan berikut:

$$(2.1.1) \{I \wedge i < n\} \text{ if } (m < A[i]) \text{ then } m := A[i] \{I[i \sim i + 1]\}$$

Dengan menerapkan aturan bersyarat, kita perlu membuktikan dua pernyataan berikut:

- (2.1.1.1) $\{I \wedge i < n \wedge m < A[i]\} m := A[i] \{I[i \sim i + 1]\}$
- (2.1.1.2) $(I \wedge i < n \wedge \neg(m < A[i])) \rightarrow I[i \sim i + 1]$

Dengan menerapkan aturan penugasan dan implikasi pada (2.1.1.1), kita perlu membuktikan

$$(2.1.1.1.1) (I \wedge i < n \wedge m < A[i]) \rightarrow I[i \sim i + 1, m \sim A[i]]$$

Membuka I di (2.1.1.1.1), kita perlu menunjukkan pernyataan berikut:

$$(m \doteq \text{arrMax}(i, A) \wedge i \leq n \wedge i < n \wedge m < A[i]) \\ \rightarrow (A[i] \doteq \text{arrMax}(i + 1, A) \wedge i + 1 \leq n)$$

Perhatikan bahwa $i + 1 \leq n$ adalah benar karena $i < n$ dan

$$\begin{aligned} \text{arrMax}(i + 1, A) &= \max(\text{arrMax}(i, A), A[i]) && \text{by the definition of arrMax} \\ &= \max(m, A[i]) && \text{by } m \doteq \text{arrMax}(i, A) \\ &= A[i] && \text{by } m < A[i] \end{aligned}$$

Oleh karena itu, (2.1.1.1.1) benar. Dengan menggunakan pendekatan yang sama, (2.1.1.2) dapat dibuktikan benar. Oleh karena itu, (2.1.1) benar.

Dengan membuka I pada (2.2), kita memiliki

$$(m \doteq \text{arrMax}(i, A) \wedge i \leq n \wedge \neg(i < n)) \rightarrow (m \doteq \text{arrMax}(n, A)).$$

Karena $i \leq n$ dan $\neg(i < n)$ menyiratkan $i = n$, maka $(m \doteq \text{arrMax}(i, A))$ menyiratkan $(m \doteq \text{arrMax}(n, A))$. Dengan demikian, bukti (1) dan (2) selesai dan I memang merupakan invarian loop yang baik untuk menunjukkan $m \doteq \text{arrMax}(n, A)$.

Contoh 9.2.9 Program berikut akan menghitung hasil bagi bilangan bulat.

```
{x, y > 0}
r := x; q := 0
while y ≤ r do
    r := r - y
    q := q + 1
od
{q ≐ [x/y]}
```

Karena baik r maupun q diubah di dalam loop, misalkan invarian loop I adalah

$$I : y > 0 \wedge r \geq 0 \wedge x \doteq r + y \cdot q$$

Berdasarkan aturan urutan, kita perlu membuktikan pernyataan-pernyataan berikut:

- (1) $\{x, y > 0\} r := x; q := 0 \{I\}$
- (2) $\{I\} \text{ **while** } y \leq r \text{ **do** } r := r - y; q := q + 1 \text{ **od** } \{q \doteq [x/y]\}$

(1) mudah dibuktikan dengan menggunakan aturan penugasan dua kali, aturan urutan sekali, dan aturan implikasi sekali: $x, y > 0 \rightarrow I[r \sim x, q \sim 0]$, yaitu, $x, y > 0 \rightarrow (y > 0 \wedge x \geq 0 \wedge x \doteq x + y \cdot 0)$ adalah benar.

(2) membutuhkan aturan iterasi dan aturan implikasi: Kita perlu membuktikan

- (2.1) $\{I \wedge y \leq r\} r := r - y; q := q + 1 \{I\}$
- (2.2) $(I \wedge \neg(y \leq r)) \rightarrow (q \doteq [x/y])$

Cara membuktikan (2.1) sama dengan cara membuktikan (1), kita perlu menunjukkan

$$(I \wedge y \leq r) \rightarrow I[r \sim r - y, q \sim q + 1]$$

yang, ketika saya membukanya, adalah

$$(y > 0 \wedge r \geq 0 \wedge x \doteq r + y \cdot q \wedge y \leq r) \rightarrow (y > 0 \wedge r - y \geq 0 \wedge x \doteq (r - y) + y \cdot (q + 1))$$

Karena $r - y \geq 0$ adalah konsekuensi logis dari $y \leq r$, dan $x \doteq (r - y) + y \cdot (q + 1)$ adalah konsekuensi logis dari $x = r + y \cdot q$, maka (2.1) benar. Bukti (2.2) mudah karena $x = r + y \cdot q$ jika dan hanya jika $x/y = r/y + q$ ketika $y > 0$. Karena $(y > r \wedge r \geq 0)$ menyiratkan $0 \leq r/y < 1$, maka $L[x/y] = L[r/y + q] = L[r/y] + q = 0 + q = q$.

Kebenaran Parsial dan Total

Aturan Hoare yang disajikan pada bagian sebelumnya berkaitan dengan kebenaran parsial program: jika program berhenti, kondisi pasca akan terpenuhi. Jika suatu program benar sebagian dan berakhir, kita katakan program tersebut benar total. Hubungan antara kebenaran parsial dan kebenaran total dapat diungkapkan secara informal dengan persamaan:

$$\text{Kebenaran total} = \text{Penghentian} + \text{Kebenaran parsial}.$$

Kita akan melihat di Bab 11 bahwa penghentian program adalah masalah yang tidak dapat diputuskan dan tidak ada prosedur keputusan untuk menyelesaikan masalah ini secara umum. Kebenaran parsial dapat dibuktikan untuk beberapa rangkap tiga Hoare dan penghentian program perlu dibuktikan secara terpisah. Dengan demikian, pembacaan intuitif dari rangkap tiga Hoare adalah setiap kali P benar dalam keadaan sebelum eksekusi C , maka Q akan benar setelah eksekusi C atau C tidak berhenti. Dalam kasus terakhir, tidak ada "setelah," jadi Q dapat berupa pernyataan apa pun. Bahkan, seseorang dapat memilih Q menjadi salah untuk menyatakan bahwa C tidak berhenti.

Contoh 9.2.10 Pertimbangkan rangkap tiga $\{I\} \text{ while } x \neq a \text{ do skip od } \{x \doteq a\}$. Misalkan invarian loop I adalah $\top$, maka baik $I \wedge x \neq a \rightarrow I$ dan $I \wedge \neg(x \neq a) \rightarrow x \doteq a$ secara trivial benar. Dengan demikian, program tersebut sebagian benar: jika program berhenti, kita harus memiliki $x \doteq a$. Program akan berulang selamanya kecuali jika dimulai dengan $x \doteq a$; oleh karena itu, program tersebut tidak sepenuhnya benar.

Kebenaran total dapat dibuktikan dengan versi yang diperluas dari aturan iterasi, di mana nilai fungsi pada variabel yang berubah dalam perulangan berkurang di bawah urutan yang mapan. Kebenaran total adalah apa yang pada akhirnya kita minati, tetapi biasanya lebih mudah untuk membuktikan kebenaran total dengan menetapkan kebenaran parsial dan penghentian secara terpisah. Penghentian seringkali mudah ditetapkan dengan metode analisis kompleksitas, yang

sering memberikan hasil yang lebih baik daripada penghentian. Misalnya, program pada Contoh 9.2.7 berhenti karena kompleksitas waktu program ini adalah $O(n)$. Untuk program pada Contoh 9.2.9, misalkan $f(q) = L[x/y] - q$, maka $f(q)$ berkurang satu setelah setiap iterasi, dan nilai minimum $f(q)$ adalah 0 ketika kondisi perulangan salah. Kompleksitas waktu program ini adalah $O(L[x/y])$.

9.3 OTOMATISASI PEMBUATAN PERNYATAAN

Setelah melihat hanya beberapa contoh, dua hal berikut ini sangat jelas:

- Bukti kebenaran dalam verifikasi program biasanya panjang dan membosankan, bahkan jika program yang diverifikasi cukup sederhana.
- Ada banyak detail kecil yang rumit untuk dikerjakan dengan benar, banyak di antaranya sepele, misalnya, membuktikan $(r \doteq x \wedge q \doteq 0) \rightarrow (x \doteq r + y \cdot q)$

Banyak upaya telah dilakukan (dan masih dilakukan) untuk mengotomatisasi bukti kebenaran dengan merancang sistem untuk melakukan bagian-bagian yang membosankan dan rumit dalam menghasilkan bukti formal dalam logika Hoare. Sayangnya, para ahli logika telah menunjukkan bahwa pada prinsipnya tidak mungkin untuk merancang prosedur pengambilan keputusan untuk secara otomatis memutuskan kebenaran atau kesalahan dari pernyataan matematika yang sembarang.

Namun, ini tidak berarti bahwa seseorang tidak dapat memiliki prosedur yang akan membuktikan banyak teorema yang berguna. Tidak adanya prosedur pengambilan keputusan umum hanya menunjukkan bahwa seseorang tidak dapat berharap untuk membuktikan semuanya secara otomatis. Dalam praktiknya, sangat mungkin untuk membangun sistem yang akan memekanisasi banyak aspek rutin verifikasi. Bagian ini menjelaskan salah satu pendekatan umum untuk melakukan hal ini.

Pada bagian sebelumnya, telah ditunjukkan bagaimana membuktikan $\{P\} S \{Q\}$ dengan mereduksi tujuan ini menjadi beberapa sub-tujuan dan kemudian menggabungkannya menggunakan aturan Hoare untuk mendapatkan properti yang diinginkan dari S itu sendiri. Misalnya, untuk membuktikan $\{P\} S; T \{Q\}$, kita pertama-tama membuktikan $\{R\} T \{Q\}$ dan $\{P\} S \{R\}$ untuk pernyataan tengah R dan kemudian menyimpulkan $\{P\} S; T \{Q\}$ dengan aturan urutan. Proses ini disebut reduksi tujuan, atau penalaran mundur, karena seseorang bekerja mundur: dimulai dari tujuan menunjukkan $\{P\} S \{Q\}$, seseorang menghasilkan sub-tujuan, sub-sub-tujuan, dan seterusnya, hingga masalah terpecahkan. Prolog juga menggunakan proses ini untuk komputasi.

Contoh 9.3.1 Misalkan seseorang ingin menunjukkan

$$\{x \doteq x_0 \wedge y \doteq y_0\} r := x; x := y; y := r \{y \doteq x_0 \wedge x \doteq y_0\},$$

Kemudian, berdasarkan aturan penugasan dan urutan, pernyataan di atas direduksi menjadi sub-tujuan:

$$\{x \doteq x_0 \wedge y \doteq y_0\} r := x; x := y \{r \doteq x_0 \wedge x \doteq y_0\}$$

karena $\{r \doteq x_0 \wedge x \doteq y_0\} y := r \{y \doteq x_0 \wedge x \doteq y_0\}$ benar menurut aturan penugasan. Dengan argumen serupa, sub-tujuan ini dapat direduksi menjadi

$$\{x \doteq x_0 \wedge y \doteq y_0\} r := x \{r \doteq x_0 \wedge y \doteq y_0\}$$

yang benar berdasarkan aturan penugasan. Pernyataan tengah untuk aturan urutan dihasilkan secara otomatis.

Kami berharap proses yang diilustrasikan oleh contoh di atas dapat diperluas ke program apa pun sehingga semua pernyataan tengah dapat dihasilkan secara otomatis, membebaskan programmer dari beban menyediakan semua pernyataan. Seorang programmer hanya perlu menyediakan prasyarat dan pascakondisi program yang berasal dari spesifikasi masalah, dan semua pernyataan yang dibutuhkan dapat dihasilkan secara otomatis. Dengan menggabungkannya dengan pembukti teorema yang ampuh, semua rumus yang berasal dari aturan Hoare dapat dibuktikan secara otomatis, dan kita sampai pada otomatisasi verifikasi perangkat lunak.

Kondisi Verifikasi

Verifikasi formal suatu program dapat dijelaskan secara ringkas sebagai berikut:

1. Program S dianotasi dengan memasukkan ke dalamnya pernyataan yang menyatakan kondisi yang dimaksudkan untuk benar pada berbagai titik perantara. Langkah ini membutuhkan pemahaman yang baik tentang cara kerja program. Pengguna perlu menyediakan setidaknya prasyarat dan pascakondisi serta invarian loop; pernyataan lain dapat dihasilkan secara otomatis.
2. Kemudian, serangkaian rumus yang disebut kondisi verifikasi dihasilkan dari spesifikasi yang telah dianotasi. Proses ini sepenuhnya mekanis dan mudah dilakukan oleh alat perangkat lunak. Kami akan menjelaskan bagaimana hal ini dilakukan di bagian ini.
3. Kondisi verifikasi dibuktikan oleh pembukti teorema. Mengotomatiskan hal ini masih menjadi tantangan besar.

Definisi 9.3.2 Diberikan program S dengan kondisi pasca Q , misalkan $vc(S, Q)$ menyatakan rumus P sedemikian sehingga $\{P\} S \{Q\}$ benar. $vc(S, Q)$ disebut kondisi verifikasi S terhadap Q .

Jika $vc(S, Q)$ dapat dihasilkan secara otomatis dari S dan Q , maka proses menghasilkan semua pernyataan menjadi otomatis. Menurut definisi, $vc(S, Q)$ tidak unik, dan sifat ini memberikan fleksibilitas untuk pembuatan $vc(S, Q)$ secara otomatis. Selain menjadi dasar untuk sistem verifikasi otomatis, kondisi verifikasi merupakan cara yang berguna untuk bekerja dengan logika Hoare secara manual.

Kita akan mendefinisikan prosedur rekursif $VC(S, Q)$ pada input S dan Q untuk menghasilkan $vc(S, Q)$, yang bertepatan dengan proses reduksi tujuan dari triple Hoare.

Definisi $VC(S, Q)$ mengikuti aturan Hoare pada tiga konstruksi dan aturan urutan dari bahasa pemrograman kecil kita. Hanya ada lima kasus yang perlu dipertimbangkan dalam definisi $VC(S, Q)$ untuk menghasilkan $vc(S, Q)$.

Faktanya, $VC(S, Q)$ akan menghasilkan prasyarat terlemah dalam sebagian besar kasus, kecuali untuk loop while, tetapi kita tidak akan membuat ini lebih tepat atau membahas detailnya di sini.

Untuk loop while, kita tidak memiliki algoritma yang dapat menghasilkan $vc(S, Q)$ untuk setiap loop; hanya beberapa metode heuristik yang tersedia. Untuk membantu alat verifikasi menghasilkan $vc(S, Q)$, seorang programmer perlu menyarankan invarian loop. Artinya, alih-alih kode **while** B **do** S **od**, kita menulis **while** B **do** $\{I\}$ S **od**, di mana I adalah invarian loop yang disarankan. Pra- dan pascakondisi juga dapat dimasukkan ke dalam kode program. Jenis kode program dengan pernyataan ini disebut "program berannotasi," di mana pernyataan disediakan oleh programmer untuk membantu verifikasi.

Pengecualian lain dengan loop while adalah bahwa, selain mengembalikan kondisi verifikasi untuk loop, ia akan membuat formula lain yang validitasnya diperlukan untuk memastikan validitas triple Hoare. Formula ini akan dimasukkan ke dalam tumpukan yang dilambangkan dengan s selama eksekusi VC . Tumpukan s adalah variabel global dan awalnya kosong; ia akan menyimpan formula sebanyak jumlah loop while dalam program.

Algoritma 9.3.3 Algoritma VC menerima program berannotasi C , sebuah formula Q sebagai kondisi pasca dari C , dan mengembalikan kondisi verifikasi P sedemikian sehingga $\{P\} C \{Q\}$ bernilai benar.

```

proc  $VC(C, Q)$ 
1  if  $C$  is " $x := E$ " return  $Q[x \sim E]$ 
2  else if  $C$  is "skip" return  $Q$ 
3  else if  $C$  is "if  $B$  then  $S$  else  $T$  fi"
4    return  $(B \rightarrow VC(S, Q)) \wedge (\neg B \rightarrow VC(T, Q))$ 
5  else if  $C$  is "while  $B$  do  $\{I\}$   $S$  od"
6     $push(I \rightarrow ((B \rightarrow VC(S, I)) \wedge (\neg B \rightarrow Q)), s)$ ; return  $I$ 
7  else if  $C$  is " $S; T$ " return  $VC(S, VC(T, Q))$ 
8  else print("unknown command"); return  $\perp$ 

```

Dalam algoritma di atas, baris 1–2 menangani kasus dasar, ketika perintahnya adalah perintah penugasan atau lewati. Baris 3–4 menangani kondisi (untuk kesederhanaan, kita mengabaikan kasus kondisi ketika bagian else hilang). Baris 5–6 menangani loop while; dan baris 7 menangani urutan perintah. Tiga kasus terakhir melibatkan panggilan rekursif VC .

Teorema 9.3.4 Asumsikan S adalah program yang sah dari bahasa pemrograman kecil kita. Keluaran $VC(S, Q)$ adalah rumus P dalam logika orde pertama sedemikian sehingga $\{P\} S \{Q\}$ benar, jika semua rumus yang dihasilkan oleh $VC(S, Q)$ dalam tumpukan s valid.

Menurut definisi, $VC(S, Q)$ mengembalikan kondisi verifikasi $vc(S, Q)$ jika kondisi teorema terpenuhi. Subbagian berikut akan dikhususkan untuk pembuktian teorema.

Bukti Teorema 9.3.4

Pertama, kita perlu menunjukkan bahwa VC akan berakhir. Ini adalah tugas yang mudah karena argumen pertama dari VC , yang merupakan sepotong kode, menjadi semakin kecil untuk setiap panggilan rekursif, dan akan bermuara pada perintah penugasan atau perintah lewati; tidak mungkin ada rantai panggilan rekursif yang tak terbatas. Jadi, algoritma VC harus berakhir.

Pembuktian teorema ini akan dilakukan dengan induksi pada struktur C . Argumen induktif tersebut memiliki dua bagian. Pertama, ditunjukkan bahwa hasilnya berlaku untuk perintah penugasan dan perintah lewati. Kedua, ditunjukkan bahwa ketika C bukan perintah penugasan, jika hasilnya berlaku untuk perintah-perintah penyusun C (ini disebut hipotesis induksi), maka hasilnya juga berlaku untuk C .

Bagian pertama dari bagian-bagian ini disebut kasus dasar induksi, dan bagian kedua disebut kasus induktif. Dari kasus dasar dan induktif, disimpulkan bahwa hasilnya berlaku untuk semua perintah. Misalnya, untuk menunjukkan bahwa VC mengembalikan sebuah formula, karena semua kasus dasar mengembalikan sebuah formula, dengan asumsi semua panggilan rekursif akan mengembalikan sebuah formula, maka VC akan mengembalikan sebuah formula dengan memeriksa semua kasus induktif. Kasus perintah skip bersifat trivial; kasus-kasus lainnya akan dibahas secara terpisah.

Penugasan

Kondisi verifikasi untuk $x := E$ dan Q adalah $Q[x \sim E]$, yang merupakan prasyarat terlemah untuk $x := E$ dan Q . Dengan demikian, kita memiliki $VC(x := E, Q) = Q[x \sim E]$, dan mudah untuk memverifikasi bahwa $VC(x := E, Q)$ adalah $vc(x := E, Q)$.

Urutan

Untuk kode " $S; T$ " dan pascakondisi Q , $VC(S; T, Q)$ mengembalikan $VC(S, VC(T, Q))$. Jika $VC(S, Qr)$ dan $Q' = VC(T, Q)$ keduanya adalah $vc(S, Q')$ dan $vc(T, Q)$, maka $VC(S, VC(T, Q))$ adalah $vc(S; T, Q)$.

Contoh 9.3.5 Suatu kondisi verifikasi untuk kode dan pascakondisi pada Contoh 9.3.1 mudah diperoleh karena hanya melibatkan dua kasus, penugasan dan urutan:

$$\begin{aligned}
 & VC(\{r := x; x := y; y := r\}, y \doteq x_0 \wedge x \doteq y_0) \\
 &= VC(\{r := x; x := y\}, VC(y := r, y \doteq x_0 \wedge x \doteq y_0)) \\
 &= VC(\{r := x; x := y\}, r \doteq x_0 \wedge x \doteq y_0) \\
 &= VC(r := x, VC(x := y, r \doteq x_0 \wedge x \doteq y_0)) \\
 &= VC(r := x, r \doteq x_0 \wedge y \doteq y_0) \\
 &= (x \doteq x_0 \wedge y \doteq y_0)
 \end{aligned}$$

yang merupakan prasyarat untuk perintah pertukaran x dan y . Tiga panggilan rekursif VC menghasilkan $vc(y := r, y \doteq x_0 \wedge x \doteq y_0)$, $vc(x := y, r \doteq x_0 \wedge x \doteq y_0)$, dan $vc(r := x, r \doteq x_0 \wedge y \doteq y_0)$, secara berturut-turut.

Kondisional

Berdasarkan Proposisi 9.2.5, $P = (B \rightarrow vc(S, Q)) \wedge (\neg B \rightarrow vc(T, Q))$ adalah kondisi verifikasi untuk kode **if B then S else T fi** dan Q . Perhatikan bahwa P adalah prasyarat terlemah untuk perintah kondisional, jika $vc(S, Q)$ dan $vc(T, Q)$ keduanya merupakan prasyarat terlemah untuk S dan T , masing-masing. Oleh karena itu, $VC(\text{if } B \text{ then } S \text{ else } T \text{ fi}, Q)$ mengembalikan $P = (B \rightarrow VC(S, Q)) \wedge (\neg B \rightarrow VC(T, Q))$. Berdasarkan hipotesis induksi, baik $\{VC(S, Q)\} S \{Q\}$ dan $\{VC(T, Q)\} T \{Q\}$ adalah benar. Berdasarkan Proposisi 9.2.5

$$\{P\} \text{if } B \text{ then } S \text{ else } T \text{ fi } \{Q\}$$

Itu benar.

Contoh 9.3.6 Kondisi pasca dari Contoh 9.2.3 adalah $x \leq 10$.

$$\begin{aligned} & VC(\text{if } (x < 10) \text{ then } x := x \text{ else } x := 10 \text{ fi}, x \leq 10) \\ &= (x < 10 \rightarrow VC(x := x, x \leq 10)) \wedge (x \geq 10 \rightarrow VC(x := 10, x \leq 10)) \\ &= (x < 10 \rightarrow x \leq 10) \wedge (x \geq 10 \rightarrow 10 \leq 10) \\ &\equiv (\top) \wedge (x \geq 10 \rightarrow \top) \\ &\equiv \top \end{aligned}$$

Perhatikan bahwa $\top$ adalah yang terlemah di antara semua pernyataan.

Iterasi

$VC(\text{while } B \text{ do } \{I\} S \text{ od}, Q)$ mengembalikan I , yang diberikan oleh pengguna untuk memastikan bahwa I adalah invarian loop. Selain itu, rumusnya

$$I \rightarrow ((B \rightarrow VC(S, I)) \wedge (\neg B \rightarrow Q)),$$

yang setara dengan $((I \wedge B) \rightarrow VC(S, I)) \wedge ((I \wedge \neg B) \rightarrow Q)$, dibuat dan disimpan dalam tumpukan s . Validitas $I \wedge B \rightarrow VC(S, I)$ memastikan bahwa I berlaku setelah eksekusi badan loop S . Berdasarkan hipotesis induksi, $\{VC(S, I)\} S \{I\}$ adalah benar. Karena $I \wedge B \rightarrow VC(S, I)$, berdasarkan aturan implikasi, $\{I \wedge B\} S \{I\}$ adalah benar, yaitu, I memang merupakan invarian loop. Dengan demikian, berdasarkan aturan iterasi, $\{I\} \text{while } B \text{ do } S \text{ od } \{I \wedge \neg B\}$ is **True**.

Terakhir, validitas $(I \wedge \neg B) \rightarrow Q$ memastikan bahwa Q berlaku ketika loop berakhir. Berdasarkan aturan implikasi, $\{I\} \text{while } B \text{ do } S \text{ od } \{Q\}$ berlaku. Jelas bahwa untuk perintah iterasi, invarian loop I yang dikembalikan oleh VC dapat menjadi kondisi verifikasi jika rumus $I \rightarrow ((B \rightarrow VC(S, I)) \wedge (\neg B \rightarrow Q))$ valid.

Tabel berikut merangkum kondisi verifikasi yang dikembalikan oleh VC untuk berbagai perintah dan postkondisi Q :

code C	what $VC(C, Q)$ returns
$skip$	Q
$x := E$	$Q[x \sim E]$
$S; T$	$VC(S, VC(T, Q))$
if B then S else T fi	$(B \rightarrow VC(S, Q)) \wedge (\neg B \rightarrow VC(T, Q))$
while B do $\{I\} S$ od	I

Semua kondisi verifikasi dapat dihasilkan secara otomatis kecuali untuk kasus perintah iterasi. Dalam kasus ini, validitas $I \rightarrow ((B \rightarrow VC(S, I)) \wedge (\neg B \rightarrow Q))$ memastikan bahwa I adalah kondisi verifikasi. Ini melengkapi bukti Teorema 9.3.4.

Algoritma $VC(S, Q)$ menunjukkan bagaimana pernyataan yang dibutuhkan untuk aturan Hoare diturunkan dari postkondisi dan invarian loop secara otomatis dalam proses reduksi tujuan atau penalaran mundur. Penalaran mundur dapat berguna untuk loop bersarang, di mana postkondisi loop dalam dapat diturunkan dari invarian loop luar.

Contoh 9.3.7 Prekondisi untuk program berikut adalah $P: n \geq 0 \wedge x \doteq n \wedge y \doteq 1$ dan invarian loop adalah $I: x! * y \doteq n!$, di mana $x!$ adalah fungsi faktorial: $0! \doteq 1$ dan $(x + 1)! \doteq x! * (x + 1)$.

$$VC(\text{while } (x \neq 0) \text{ do } \{I : x! y \doteq n!\} y := x * y; x := x - 1 \text{ od}, y \doteq n!) = (x! y \doteq n!)$$

Rumus yang dimasukkan ke dalam tumpukan dan perlu dibuktikan adalah

$$\begin{aligned}
 & x! y \doteq n! \rightarrow ((x! \doteq 1 \wedge x \neq 0 \rightarrow VC([y := y * x; x := x - 1], x! y \doteq n!)) \wedge (x \doteq 0 \rightarrow y \doteq n!)) \\
 &= x! y \doteq n! \rightarrow ((x! \doteq 1 \wedge x \neq 0 \rightarrow VC(y := y * x, VC(x := x - 1, x! y \doteq n!))) \wedge (x \doteq 0 \rightarrow y \doteq n!)) \\
 &= x! y \doteq n! \rightarrow ((x! \doteq 1 \wedge x \neq 0 \rightarrow VC(y := y * x, (x - 1)! y \doteq n!)) \wedge (x \doteq 0 \rightarrow y \doteq n!)) \\
 &= x! y \doteq n! \rightarrow ((x! \doteq 1 \wedge x \neq 0 \rightarrow (x - 1)! (yx) \doteq n!) \wedge (x \doteq 0 \rightarrow y \doteq n!)) \\
 &= x! y \doteq n! \rightarrow ((x \doteq 1 \rightarrow x! y \doteq n!) \wedge (x \doteq 0 \rightarrow y \doteq n!)) \\
 &= x! y \doteq n! \rightarrow (x \doteq 0 \rightarrow y \doteq n!)
 \end{aligned}$$

Langkah kedua terakhir menggunakan sifat bahwa $x! \doteq 1 \wedge x \neq 0$ setara dengan $x \doteq 1$ karena $x! \doteq 1$ setara dengan $x \doteq 0 \vee x \doteq 1$. Langkah terakhir dapat diperoleh dengan dua cara: (a): menggunakan kesetaraan logis $X \rightarrow ((Y \rightarrow X) \wedge Z) \equiv X \rightarrow Z$, di mana X adalah $(x! y \doteq n!)$, Y adalah $(x! \doteq 1 \wedge x \neq 0)$, dan Z adalah $(x \doteq 0 \rightarrow y \doteq n!)$. (b): menggunakan persilangan kesamaan untuk menyederhanakan $x! y \doteq n! \rightarrow (\dots \rightarrow x! y \doteq n!) \wedge \dots$ menjadi $x! y \doteq n! \rightarrow (\dots \rightarrow n! \doteq n!) \wedge \dots$

Menerapkan persilangan kesamaan dua kali $x! y \doteq n! \rightarrow (x \doteq \rightarrow y \doteq n!)$ disederhanakan menjadi $x! y \doteq n! \rightarrow (x \doteq 0 \rightarrow y \doteq x! y)$, lalu menjadi $x! y \doteq n! \rightarrow (x \doteq 0 \rightarrow y \doteq 0! y)$ yang mudah disederhanakan menjadi benar dengan $0! \doteq 1$ dan $1 * y \doteq y$.

Bisakah kita menghilangkan tumpukan yang digunakan dalam VC ? Artinya, alih-alih menyimpan $I \rightarrow ((B \rightarrow VC(S, I)) \wedge (\neg B \rightarrow Q))$, bisakah kita membiarkan $VC(\text{while } B \text{ do } \{I\} S \text{ od}, Q)$ mengembalikan $I \rightarrow ((B \rightarrow VC(S, I)) \wedge (\neg B \rightarrow Q))$ sebagai bagian dari outputnya? Contoh tandingan berikut menunjukkan bahwa pendekatan ini tidak tepat.

Contoh 9.3.8 Misalkan tripel Hoare adalah

$$\{x \doteq 0\} \text{ while } (x < 5) \text{ do } x := x + 1 \text{ od } \{x \doteq 5\}$$

yang dapat dibuktikan kebenarannya dengan aturan Hoare dengan invarian loop $x \leq 5$.

Jika kita memodifikasi VC sedemikian rupa sehingga $VC(\text{while } B \text{ do } \{I\} S \text{ od}, Q)$ mengembalikan

$$I \wedge \left(I \rightarrow ((B \rightarrow VC(S, I)) \wedge (\neg B \rightarrow Q)) \right)$$

yang setara dengan

$$I \wedge (B \rightarrow VC(S, I)) \wedge (\neg B \rightarrow Q)$$

Misalkan I adalah $x \doteq 0 \vee x \doteq 1$ untuk program di atas, maka $VC(\text{while } (x < 5) \text{ do } \{I\} x := x + 1 \text{ od}, x \doteq 5)$ akan mengembalikan

$$I \wedge (x < 5 \rightarrow VC(x := x + 1, I)) \wedge (x \geq 5 \rightarrow x \doteq 5)$$

yang dapat disederhanakan menjadi

$$P : (x \doteq 0 \vee x \doteq 1) \wedge (x < 5 \rightarrow (x \doteq 1 \vee x \doteq 0)) \wedge (x \geq 5 \rightarrow x \doteq 5)$$

Karena $VC(x := x + 1, I)$ mengembalikan $(x + 1 \doteq 0 \vee x + 1 \doteq 1)$. P dapat disederhanakan lebih lanjut menjadi $x \doteq 0 \vee (x \doteq 1 \wedge x \doteq -1)$. Sekarang, mudah untuk menunjukkan bahwa $x \doteq 0 \rightarrow P$ adalah benar.

Menurut aturan Hoare asli, kita tidak dapat menunjukkan bahwa $x \doteq 0 \vee x \doteq 1$ adalah invarian loop yang valid. Dengan demikian, modifikasi VC tidak valid.

Mengimplementasikan VC dalam Prolog

Karena Prolog mahir dalam pencocokan pola, mudah untuk mengimplementasikan algoritma $VC(S, Q)$ dalam Prolog (sebagai $vc(S, Q)$ karena VC adalah variabel Prolog). Tugas pertama adalah membiarkan program Prolog menguraikan kode dengan mudah. Untuk tujuan ini, kita mengubah sintaks bahasa pemrograman kita sebagai berikut:

<i>command</i>	<i>before</i>	<i>now</i>
assignment	$x := E$	$\text{let}(x, E)$
conditional	if B then S else T fi	$\text{ite}(B, S, T)$
iteration	while B do $\{I\}$ S od	$\text{while}(B, I, S)$
sequence	$S_1; S_2; \dots; S_n$	$[S_1, S_2, \dots, S_n]$

Untuk iterasi, alih-alih menggunakan tumpukan untuk menyimpan rumus, kita cukup menulis rumus di layar. Berikut adalah kode Prolog untuk menangani penugasan, kondisi, iterasi, dan lompatan:

```
% vc1(S1, Post, VC): VC = vc(S1, Post), S1 is a single command

% assignment
vc1(let(X, E), Post, VC) :- substitute(X, E, Post, VC).
% VC is the result of Post by replacing X in Post by E.

% conditional
vc1(ite(B, S, T), Post, VC) :-
    vc(S, Post, VC1), vc(T, Post, VC2),
    VC = and((B -> VC1), (neg(B) -> VC2)).

% iteration
% use vc1(while(B, Inv, S), Post, Inv) as an alternative.
vc1(while(B, Inv, S), Post, or(and(neg(B), Post), Inv)) :-
    vc(S, Inv, VC), % VC is vc(S, Inv)
    write((and(Inv, B) -> VC)), nl, % part 1 of a conjunction
    write((and(Inv, neg(B)) -> Post)), nl. % part 2 of a conjunction
```

Penanganan penugasan menggunakan fungsi utilitas pengganti, yang melakukan substitusi:

```
% substitute(X, V, E, R) succeeds when every occurrence of X in
% expression E is replaced by V and the modified E is R.
substitute(X, V, X, V) :- !.
substitute(X, V, E, R) :-
    E =.. [F|Args0],
    maplist(substitute(X, V), Args0, Args),
    R =.. [F|Args].
```

Untuk urutan perintah S , pertama-tama kita membalik S menjadi S_2 dan mengerjakan S_2 dari awal hingga akhir, seperti yang diilustrasikan oleh kode berikut:

```
% vc(S, Post, VC) succeeds when VC is generated
% from the program S and the postcondition Post.

vc(P, Post, VC) :- is_list(P), !, reverse(P, P2), vcR(P2, Post, VC).
```

```

vc(P, Post, VC) :- vc1(P, Post, VC). % if S is not a list

% vcR(SR, Post, VC) generates VC from SR and Post, where
% SR is a list of instructions in reversed order.
vcR([], Post, Post).
vcR([Last | Rest], Post, VC) :-
    vc1(Last, Post, Post1), vcR(Rest, Post1, VC).

```

Pada contoh berikut, kita mendefinisikan sebuah triple $tri_i(P, S, Q)$, di mana i adalah sebuah angka, dan kueri q_i untuk menghasilkan dan menampilkan kondisi verifikasi untuk $tri_i(P, S, Q)$. Validitas semua rumus yang ditampilkan memastikan bahwa triple tersebut benar.

Contoh 9.3.9 Triple tersebut ditentukan sebagai berikut:

```

tri1(true,
    [ ite(a>b, let(x, b), let(x, a)), ite(x > c, let(x, c), skip)],
    x=min(a,min(b,c))).

```

```

q1 :- tri1(P, S, Q), vc(S, Q, VC), write((P -> VC)), nl.

```

The query “?- q1.” produces one formula:

```

true -> and((a>b -> and((b>c -> c=min(a, min(b,c))),
    (neg(b>c) -> b=min(a, min(b,c))))),
    (neg(a>b) -> and((a>c -> c=min(a, min(b,c))),
    (neg(a>c) -> a=min(a, min(b,c))))))

```

yang mewakili rumus

$$T \rightarrow ((a > b \rightarrow ((b > c \rightarrow c = m) \wedge (\neg(b > c) \rightarrow b = m))) \wedge (\neg(a > b) \rightarrow ((a > c \rightarrow c = m) \wedge (\neg(a > c) \rightarrow a = m))))$$

di mana m adalah singkatan dari $\min(a, \min(b, c))$.

Contoh 9.3.10 Triple tersebut ditentukan sebagai berikut:

```

tri2(and(n>=0, and(k=0, result=0)),
    while(k<n, and(0=<k, and(k=<n, result=sum(0, k, a))),
        [let(result, result+a(k)), let(k, k+1)]),
    result=sum(0, n, a)).

```

```

q2 :- tri2(P, S, Q), vc(S, Q, VC), write((P -> VC)), nl.

```

Permintaan “?- q2” menghasilkan tiga rumus, dua rumus pertama dari tumpukan dan rumus terakhir dari “write((P -> VC))”:

```
and(and(0=<k, and(k=<n, result=sum(0,k,a))), k<n) ->
  and(0=<k+1, and(k+1=<n, result+a(k)=sum(0,k+1,a)))
and(and(0=<k, and(k=<n, result=sum(0,k,a))), neg(k<n)) ->
  result=sum(0,n,a)
and(n>=0, and(k=0, result=0)) -> and(0=<k, and(k=<n, result=sum(0,k,a)))
```

di mana $sum(b, e, A) = 0$ jika $(b \geq e)$; jika tidak, $sum(b, e, A) = sum(b, e - 1, A) + A[e - 1]$. Ketiga rumus tersebut dapat ditulis ulang sebagai

$$\begin{aligned} (0 \leq k \wedge k \leq n \wedge result = sum(0, k, a) \wedge k < n) &\rightarrow \\ (0 \leq k + 1 \wedge k + 1 \leq n \wedge result + a(k) = sum(0, k + 1, a)) & \\ (0 \leq k \wedge k \leq n \wedge result = sum(0, k, a) \wedge \neg(k < n)) &\rightarrow result = sum(0, n, a) \\ (n \geq 0 \wedge k = 0 \wedge result = 0) &\rightarrow (0 \leq k \wedge k \leq n \wedge result = sum(0, k, a)) \end{aligned}$$

dan dapat dibuktikan dengan mudah.

Contoh 9.3.11 Triple tersebut ditentukan sebagai berikut:

```
tri3(and(n >= 0, and(x=n, y=1)),
     while(neg(x = 0), fac(x)*y = fac(n),
           [let(y, x*y), let(x, x-1)]),
     y=fac(n)).
```

```
q3 :- tri3(P, S, Q), vc(S, Q, VC), write((P -> VC)), nl.
```

Permintaan “?- q3” menghasilkan tiga rumus:

```
and(fac(x)*y=fac(n), neg(x=0)) -> fac(x-1)*(x*y)=fac(n)
and(fac(x)*y=fac(n), neg(neg(x=0))) -> y=fac(n)
and(n>=0, and(x=n,y=1)) -> fac(x)*y=fac(n)
```

di mana $fac(n)$ adalah fungsi faktorial: $fac(0) = 1, fac(i + 1) = fac(i) * (i + 1)$.

Ketiga rumus tersebut dapat ditulis ulang sebagai

$$\begin{aligned} (fac(x) * y = fac(n) \wedge \neg(x = 0)) &\rightarrow fac(x - 1) * (x * y) = fac(n) \\ (fac(x) * y = fac(n) \wedge \neg\neg(x = 0)) &\rightarrow y = fac(n) \\ (n \geq 0 \wedge x = n \wedge y = 1) &\rightarrow fac(x) * y = fac(n) \end{aligned}$$

dan dapat dibuktikan dengan mudah.

Ketiga contoh di atas menunjukkan bagaimana rumus dapat dihasilkan dari kondisi pasca dan invarian loop. Semua pernyataan tengah, kecuali invarian loop, dihasilkan secara otomatis. Rumus-rumus ini kemudian dimasukkan ke dalam pembukti teorema otomatis. Setelah terbukti valid, triple Hoare terbukti benar, dan program tersebut sebagian benar.

Lebih sering, pembukti teorema tidak dapat membuktikan rumus-rumus ini secara otomatis; jika gagal, saran diminta dari pengguna. Analisis kegagalan tersebut mungkin menunjukkan bahwa kita perlu menambahkan konjungsi ke invarian loop, ke kondisi pra, atau memodifikasi kondisi pasca. Kemudian kita menjalankan *vc* dengan kode beranotasi baru, mendapatkan rumus baru, dan memasukkannya ke dalam pembukti teorema. Proses ini berulang hingga kita menemukan kondisi pra, kondisi pasca, dan invarian loop yang baik.

Tujuan penelitian saat ini adalah untuk membangun sistem yang mengurangi peran ahli manusia yang lambat dan mahal seminimal mungkin. Hal ini dapat dicapai dengan cara berikut:

- Mengurangi jumlah dan kompleksitas anotasi yang dibutuhkan
- Meningkatkan kekuatan pembukti teorema

Dari implementasi *vc*, sekarang jelas bahwa keberhasilan logika Hoare untuk verifikasi formal bergantung pada (i) memperoleh invarian loop yang baik untuk setiap loop *while* dalam suatu program dan (ii) memiliki pembukti teorema yang kuat yang dapat membuktikan validitas rumus yang diturunkan oleh *vc*. Masalah pertama akan dibahas di bagian selanjutnya. Masalah kedua telah dibahas di bab-bab sebelumnya dalam buku ini dan akan dibahas di Bab 12. Karena bukti kebenaran program biasanya sangat panjang dan membosankan, maka berguna untuk memeriksa bukti secara mekanis atau otomatis, meskipun hanya dapat dilakukan dengan bantuan manusia.

9.4 MEMPEROLEH INVARIAN LOOP YANG BAIK

Konsep invarian loop adalah salah satu ide dasar konstruksi perangkat lunak. Kita telah melihat beberapa contoh invarian loop. Kami berharap contoh-contoh ini membuktikan klaim bahwa invarian adalah kunci setiap loop: untuk merancang loop baru agar benar membutuhkan pemanggilan invarian yang tepat dan untuk memahami loop yang ada membutuhkan pemahaman invarian loop tersebut.

Sejauh ini, menemukan invarian untuk sebuah loop secara tradisional merupakan tanggung jawab manusia: baik orang yang melakukan verifikasi atau programmer yang menulis loop tersebut sejak awal. Solusi yang lebih baik, jika memungkinkan, adalah sintesis program. Generasi otomatis invarian loop masih merupakan topik penelitian yang sedang berlangsung.

Untuk rangkap tiga $\{I\} \textbf{while } B \textbf{ do } S \textbf{ od } \{Q\}$, *I* adalah invarian yang benar untuk loop jika memenuhi kondisi berikut:

1. *I* benar dalam keadaan sebelum eksekusi loop.
2. Setiap eksekusi badan loop *S*, yang dimulai dalam keadaan apa pun di mana *I* dan *B* keduanya benar, akan menghasilkan keadaan di mana *I* berlaku lagi.

Jika sifat-sifat ini terpenuhi, maka setiap eksekusi akhir dari loop akan menghasilkan keadaan di mana *I* dan $\neg B$ terpenuhi; yang terakhir disebut kondisi keluar.

Kita dapat melihat gagasan invarian loop dari perspektif konstruktif seorang programmer yang mengarahkan programnya untuk mencapai keadaan yang memenuhi sifat yang diinginkan, yaitu *postcondition*. Dalam pandangan ini, konstruksi program adalah bentuk pemecahan masalah, dan berbagai struktur kontrol adalah teknik pemecahan masalah; sebuah loop memecahkan masalah melalui pendekatan bertahap. Berikut ini, presentasi kami didasarkan pada survei mereka.

- Menggeneralisasi *postcondition* (karakterisasi solusi yang mungkin) menjadi kondisi yang lebih luas: invarian.
- Akibatnya, *postcondition* dapat didefinisikan sebagai gabungan dari invarian dan kondisi lain, yaitu kondisi keluar.
- Temukan cara untuk mencapai invarian dari keadaan komputasi sebelumnya: inisialisasi.
- Temukan cara, dengan diberikan keadaan yang memenuhi invarian, untuk sampai ke keadaan lain, yang masih memenuhi invarian tetapi lebih dekat, dalam arti yang sesuai, ke kondisi keluar: badan.

Pentingnya penyajian proses loop ini adalah bahwa hal itu menyoroti sifat invarian: itu adalah bentuk umum dari kondisi pasca yang diinginkan, yang dalam kasus khusus (diwakili oleh kondisi keluar) akan memberi kita kondisi pasca tersebut.

Pandangan tentang invarian ini, sebagai cara khusus untuk menggeneralisasi tujuan yang diinginkan dari komputasi loop, menjelaskan mengapa invarian loop merupakan sifat yang sangat penting dari loop; dapat dikatakan bahwa memahami loop berarti memahami invarian-nya.

Invarian dari Generalisasi Postkondisi

Banyak programmer masih kesulitan untuk menemukan invarian. Meskipun ide dasarnya seringkali jelas, menemukan invarian yang baik adalah tugas yang sulit. Kita membutuhkan metode untuk menghasilkan invarian loop secara otomatis. Untuk mengilustrasikan ide generalisasi postkondisi untuk mendapatkan invarian, kita akan menggunakan algoritma Euclid yang telah berusia ribuan tahun untuk menghitung pembagi persekutuan terbesar (FPB) dari dua bilangan bulat positif.

Contoh 9.4.1 Postkondisi dari algoritma FPB adalah

$$y \doteq \gcd(a, b),$$

di mana prasyaratnya adalah $a > 0 \wedge b > 0$. Langkah pertama generalisasi adalah mengganti kondisi ini dengan

$$x \doteq 0 \wedge \gcd(x, y) \doteq \gcd(a, b)$$

dengan variabel baru x , memanfaatkan sifat matematika bahwa, untuk setiap y , $\gcd(0, y) \doteq y$. Konjungsi kedua, yaitu $\gcd(x, y) \doteq \gcd(a, b)$, generalisasi dari kondisi pasca, akan

berfungsi sebagai invarian; konjungsi pertama akan berfungsi sebagai kondisi keluar. Untuk mendapatkan badan loop, kita memanfaatkan sifat matematika lain: untuk setiap x dan y , $\gcd(x, y) \doteq \gcd(y, x)$ dan $\gcd(x, y) \doteq \gcd(x - y, y)$ jika $x > y$, menghasilkan algoritma yang terkenal:

```

 $x := a; y := b$ 
 $\{I : x \geq 0 \wedge y > 0 \wedge \gcd(x, y) \doteq \gcd(a, b)\}$ 
while  $x \neq 0$  do
    if  $x < y$  then  $z := x; x := y; y := z$  fi
     $x := x - y$ 
od

```

Bukti kebenarannya langsung mengikuti dari sifat matematis $\gcd$. Bentuk algoritma Euclid ini menggunakan pengurangan; kita dapat mengganti pengurangan dengan fungsi sisa, yaitu mengganti $x - y$ dengan $x \% y$ (% adalah fungsi sisa), sehingga menghasilkan algoritma yang lebih efisien. Dalam hal ini, kita membutuhkan sifat bahwa $\gcd(x, y) \doteq \gcd(x \% y, y)$.

Dengan menggunakan kriteria klasifikasi yang diusulkan dalam artikel survei Furia dkk., invarian untuk algoritma $\gcd$, yaitu $I: x \geq 0 \wedge y > 0 \wedge \gcd(x, y) \doteq \gcd(a, b)$, dibagi menjadi dua kategori:

- Konjungsi terakhir dari invarian adalah invarian esensial, yang mewakili pelemahan kondisi pasca. Dua konjungsi pertama adalah invarian pembatas, yang menunjukkan bahwa keadaan tetap berada dalam batas umum tertentu dan memastikan bahwa bagian "esensial" didefinisikan dengan baik.
- Strategi yang mengarah ke invarian esensial adalah pemisahan (uncoupling), yang menggantikan properti dari satu variabel (y dalam kasus ini), yang digunakan dalam kondisi pasca, dengan properti dari dua variabel (y dan x), yang digunakan dalam invarian.

Seperti yang diilustrasikan oleh contoh di atas, invarian esensial adalah mutasi (seringkali, pelemahan) dari kondisi pasca loop. Teknik mutasi berikut ini sangat umum.

- *Relaksasi Konstan*

Relaksasi konstan menggantikan konstanta n (lebih umum, ekspresi yang tidak berubah selama eksekusi algoritma) dengan variabel i dan menggunakan $i \doteq n$ sebagai bagian atau seluruh kondisi keluar. Dalam Contoh 9.2.7, relaksasi konstan digunakan untuk mendapatkan invarian loop, yaitu $s \doteq \text{sum}(i, A)$, dari kondisi pasca $s \doteq \text{sum}(n, A)$. Kondisi ini mudah ditetapkan pada awalnya untuk $i \doteq 0$ dan $s \doteq 0$ dan mudah diperluas ke i yang bertambah, menghasilkan kondisi pasca ketika i mencapai n .

- *Pemisahan*

Pemisahan adalah mengganti sebuah variabel dengan dua variabel, menggunakan kesamaan keduanya sebagai bagian atau seluruh kondisi keluar. Pemisahan digunakan dalam algoritma $\gcd$ di mana kita mengganti $y \doteq \gcd(a, b)$ dengan $x \doteq 0 \wedge$

$\gcd(x, y) \doteq \gcd(a, b)$. Uncoupling juga digunakan dalam Contoh 9.3.7 di mana kita mengganti $y \doteq n!$ dengan $x \doteq 0 \wedge x! \cdot y \doteq n!$.

- **Penghapusan suku**

Penghapusan suku menghilangkan subformula (biasanya konjungsi), yang memberikan pelemahan langsung. Penghapusan suku sering diterapkan setelah uncoupling. $x \doteq 0$ dihilangkan dari $x \doteq 0 \wedge \gcd(x, y) \doteq \gcd(a, b)$ dalam algoritma $\gcd$ dan juga dari $x \doteq 0 \wedge x! \cdot y \doteq n!$ dalam algoritma faktorial, untuk mendapatkan invarian esensial. Suku yang dihilangkan sering masuk ke kondisi keluar dari loop.

- **Penuaan**

Penuaan adalah mengganti variabel (lebih umum, ekspresi) dengan ekspresi yang mewakili nilai variabel pada iterasi sebelumnya dari loop. Penuaan biasanya mengakomodasi perbedaan "selisih satu" antara saat variabel dievaluasi dalam invarian dan saat diperbarui dalam badan loop. Biasanya, proses penuaan (aging) digunakan untuk menghasilkan invarian pembatas. Misalnya, jika $i > 0$ muncul dalam kondisi loop dan $i := i - 1$ muncul dalam badan loop, proses penuaan akan menghasilkan $i + 1 > 0$ (atau $i \geq 0$) sebagai invarian pembatas. Di sisi lain, jika $j < n$ muncul dalam kondisi loop dan $j := j + 1$ muncul dalam badan loop, maka proses penuaan akan menghasilkan $j - 1 < n$ (atau $j \leq n$) sebagai invarian pembatas.

Teknik generalisasi di atas dapat diimplementasikan dengan metode heuristik dan dengan demikian menghasilkan invarian secara otomatis. Jumlah penelitian tentang pembangkitan invarian otomatis sangat besar dan tersebar selama lebih dari tiga dekade; ini mencerminkan peran penting yang dimainkan invarian dalam analisis formal dan verifikasi program.

Sintesis Program dari Invarian

Sintesis program adalah tugas membangun program yang terbukti memenuhi postcondition. Berbeda dengan verifikasi program, program harus dibangun daripada diberikan. Kedua bidang tersebut menggunakan teknik pembuktian formal, dan keduanya mencakup pendekatan dengan tingkat otomatisasi yang berbeda. Berbeda dengan teknik pemrograman otomatis, postcondition dalam sintesis program biasanya berupa pernyataan non-algoritma dalam bahasa logika yang sesuai.

Jika kita mampu menghasilkan invarian loop dari postcondition secara otomatis, maka relatif mudah untuk membangun kode yang memenuhi postcondition. Karena algoritma yang berbeda untuk menyelesaikan masalah yang sama membutuhkan invarian yang berbeda, menghasilkan invarian yang berbeda akan menghasilkan algoritma yang berbeda pula.

Misalkan kita ingin mengurutkan sebuah array A yang terdiri dari n elemen. Misalkan A_0 menyatakan array asli A . Kondisi pasca dari algoritma pengurutan adalah $\text{permu}(A, A_0) \wedge \text{sorted}(A, 0, n)$, di mana $\text{permu}(A, A_0)$ bernilai benar jika dan hanya jika A merupakan permutasi dari A_0 dan $\text{sorted}(A, 0, n)$ bernilai benar jika dan hanya jika elemen-elemen A dari 0 hingga $n - 1$ terurut. Misalkan satu-satunya cara untuk mengubah A adalah dengan $\text{swap}(A, i, j)$, yang menukar elemen $A[i]$ dan $A[j]$, maka $\text{permu}(A, A_0)$ akan tetap bernilai benar selama algoritma berlangsung. Artinya, triple tersebut

$$\{permu(A, A_0)\} swap(A, i, j) \{permu(A, A_0)\}$$

selalu benar ($swap(A, i, j)$ mewakili kode “ $z := A[i]; A[i] := A[j]; A[j] := z$ ”). Dengan demikian, untuk kesederhanaan, kita mengasumsikan $swap(A, i, j)$ adalah satu-satunya cara untuk memodifikasi A dan menghilangkan $permu(A, A_0)$ dari pembahasan kita, dengan fokus pada $sorted(A, 0, n)$.

Secara intuitif, $sorted(A, i, j)$ benar jika dan hanya jika $A[i] \leq A[i + 1] \leq \dots \leq A[j - 1]$. Untuk tujuan verifikasi, pustaka program tidak memerlukan implementasi $sorted(A, i, j)$. Di sisi lain, programmer dan pembukti teorema perlu mengetahui arti $sorted(A, i, j)$, yang dapat didefinisikan secara rekursif sebagai berikut:

$$\begin{aligned} sorted(A, i, j) &= \text{true} \text{ if } i \geq j - 1 \\ sorted(A, i, j + 1) &\doteq sorted(A, i, j) \wedge A[j - 1] \leq A[j] \text{ if } i < j \end{aligned}$$

Contoh 9.4.2 Jika kita ingin menghasilkan invarian untuk algoritma pengurutan penyisipan, teknik pertama yang diterapkan adalah relaksasi konstan: ganti $sorted(A, 0, n)$ dengan $k \doteq n \wedge sorted(A, 0, k)$. Dengan mengambil $k \doteq n$ sebagai kondisi keluar, kita memperoleh invarian $k > 0 \wedge sorted(A, 0, k)$, di mana $k > 0$ berasal dari prasyarat $n > 0$, dan loop luarnya terlihat sebagai berikut:

```

{n > 0}
1  k := 1
   {I : k > 0 ∧ sorted(A, 0, k)}
2  while k ≠ n do
   ...
8    k := k + 1
9  od

```

Mudah dilihat bahwa kondisi keluar dari loop luar, yaitu $k = n$, akan memberi kita postcondition $sorted(A, 0, n)$.

Untuk menghasilkan invarian loop untuk loop dalam, kita menerapkan uncoupling dan aging pada $sorted(A, 0, k)$ dengan memperkenalkan variabel baru j , yang memecah $k + 1$ elemen pertama dari A menjadi dua daftar yang diurutkan: $A[0], \dots, A[j - 1]$, dan $A[j], \dots, A[k]$. Kita memperoleh invarian I' untuk loop dalam:

$$I' : 0 \leq j \wedge j \leq k \wedge sorted(A, 0, j) \wedge sorted(A, j, k + 1)$$

yang memiliki empat konjungsi. Dua konjungsi pertama adalah invarian pembatas. Jika kita memilih $\neg(j > 0 \wedge A[j - 1] > A[j])$ sebagai kondisi keluar dari loop dalam, bersama dengan dua konjungsi terakhir dari I' , kita dapat menunjukkan bahwa $sorted(A, 0, k + 1)$ bernilai benar ketika kondisi keluar bernilai benar:

```

      { $n > 0$ }
1    $k := 1$ 
      { $I : k > 0 \wedge \text{sorted}(A, 0, k)$ }
2   while  $k \neq n$  do
3      $j := k$ 
      { $I : 0 \leq j \wedge j \leq k \wedge \text{sorted}(A, 0, j) \wedge \text{sorted}(A, j, k + 1)$ }
4     while ( $j > 0 \wedge A[j - 1] > A[j]$ ) i
5        $\text{swap}(A, j, j - 1)$ 
6        $j := j - 1$ 
7     od
8      $k := k + 1$ 
9   od

```

Mudah untuk melihat bahwa $\text{sorted}(A, 0, j)$ dan $\text{sorted}(A, j, k + 1)$ bernilai benar ketika $j \doteq k$, karena $\text{sorted}(A, 0, k)$ berasal dari I dan $\text{sorted}(A, k, k + 1)$ dari definisi sorted . Untuk menunjukkan bahwa $0 \leq j \wedge j \leq k$ bernilai benar ketika $j \doteq k$, kita perlu mengetahui $k \geq 0$.

Itulah mengapa kita telah menambahkan $k > 0$ sebagai invarian pembatas ke dalam I dari loop luar. Jika ini belum dilakukan, kita akan melakukannya sekarang. Jadi, I' dapat dibuktikan bernilai benar sebelum memasuki loop dalam.

Untuk melihat bahwa I' adalah invarian untuk loop dalam, di dalam loop kedua, $0 \leq j - 1$ bernilai benar karena kondisi loop $j > 0$; $\text{sorted}(A, 0, j - 1)$ bernilai benar karena $\text{sorted}(A, 0, j)$, $\text{sorted}(A, j - 1, k + 1)$ bernilai benar karena $\text{sorted}(A, j, k + 1)$ dan $A[j - 1] \leq A[j]$ (setelah pertukaran). Dengan demikian, I' memang merupakan invarian loop.

Kondisi keluar dari loop dalam adalah $j \leq 0 \vee A[j - 1] \leq A[j]$. Jika $j \leq 0$ bernilai benar, dari $0 \leq j$, kita tahu $j \doteq 0$; oleh karena itu, $\text{sorted}(A, 0, k + 1)$ bernilai benar dari $\text{sorted}(A, j, k + 1)$. Jika $A[j - 1] \leq A[j]$ benar, dari $\text{sorted}(A, 0, j)$, $A[j - 1] \leq A[j]$, dan $\text{sorted}(A, j, k + 1)$, kita juga menyimpulkan bahwa $\text{sorted}(A, 0, k + 1)$ benar. Setelah eksekusi baris 8, $k := k + 1$, kita sampai pada invarian loop luar, yaitu $k > 0 \wedge \text{sorted}(A, 0, k)$.

Memilih I' sebagai invarian loop dalam dengan kondisi keluar $j \leq 0 \vee A[j - 1] \leq A[j]$ menghasilkan algoritma insertion sort seperti yang disajikan pada contoh di atas. Jika kita memilih invarian loop yang berbeda dengan kondisi keluar yang tepat, kita dapat memperoleh algoritma *selection sort*. Algoritma-algoritma ini dianggap sebagai contoh "sintesis program," jika invarian dapat dihasilkan secara otomatis.

Contoh 9.4.3 Misalkan kita diminta untuk menulis program yang “menyimpan dalam y nilai maksimum dari x dan y .” Kondisi pascanya tentu saja adalah T . Asumsikan kondisi pranya adalah T , apa kode C dalam $y \doteq \max(x, y)$? Faktanya, $\{T\} C \{y \doteq \max(x, y)\} C$ tidak unik. Misalnya, beberapa pilihan adalah sebagai berikut:

- **if** $x > y$ **then** $y := x$ **fi**

- **if** $x > y$ **then** $x := y$ **fi**
- $y := x$
- $x := y$

Kode mana yang akan menjadi kode yang “benar”? Sehubungan dengan postcondition, keempat kode di atas semuanya benar. Jika maksudnya adalah bahwa y berisi nilai maksimum dari nilai asli x dan y , dalam hal ini, kita membutuhkan variabel bantu, seperti x_0 dan y_0 , untuk menunjukkan nilai asli x dan y . Postcondition yang benar adalah $y \doteq \max(x_0, y_0)$, dan kode pertama di atas benar sehubungan dengan postcondition ini. Jika kita bertanya lebih lanjut bahwa x dan y menyimpan nilai asli x dan y , maka postcondition yang benar adalah $y = \max(x_0, y_0) \wedge ((x = x_0 \wedge y = y_0) \vee (x = y_0 \wedge y = x_0))$, dan tidak ada kode di atas yang benar.

Contoh sederhana di atas menggambarkan bahwa ada banyak kode untuk mencapai postcondition yang sama; postcondition harus dipilih dengan cermat untuk mencerminkan persyaratan secara tepat. Ketika invarian loop dibutuhkan, kompleksitas sintesis program meningkat secara eksponensial. Karena terlalu banyak pilihan untuk membuat invarian loop yang tepat, sehingga menghasilkan berbagai algoritma, sintesis program tetap menjadi tugas yang sangat menantang.

Memilih Bahasa yang Baik untuk Asertasi

Asertasi informal bersifat intuitif, tetapi mampu mengekspresikannya secara formal bahkan lebih baik. Mencapai tujuan ini dan, secara lebih umum, terus menjadikan asertasi semakin umum dalam pengembangan perangkat lunak membutuhkan notasi yang nyaman, jelas, dan ekspresif untuk mengekspresikan invarian loop dan asertasi program lainnya.

Apa bahasa dasar yang baik untuk asertasi? Sejauh ini, asertasi ditulis dalam bahasa orde pertama. Apakah kita membutuhkan bahasa logika orde tinggi? Kita harus memastikan bahwa konvensi apa pun untuk asertasi kompatibel dengan bahasa yang digunakan untuk prasyarat dan pascakondisi. Keputusan bahasa dasar ini menyisakan pertanyaan tentang tingkat ekspresivitas asertasi.

Untuk bahasa orde pertama, ada banyak cara untuk menentukan pernyataan, karena bahasa orde pertama sangat ekspresif, dan kita dapat menggunakan berbagai predikat dan fungsi untuk menentukan properti yang sama. Setidaknya ada dua kemungkinan:

- Mengizinkan pernyataan, khususnya postkondisi dan invarian loop, untuk menggunakan fungsi dan predikat yang didefinisikan menggunakan beberapa mekanisme yang sesuai (seringkali, konstruksi deklarasi fungsi bahasa pemrograman) untuk mengekspresikan properti tingkat tinggi berdasarkan teori domain yang mencakup kekhususan area aplikasi. Kita menyebut ini pendekatan teori domain.
- Melarang kemungkinan sebelumnya, yang mengharuskan pernyataan selalu diekspresikan dalam hal konstruksi bahasa pernyataan, tanpa fungsi. Kita menyebut ini pendekatan pernyataan atomik.

Contoh 9.4.4 Contoh 9.2.8 adalah contoh yang baik dari pendekatan teori domain. Dalam contoh tersebut, kita mendefinisikan fungsi $arrMax(n, A)$, yang mengembalikan elemen maksimal di antara n elemen pertama dalam array A , dalam domain array. *Postcondition* tersebut secara singkat dijelaskan oleh $m \doteq arrMax(n, A)$. Dengan menggunakan pendekatan *assertion* atomik, *postcondition* yang sama akan menjadi sebagai berikut:

$$(\exists x \ 0 \leq x < n \wedge m \doteq A[x]) \wedge \forall y \ 0 \leq y < n \rightarrow m \geq A[y]$$

di mana bagian pertama menyatakan bahwa m adalah elemen dari A dan bagian kedua menyatakan bahwa tidak ada elemen A yang lebih besar dari m . Pernyataan serupa dapat ditulis untuk invarian loop.

Bahkan untuk contoh sesederhana itu, keterbatasan pendekatan pernyataan atomik jelas: pernyataan menjadi panjang dan rumit; pernyataan tersebut mengandung kuantifikasi yang sulit bagi pembukti teorema. Secara umum, pendekatan ini mengharuskan kembali ke konstruksi logika dasar setiap kali, dan tidak dapat diskalakan. Pengalaman masa lalu telah menunjukkan bahwa pendekatan pernyataan atomik tidak berguna.

Pendekatan teori domain berarti bahwa, sebelum upaya apa pun untuk menalar tentang suatu algoritma, kita harus mengembangkan model yang sesuai dari domain yang mendasarinya dengan mendefinisikan konsep yang sesuai seperti pembagi persekutuan terbesar untuk algoritma pada bilangan bulat (Contoh 9.4.1) dan menetapkan teorema yang relevan (misalnya, $x > y \rightarrow gcd(x, y) \doteq gcd(x - y, y)$).

Konsep dan teorema ini hanya perlu dikembangkan sekali untuk setiap domain aplikasi yang diminati, bukan baru untuk setiap program di atas domain tersebut. Program-program tersebut kemudian dapat menggunakan fungsi-fungsi yang sesuai dalam pernyataan mereka. Pendekatan teori domain memanfaatkan mekanisme abstraksi standar matematika. Satu-satunya kelemahan praktisnya, untuk pernyataan yang tertanam dalam bahasa pemrograman, adalah bahwa fungsi-fungsi pada suatu domain (seperti gcd) harus berasal dari beberapa pustaka dan, jika ditulis dalam bahasa pemrograman, harus memenuhi batasan-batasan yang ketat.

Pernyataan tanpa kuantifikasi lebih mudah untuk penalaran otomatis. Namun, pendekatan teori domain memiliki keterbatasannya sendiri: pendekatan ini membutuhkan beberapa fungsi (misalnya, $arrMax$) yang tidak dibutuhkan oleh program. Beberapa properti dari fungsi-fungsi ini mungkin dibutuhkan dalam pembuktian pernyataan, dan pembuktian properti ini mungkin membutuhkan alat penalaran lain (misalnya, penalaran induktif). Misalnya, dalam Contoh 9.2.8, kita mungkin membutuhkan properti bahwa $arrMax(n, A)$ adalah elemen dari A ketika $n > 0$.

Kemajuan dalam pendekatan teori domain, baik secara teoritis maupun di sisi rekayasa, tetap penting. Masih banyak yang harus dilakukan untuk membawa alat-alat tersebut ke tingkat di mana alat-alat tersebut dapat diintegrasikan dalam lingkungan pengembangan

standar dan secara rutin menyarankan pernyataan, termasuk invarian, dengan mudah dan benar, setiap kali seorang programmer menulis postcondition.

Alat Verifikasi untuk Bahasa Konvensional

Verifikasi formal telah lama menjadi bidang penelitian aktif baik di akademisi maupun industri. Banyak sistem verifikasi formal telah dikembangkan dengan keberhasilan yang luar biasa. Di sini, kami memilih beberapa di antaranya untuk bahasa pemrograman konvensional:

- *Verified Software Toolchain*: Alat ini dikembangkan di Universitas Princeton dengan peserta dari seluruh dunia. Alat ini menyediakan bahasa C yang dapat diverifikasi dan memastikan dengan bukti yang diperiksa mesin bahwa pernyataan yang diklaim benar-benar berlaku dalam program C. Alat ini mencakup penganalisis statis untuk memeriksa pernyataan, mengoptimalkan kompiler untuk menerjemahkan program C ke bahasa mesin, dan menyediakan pustaka C yang terverifikasi.
- *KeY*: Alat KeY awalnya dikembangkan di Institut Teknologi Karlsruhe Jerman dan bersama-sama oleh beberapa lembaga Eropa. Alat ini mencakup kumpulan alat perangkat lunak yang mendukung pemrograman Java yang terverifikasi di Eclipse IDE (lingkungan pengembangan terintegrasi). Alat ini menerima spesifikasi yang ditulis dalam Java Modeling Language (JML) untuk program Java. Spesifikasi ini diubah menjadi teorema logika dinamis dan kemudian dibuktikan secara interaktif (yaitu, dengan pengguna) atau secara otomatis. Upaya pembuktian yang gagal dapat digunakan untuk debugging yang lebih efisien atau pengujian berbasis verifikasi. Telah ada beberapa perluasan KeY untuk verifikasi program C atau sistem hibrida.

9.5 LATIHAN SOAL

1. Untuk $i = 1, 2, \dots, 5$ dalam program berikut, silakan hasilkan prasyarat P_i dan kondisi tengah dengan aturan Hoare dari Q_i dan tunjukkan detail setiap langkah pembuatannya:
 - (a) $\{P_1\} \text{ if } (x > y) \ z := x \ \text{else} \ z := y \ \{Q_1 : z \doteq \max(x, y)\}$
 - (b) $\{P_2\} \ x := x + 1; \ y := y + x \ \{Q_2 : y \doteq x(x + 1)/2\}$
 - (c) $\{P_3\} \ y := y + x; \ x := x - 1 \ \{Q_3 : c - y \doteq x(x + 1)/2\}$
 - (d) $\{P_4\} \ y := y + 1; \ z := x * z \ \{Q_4 : z \doteq x^y\}$
 - (e) $\{P_5\} \text{ if } (y \% 2 = 1) \ y := y - 1; \ z := x * z \ \text{else} \ x := x * x; \ y := y/2 \ \{Q_5 : x^y z \doteq c\}$
2. Buktikan secara formal bahwa tiga pernyataan berikut ini benar dengan memilih invarian loop yang tepat:

$\{x \doteq 0\} \ \text{while } (x < 100) \ \text{do } x := x + 1 \ \text{od } \{x \doteq 100\}$

3. Diberikan dua program berannotasi berikut, berikan invarian loop dan buktikan kebenaran totalnya dengan logika Hoare. Untuk setiap langkah penyederhanaan yang digunakan dalam pembuktian, nyatakan dengan jelas hukum aritmatika/aljabar/logika mana yang digunakan untuk penyederhanaan:


```

{A :  $a \geq 0$ }
int  $x := 0$ ; int  $y := 0$ 
while  $x < a$  do
     $x := x + 1$ 
     $y := y + x$ 
{B :  $y \doteq a(a + 1)/2$ }

```

```

{A :  $a \geq 0$ }
int  $x := a$ ; int  $y := 0$ 
while  $x \neq 0$  do
     $y := y + x$ 
     $x := x - 1$ 
{B :  $y \doteq a(a + 1)/2$ }

```

4. Ulangi latihan sebelumnya untuk dua program berikut untuk menghitung fungsi daya:

```

{A : int  $b \geq 0$ }
int  $x := a$ ; int  $y := 0$ ; int  $z := 1$ 
while  $y \neq b$  do
     $y := y + 1$ 
     $z := x * z$ 
{B :  $z \doteq a^b$ }

```

```

{A : int  $b \geq 0$ }
int  $x := a$ ; int  $y := b$ ; int  $z := 1$ 
while  $y \neq 0$  do
    if ( $y \% 2 = 1$ ) // i.e.,  $y$  is odd
         $y := y - 1$ ;  $z := x * z$ 
    else  $x := x * x$ ;  $y := y / 2$ 
{B :  $z \doteq a^b$ }

```

5. Untuk keempat program dalam dua latihan sebelumnya, jalankan algoritma VC dengan invarian loop yang tepat dan tunjukkan kondisi verifikasi yang dikembalikan dari postkondisi.
6. Diberikan sebuah array A yang terdiri dari n bilangan bulat, masalah Max-Sum adalah untuk menemukan jumlah maksimum elemen dalam setiap sub-array A yang berurutan dan tidak kosong. Misalnya, jika $A = [2, -5, 8, -4, 6, -3, -1, 2]$, maka solusinya adalah 10, yang merupakan jumlah dari sub-array $[8, -4, 6]$. Berikan postkondisi formal untuk program berikut dan buktikan secara formal bahwa program tersebut dapat menyelesaikan masalah Max-Sum.

7. Ulangi latihan sebelumnya menggunakan implementasi Prolog dari algoritma *VC*, dan tunjukkan kode Prolog dan hasil yang dikembalikan:

```
{A[1..n] : array of integers,  $n \geq 1$ }  
int x := 2; int y := A[1]; int z := A[1]  
while  $x \leq n$  do  
     $y := \max(y + A[x], A[x])$   
     $z := \max(y, z)$   
     $x := x + 1$   
{z is the solution of Max – Sum }
```

8. Pilih invarian loop yang berbeda dengan kondisi keluar yang tepat untuk loop dalam pada Contoh 9.4.2, sehingga algoritma yang dihasilkan adalah “selection sort.”

BAB 10

LOGIKA TEMPORAL

Pada bab sebelumnya, kita telah menunjukkan bahwa logika Hoare adalah kerangka kerja untuk verifikasi perangkat lunak: logika ini dapat menunjukkan bahwa suatu program dapat mencapai properti yang diinginkan setelah eksekusinya. Namun, ketika suatu program melibatkan banyak proses konkuren, verifikasinya menjadi jauh lebih kompleks, dan logika orde pertama tidak memadai.

Misalnya, ketika Anda menyetor uang tunai ke mesin teller otomatis (ATM), program yang berjalan di dalam ATM akan membaca saldo rekening bank Anda dan menambahkan jumlah uang tunai ke rekening Anda. Pada saat yang sama, program lain yang berjalan di dalam bank dapat menggunakan rekening yang sama untuk melunasi kartu kredit Anda. Kedua program tersebut tidak dapat dijalankan secara bersamaan; jika tidak, saldo rekening Anda mungkin akan salah. Memodifikasi data yang sama secara bersamaan oleh banyak proses disebut "kondisi balapan" dalam pemrograman konkuren.

Untuk menghindari kondisi balapan, diperlukan mekanisme penguncian agar suatu rekening dapat diakses ketika rekening tersebut bebas (tidak terkunci). Karena mekanisme penguncian, dua program dapat mengalami keadaan "kebuntuan" (deadlock). Misalnya, ketika Anda dan teman Anda saling mengirim uang pada saat yang bersamaan, satu program mengunci akun A dan meminta akun B, dan program lain mengunci akun B dan meminta akun A. Dalam kasus seperti itu, kita tidak dapat membuktikan penghentian program dengan menghitung langkah. Untuk memverifikasi bahwa program bekerja dengan benar tanpa kondisi persaingan atau kebuntuan, kita membutuhkan formalisme untuk mengekspresikan waktu, seperti "dua proses tidak dapat mengakses data penting pada saat yang bersamaan," atau "program akan berakhir pada akhirnya."

Logika orde pertama adalah logika yang sangat baik untuk banyak aplikasi seperti yang telah kita demonstrasikan di Bab 5–9. Apa yang disebut paradoks suhu menunjukkan bahwa tidak mudah untuk mengekspresikan waktu dalam logika orde pertama dengan cara yang sederhana. Misalkan t adalah variabel untuk suhu saat ini, seperti $t = 30$ untuk menunjukkan bahwa suhu saat ini adalah 30° . Misalkan $p(t)$ menunjukkan "suhu t sedang naik," maka $p(30)$ adalah rumus yang terbentuk dengan baik. Namun, interpretasi $p(30)$ tidak bermakna: "suhu 30 sedang naik."

Untuk menyatakan bahwa suhu berubah seiring waktu dalam logika orde pertama, kita dapat mendefinisikan suhu sebagai fungsi $t(x)$, di mana x adalah variabel waktu. Untuk menyatakan "suhu t meningkat," kita memiliki $(s > 0) \rightarrow t(x + s) > t(x)$. Dalam logika proposisional, misalkan p menyatakan "suhu sedang" dan q menyatakan "suhu sedang." Untuk menyatakan pernyataan bahwa "suhu berubah dari sedang menjadi panas," kita dapat menggunakan $p \wedge \circ (q)$, bersama dengan beberapa sifat umum dari p dan q . Di sini, $\circ$

menunjukkan “apa yang akan benar selanjutnya” dan merupakan salah satu operator temporal yang kita pelajari dalam bab ini.

Dalam logika, istilah *logika temporal* adalah sistem aturan dan simbolisme untuk merepresentasikan dan menalar tentang waktu dan informasi temporal, serta representasi formalnya. Penerapan logika temporal mencakup penggunaannya sebagai formalisme untuk mengklarifikasi isu-isu filosofis tentang waktu, sebagai kerangka kerja untuk mendefinisikan semantik ekspresi temporal dalam bahasa alami, dan sebagai bahasa untuk mengkodekan pengetahuan temporal dalam kecerdasan buatan.

Logika temporal memainkan peran penting dalam verifikasi formal program konkuren, di mana ia digunakan untuk menentukan persyaratan program komputer dan untuk melakukan analisis formal dan verifikasi eksekusi program. Misalnya, seseorang mungkin ingin mengatakan bahwa setiap kali permintaan dibuat, akses ke suatu sumber daya pada akhirnya diberikan, tetapi tidak pernah diberikan secara bersamaan kepada dua pihak yang membuat permintaan. Pernyataan seperti itu dapat dengan mudah diungkapkan dalam logika temporal.

Pertimbangkan pernyataan "Saya memiliki akun A." Meskipun maknanya tidak berubah seiring waktu, nilai kebenaran pernyataan tersebut dapat bervariasi seiring waktu. Terkadang benar, dan terkadang salah, tetapi tidak pernah benar dan salah secara bersamaan. Dalam logika temporal, suatu pernyataan dapat memiliki nilai kebenaran yang bervariasi seiring waktu. Perlakuan terhadap nilai kebenaran dari rumus yang sama dari waktu ke waktu ini membedakan logika temporal dari logika klasik.

10.1 PENDEKATAN DARI LOGIKA MODAL

Modal adalah ekspresi (seperti "pasti" atau "mungkin") yang digunakan untuk mengkualifikasi kebenaran suatu pernyataan. Logika modal adalah studi tentang perilaku deduktif dari ekspresi yang mengandung frasa seperti "pasti bahwa" atau "mungkin bahwa." Logika modal telah diperluas untuk mencakup logika kepercayaan, kala, dan ekspresi temporal di mana "pasti" (modal) diinterpretasikan sebagai "selalu" (temporal) dan "mungkin" sebagai "akhirnya." Pemahaman tentang logika modal dan perluasannya sangat berharga dalam analisis formal argumen filosofis, di mana ekspresi dari keluarga modal bersifat umum dan membingungkan.

Seperti kebanyakan sistem logika formal, logika modal serta logika temporal dapat didefinisikan sebagai perluasan dari logika klasik. Di sini, untuk kesederhanaan, kami menyajikan logika temporal (TL) dengan menambahkan dua operator modal, $\Box$ dan $\Diamond$, ke logika proposisional.

Operator Modal

Operator modal pertama, $\Box$, menunjukkan "selalu" dan mengambil formula sebagai argumennya. Misalnya, jika p menunjukkan "hari ini hujan," maka $\Box p$ adalah formula TL, dan makna intuitifnya adalah "akan hujan setiap hari."

Operator kedua, $\Diamond$, menunjukkan “pada akhirnya” dan juga mengambil formula sebagai argumennya. Jadi, jika p menunjukkan “hari ini hujan,” $\Diamond p$ adalah formula TL, dan makna intuitifnya adalah “suatu hari nanti akan hujan.”

Secara informal, $\Box$ menyatakan properti universal di masa depan, sedangkan $\Diamond$ menyatakan properti eksistensial di masa depan. Misalnya, untuk menentukan pernyataan “jika permintaan dibuat untuk mencetak file, file tersebut pada akhirnya akan dicetak,” pernyataan tersebut dapat dilambangkan dengan $\Box(r \rightarrow \Diamond p)$, di mana r berarti “permintaan dibuat” dan p berarti “file dicetak.”

Kedua operator, $\Box$ dan $\Diamond$, muncul hampir di setiap logika modal, di mana $\Box$ biasanya dibaca sebagai “pasti” dan $\Diamond$ sebagai “mungkin.” Kami meminjamnya dari logika modal klasik untuk TL dan memberinya makna temporal, yaitu, “selalu” dan “pada akhirnya.”

Dengan menggunakan tata bahasa BNF, rumus-rumus TL dapat didefinisikan secara formal sebagai berikut:

Definisi 10.1.1 Misalkan op adalah operator logika biner dari logika proposisional dan V_P adalah variabel proposisional yang digunakan dalam aplikasi saat ini, maka rumus-rumus TL dapat didefinisikan dengan tata bahasa BNF berikut:

$$\begin{aligned} \langle op \rangle &::= \wedge \mid \vee \mid \rightarrow \mid \oplus \mid \leftrightarrow \\ \langle V_P \rangle &::= p \mid q \mid r \mid s \mid t \\ \langle Formulas \rangle &::= \top \mid \perp \mid \langle V_P \rangle \mid \neg \langle Formulas \rangle \mid (\langle Formulas \rangle \langle op \rangle \langle Formulas \rangle) \mid \\ &\quad \Box \langle Formulas \rangle \mid \Diamond \langle Formulas \rangle \end{aligned}$$

Berikut beberapa contoh rumus TL:

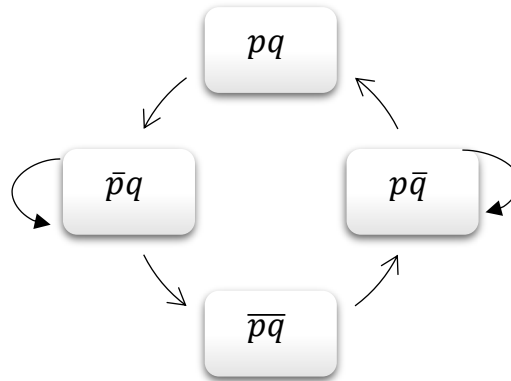
$$(p \wedge \Box \neg q), \Diamond(r \rightarrow (\Diamond p \wedge \neg q)), (\Box(p \wedge \neg q) \vee \Diamond r), (\Box(p \wedge \neg q) \vee \Diamond r)$$

Perhatikan perbedaan dua rumus terakhir, di mana $\Box$ berlaku untuk seluruh rumus pada rumus pertama dan hanya untuk argumen pertama dari $\vee$ pada rumus kedua.

Semantik Kripke

Semantik formal konvensional dari logika modal disebut semantik Kripke, juga dikenal sebagai kerangka Kripke, semantik relasional, atau semantik kerangka. Semantik Kripke adalah semantik formal untuk sistem logika modal yang diciptakan pada akhir tahun 1950-an oleh Saul Kripke dan Andre Joyal. Semantik ini diadaptasi untuk logika temporal dan sistem nonklasik lainnya. Pengembangan semantik Kripke merupakan terobosan dalam teori logika nonklasik, karena teori model logika tersebut hampir tidak ada sebelum Kripke.

Seperti yang telah kita lihat dari Bab 2, interpretasi dari rumus proposisional adalah pemetaan variabel proposisional ke nilai kebenaran. Dengan n variabel proposisional, kita memiliki tepat 2^n interpretasi yang berbeda. Misalnya, jika kita memiliki dua variabel proposisional, katakanlah p dan q , maka ada empat interpretasi, $I_1 = pq$ (artinya $I_1(p) = 1$, $I_1(q) = 1$), $I_2 = p\bar{q}$, $I_3 = \bar{p}q$, dan $I_4 = \bar{p}\bar{q}$. Menggunakan notasi dari Bab 2, $V_P = \{p, q\}$ dan $All_P = \{pq, p\bar{q}, \bar{p}q, \bar{p}\bar{q}\}$. Semantik Kripke didasarkan pada relasi biner atas All_P . Perhatikan bahwa setiap relasi biner R atas S dapat direpresentasikan oleh graf berarah $G = (S, R)$.



Gambar 10.1 Tampilan grafis dari kerangka Kripke pada dua variabel

Definisi 10.1.2 Diberikan himpunan variabel proposisional V_p , kerangka Kripke adalah graf berarah $K = (S, R)$, di mana S , yang disebut keadaan, adalah himpunan bagian tak kosong dari All_p , himpunan semua interpretasi atas V_p , dan R adalah relasi biner atas S .

Contoh 10.1.3 Misalkan $V_p = \{p, q\}$, $S = All_p = \{pq, p\bar{q}, \bar{p}q, \bar{p}\bar{q}\}$, dan $R = \{(pq, \bar{p}q), (pq, \bar{p}\bar{q}), (p\bar{q}, p\bar{q}), (\bar{p}q, \bar{p}\bar{q}), (\bar{p}\bar{q}, p\bar{q})\}$. Maka, $G = (S, R)$, yang ditampilkan pada Gambar 10.1, adalah kerangka Kripke.

Harus dipahami bahwa dalam suatu aplikasi, setiap rumus TL menggunakan variabel proposisional dari V_p dan setiap keadaan dalam kerangka Kripke $K = (S, R)$ adalah interpretasi dari V_p . Jika $(w, w') \in R$, kita katakan w' adalah *penerus* w di K .

Definisi 10.1.4 Diberikan kerangka Kripke $K = (S, R)$ dan keadaan $s \in S$, rumus TL C benar di s dari K , yang dilambangkan dengan $K, s \models C$, jika salah satu dari kondisi berikut benar:

1. C is T or 1.
2. $C \in V_p, s(C) = 1$.
3. C is $\neg A, K, s \not\models A$.
4. C is $A \wedge B, K, s \models A$ and $K, s \models B$.
5. C is $A \vee B, K, s \models A$ or $K, s \models B$.
6. C is $A \rightarrow B, K, s \models \neg A \vee B$.
7. C is $\Box A$, for every successor s' of $s, K, s' \models A$.
8. C is $\Diamond A$, for some successor s' of $s, K, s' \models A$.

Jika tidak, kita katakan C salah di s dari K .

Tampaknya, definisi di atas adalah definisi rekursif, dan kita dapat menulis algoritma rekursif untuk menghitung nilai $K, s \models A$.

Contoh 10.1.5 Perhatikan kerangka Kripke pada Gambar 10.1 dan misalkan A adalah $\Box \neg p$. Maka, A benar di $s = pq$ karena satu-satunya penerus s adalah $s' = \bar{p}q$ dan $\neg p$ benar di s' .

Misalkan B adalah $p \wedge q \rightarrow \Box \neg p$. B benar di $s = \bar{p}q$ atau $s = \bar{p}\bar{q}$, karena p salah di s . B benar di $s = p\bar{q}$ karena q salah di s . B juga benar di $s = pq$ karena $A = \Box \neg p$ benar di pq . Artinya, B benar di setiap keadaan kerangka.

Dengan menggunakan analisis yang sama, kita dapat menunjukkan bahwa $p \wedge q \rightarrow \Box \Box \neg p$ benar di setiap keadaan. Namun, $p \wedge q \rightarrow \Box \Box \Box \neg p$ salah di $s = pq$. Dalam logika modal, kerangka Kripke menentukan transisi keadaan (juga disebut dunia). "Harus" berarti "di semua keadaan berurutan (penerus)" dan "mungkin" berarti "di salah satu penerus." Demikian pula, dalam TL, rumus $\Box A$ benar di keadaan saat ini jika A benar di setiap penerus; $\Diamond A$ benar di keadaan saat ini jika A benar di salah satu penerus. Perhatikan bahwa jika keadaan saat ini tidak memiliki penerus, maka $\Box A$ benar dan $\Diamond A$ salah dalam keadaan saat ini untuk formula A apa pun.

Definisi 10.1.6 Diberikan kerangka Kripke $K = (S, R)$, formula TL A benar di K , ditulis $K \models A$, jika untuk setiap keadaan $s \in S$, $K, s \models A$, dan kita katakan K adalah model Kripke dari A .

A dapat dipenuhi di TL jika A memiliki model Kripke. Artinya, ada kerangka Kripke K sedemikian sehingga $K \models A$.

A valid di TL, ditulis $\models A$, jika A benar di setiap kerangka Kripke. Artinya, setiap kerangka Kripke adalah model dari A .

Contoh 10.1.5 menunjukkan bahwa baik $p \wedge q \rightarrow \Box \neg p$ dan $p \wedge q \rightarrow \Box \Box \neg p$ dapat dipenuhi karena keduanya benar di kerangka Kripke pada Gambar 10.1. Namun, kerangka ini bukanlah model dari $p \wedge q \rightarrow \Box \Box \Box \neg p$.

Diberikan sebuah formula TL A , misalkan $M(A)$ menyatakan himpunan semua model Kripke dari A .

Definisi 10.1.7 Diberikan dua formula TL A dan B , A dan B setara, ditulis $A \equiv B$, jika $M(A) = M(B)$. Kita katakan A menyiratkan B , atau B adalah konsekuensi logis dari A , ditulis $A \models B$, jika $M(A) \subseteq M(B)$.

Diberikan himpunan V_p dari n variabel proposisional, jumlah pilihan untuk S dalam kerangka Kripke sangat besar ($2^{2^n} - 1$) dan jumlah pilihan untuk R juga sangat besar ($2^{|S|^2}$ untuk setiap S). Oleh karena itu, tidak mungkin untuk memeriksa apakah formula TL valid dengan menghitung semua kerangka. Namun, semua tautologi dalam logika proposisional valid dalam TL, dan semua hasil utama dari logika proposisional masih berguna dalam TL.

Teorema 10.1.8

- (a) **(Substitusi)** Setiap contoh rumus TL yang valid juga valid dalam TL. Artinya, jika p adalah variabel proposisional dalam A , maka $A[p \sim B]$ valid setiap kali A valid, di mana B adalah rumus TL apa pun.
- (b) **(Substitusi kesetaraan)** Untuk setiap rumus TL A, B , dan C , di mana B adalah subrumus dari A , dan $B \equiv C$, maka $A \equiv A[B \sim C]$.
- (c) **(Kesetaraan Logis)** Diberikan dua rumus TL A dan B , $\models A \leftrightarrow B$ jika dan hanya jika $A \equiv B$.
- (d) **(Keterkaitan)** Diberikan dua rumus TL A dan B , $\models A \rightarrow B$ jika dan hanya jika $A \models B$.

Bukti teorema di atas serupa dengan bukti untuk logika proposisional dan dihilangkan di sini.

Di bawah ini kami menunjukkan beberapa contoh bukti dalam TL. Nantinya, kami akan memberikan metode pembuktian berdasarkan tabel semantik.

Teorema 10.1.9 (Dualitas) $\neg \Box p \equiv \Diamond \neg p$.

Bukti Untuk setiap kerangka Kripke $K = (S, R)$, jika K adalah model dari $\neg \Box p$, maka untuk setiap keadaan $s \in S$, $\neg \Box p$ bernilai benar di s , atau setara dengan, $\Box p$ bernilai salah di s . Itu berarti ada penerus s' dari s sedemikian sehingga p bernilai salah di s' , atau setara dengan, $\neg p$ bernilai benar di s' . Oleh karena itu, $\Diamond \neg p$ bernilai benar di s berdasarkan semantik $\Diamond$. Karena s bersifat sembarang, $\Diamond \neg p$ bernilai benar di setiap keadaan K . Dengan kata lain, K adalah model dari $\Diamond \neg p$.

Di sisi lain, jika K bukan model dari $\neg \Box p$, maka ada keadaan $s \in S$ sedemikian sehingga $\neg \Box p$ bernilai salah (sehingga $\Box p$ bernilai benar) di s . Berdasarkan definisi, s tidak memiliki penerus, atau untuk setiap penerus s' dari s , p harus bernilai benar di s' . Oleh karena itu, jika s tidak memiliki penerus, maka $\Diamond \neg p$ bernilai salah berdasarkan definisi. Jika s memiliki, karena $\neg p$ salah di setiap penerus s , maka $\Diamond \neg p$ harus salah di s . Jadi, K tidak dapat menjadi model dari $\Diamond \neg p$. Karena K bersifat sembarang, $\neg \Box p$ dan $\Diamond \neg p$ harus memiliki himpunan model yang sama.

Dengan menerapkan substitusi dan kesetaraan, kita dapat dengan mudah menyimpulkan dari teorema di atas bahwa $\Box \neg p \equiv \neg \Diamond p$ dan $\Box \neg p \leftrightarrow \neg \Diamond p$ adalah valid.

Contoh 10.1.10 Kita dapat membuktikan bahwa $(\Box p \rightarrow p) \equiv (\neg p \rightarrow \Diamond \neg p)$ dengan mengubah kedua sisi $\equiv$ menjadi rumus yang sama menggunakan relasi kesetaraan yang diketahui:

$$\begin{aligned}\Box p \rightarrow p &\equiv \neg \Box p \vee p \\ &\equiv \Diamond \neg p \vee p\end{aligned}$$

$$\begin{aligned}\neg p \rightarrow \Diamond \neg p &\equiv \neg \neg p \vee \Diamond \neg p \\ &\equiv \Diamond \neg p \vee p\end{aligned}$$

Rumus $p \vee \Diamond \neg p$ adalah bentuk normal negasi (NNF) dari TL, dan kita akan membahas NNF nanti.

Teorema 10.1.11 (Distribusi) $\models \Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$.

Bukti $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$ secara logis setara dengan

$$(1) \quad \neg \Box(\neg p \vee q) \vee \neg \Box p \vee \Box q.$$

Dengan menerapkan teorema dualitas pada (1), kita memperoleh rumus ekuivalen lainnya:

$$(2) \quad \Diamond(p \wedge \neg q) \vee \Diamond \neg p \vee \Box q.$$

Untuk setiap kerangka Kripke $K = (S, R)$ dan setiap $s \in S$, jika (2) salah di s , yaitu, $\Diamond(p \wedge \neg q)$, $\Diamond \neg p$, dan $\Box q$ semuanya salah di s , maka s harus memiliki beberapa penerus (jika tidak, $\Box q$ benar di s). Karena $\Box q$ salah di s , harus ada penerus s' dari s sedemikian sehingga $\Box q$ salah di s' . p harus benar di s' karena $\Diamond \neg p$ salah di s . Oleh karena itu, $p \wedge \neg q$ benar di s' dan $\Diamond(p \wedge \neg q)$ benar di s , sebuah kontradiksi. Jadi, (2) harus benar di s dan dengan demikian benar di setiap kerangka Kripke.

Pembatasan dan Keterbatasan

Misalkan r menyatakan “hujan,” maka makna intuitif dari $\Box r$ adalah “selalu hujan.” Diberikan kerangka Kripke $K = (S, R)$, jika $\Box r$ benar dalam keadaan $s \in S$, kita tahu r akan benar dalam setiap penerus s . Berapakah nilai kebenaran r dalam keadaan s ? Jika $\Box$ berarti “selalu,” haruskah r benar sekarang dan kemudian di setiap keadaan, termasuk s ? Artinya, kita dapat bertanya bahwa $\Box A \rightarrow A$ adalah aksioma (rumus yang diasumsikan benar) dari TL. Kita mengatakan kerangka Kripke $K = (S, R)$ bersifat refleksif jika setiap keadaan $s \in S$ adalah penerus dari dirinya sendiri, yaitu, $(s, s) \in R$ (disebut loop dalam teori graf).

Teorema 10.1.12 (Refleksivitas) Kerangka Kripke K bersifat refleksif jika dan hanya jika K merupakan model dari $\Box A \rightarrow A$ untuk setiap formula A .

Bukti Misalkan $K = (S, R)$ bersifat refleksif. Jika $\Box A \rightarrow A$ salah di K , maka terdapat $s \in S$ sedemikian sehingga $\Box A$ benar tetapi A salah di s . Karena $\Box A$ benar, maka A benar di setiap penerus s . Namun, s juga merupakan penerus s karena K bersifat refleksif. Jadi, A juga benar di s . Ini merupakan kontradiksi dengan $\Box A \rightarrow A$ yang salah di K .

Di sisi lain, misalkan $K = (S, R)$ tidak bersifat refleksif. Pilih suatu keadaan $s \in S$ sedemikian sehingga s bukan penerus dirinya sendiri. Misalkan I_s adalah interpretasi

proposisional yang terkait dengan s . Jika kita memandang I_s sebagai konjungsi literal yang masing-masing benar di s , maka $\neg(I_s)$ adalah klausa yang salah di s . Klausul ini akan benar dalam setiap interpretasi (termasuk semua penerus s) selain s . Dengan demikian, $\Box\neg(I_s)$ benar dalam s . Misalkan A adalah $\neg(I_s)$, maka $\Box A \rightarrow A$ salah dalam s dan K bukan model dari $\Box A \rightarrow A$. Karena K sembarang, teorema tersebut berlaku.

Teorema di atas memainkan peran ganda: dengan membatasi kerangka Kripke pada kerangka refleksif, kita memperoleh validitas $\Box A \rightarrow A$ dalam kerangka ini; di sisi lain, jika $\Box A \rightarrow A$ diberikan sebagai aksioma, kita mengecualikan semua kerangka Kripke nonrefleksif sebagai kandidat model.

Kita tahu dari kerangka Kripke bahwa jika $\Box p$ benar dalam keadaan s saat ini, maka p akan benar di semua penerus s . Bagaimana dengan nilai kebenaran p dalam penerus dari penerus s ? Jika $\Box$ berarti "selalu," haruskah p juga benar di dalamnya? Pada dasarnya, kita meminta agar $\Box A \rightarrow \Box\Box A$ benar di setiap keadaan untuk setiap rumus A . Katakanlah kerangka Kripke $K = (S, R)$ bersifat transitif jika R bersifat transitif: untuk setiap keadaan u, v, w , $(u, v) \in R$ dan $(v, w) \in R$ menyiratkan bahwa $(u, w) \in R$. Teorema berikut memberi tahu kita tentang hubungan antara transitivitas dan rumus $\Box A \rightarrow \Box\Box A$.

Teorema 10.1.13 (Transitivitas) Suatu kerangka Kripke K bersifat transitif jika dan hanya jika K merupakan model dari $\Box A \rightarrow \Box\Box A$ untuk setiap rumus A .

Bukti Misalkan $K = (S, R)$ bersifat transitif. Jika $\Box A \rightarrow \Box\Box A$ salah di K , maka ada $s \in S$ sedemikian sehingga $\Box A$ benar tetapi $\Box\Box A$ salah di s . Hal ini menyiratkan bahwa harus ada penerus s' dari s dan penerus s'' dari s' sedemikian sehingga A salah di s'' . Karena A benar, maka A benar di setiap penerus s . Namun, s'' juga merupakan penerus dari s karena K bersifat transitif. Jadi, A juga benar di s'' . Ini merupakan kontradiksi dengan pernyataan $\Box A \rightarrow \Box\Box A$ yang salah di K .

Di sisi lain, misalkan $K = (S, R)$ tidak transitif. Harus ada tiga keadaan $s_1, s_2, s_3 \in S$ sedemikian sehingga s_2 merupakan penerus s_1 , s_3 merupakan penerus s_2 tetapi bukan penerus s_1 . Misalkan I_3 adalah interpretasi proposisional yang terkait dengan s_3 . Jika kita memandang I_3 sebagai konjungsi literal yang masing-masing bernilai benar di s_3 , maka $\neg(I_3)$ adalah klausa yang hanya bernilai salah di s_3 tetapi bernilai benar di setiap interpretasi (termasuk semua penerus s_1) selain s_3 . Misalkan A adalah $\neg(I_3)$, maka A bernilai salah di s_3 dan bernilai benar di setiap keadaan lainnya, dan $\Box A$ bernilai salah di s_2 dan bernilai benar di s_1 ; $\Box\Box A$ bernilai salah di s_1 . Oleh karena itu, $\Box A \rightarrow \Box\Box A$ salah di s_1 dan K bukan model dari $\Box A \rightarrow \Box\Box A$.

Dengan menggabungkan kedua teorema di atas, kita dapat menyimpulkan bahwa jika suatu kerangka Kripke bersifat refleksif dan transitif, maka $\Box A \leftrightarrow \Box\Box A$ akan benar di dalam kerangka ini.

Aturan *necessitasi* logika modal menyatakan bahwa jika A benar, maka $\Box A$ juga benar. Artinya, jika A dapat dibuktikan benar sekarang, bukti yang sama dapat digunakan di mana

saja. Aksioma untuk menyatakan aturan necessitasi adalah rumus $A \rightarrow \Box A$. Model Kripke dari $A \rightarrow \Box A$ adalah kerangka Kripke $K = (S, R)$ di mana satu-satunya penerus (jika ada) dari setiap keadaan adalah dirinya sendiri, yaitu, jika $(w, v) \in R$, maka $w = v$. Kerangka seperti itu disebut diskrit. Jelas, satu-satunya sisi yang diizinkan dalam kerangka diskrit adalah loop (yaitu, (w, w)), dan kerangka seperti itu tidak terlalu berguna. Karena beberapa loop mungkin hilang dalam kerangka diskrit, kerangka diskrit tidak selalu refleksif.

Selain reflektivitas, transitivitas, dan diskrit, ada banyak batasan lain pada relasi dalam kerangka Kripke, seperti simetri, kepadatan, dll. Ada aksioma yang sesuai dengan batasan-batasan ini.

Semantik berguna untuk logika hanya jika konsekuensi semantik mencerminkan padanan sintaksisnya, yaitu, hasil dari sistem pembuktian untuk logika tersebut. Sangat penting untuk mengetahui logika modal mana yang sah dan lengkap sehubungan dengan himpunan kerangka Kripke. Misalkan C adalah himpunan rumus yang menentukan pemenuhan beberapa sifat. Misalnya, C dapat berupa sifat refleksif (atau transitif). Kita mendefinisikan $T(C)$ sebagai himpunan semua rumus yang disiratkan oleh C . Sistem pembuktian dalam logika modal sah sehubungan dengan C jika setiap rumus yang terbukti benar termasuk dalam $T(C)$; sistem tersebut lengkap sehubungan dengan C jika setiap rumus dalam $T(C)$ dapat dibuktikan benar. Terdapat lebih dari selusin sistem pembuktian untuk logika modal, dan pembaca yang tertarik dapat merujuk ke buku teks tentang logika modal.

10.2 LOGIKA TEMPORAL LINIER

Biasanya, logika temporal memiliki kemampuan untuk bernalar tentang garis waktu, yang pada dasarnya linier. Semantik Kripke menggunakan graf berarah $K = (S, R)$ dan cocok untuk bernalar tentang beberapa garis waktu, karena setiap keadaan memungkinkan beberapa penerus. Kita dapat memberikan batasan pada R sehingga R bertindak seperti relasi linier: setiap keadaan memiliki paling banyak satu penerus. Batasan ini dapat bekerja dengan reflektivitas dengan memungkinkan setiap keadaan memiliki paling banyak satu penerus selain dirinya sendiri.

Namun, batasan ini tidak bekerja dengan baik dengan transitivitas karena transitivitas menginginkan suatu keadaan untuk mencakup penerus dari penerus sebagai penerusnya sendiri. Selain itu, jumlah keadaan dalam kerangka Kripke dibatasi oleh jumlah interpretasi proposisional. Kita tidak dapat memodelkan setiap garis waktu dengan panjang tak terbatas dengan kerangka Kripke, yang hanya menyediakan siklus dalam graf. Oleh karena itu, kita membutuhkan semantik yang berbeda dari kerangka Kripke untuk logika temporal linier. Logika temporal berdasarkan kerangka Kripke dapat dipandang sebagai logika temporal bercabang.

Logika temporal linier (LTL) atau logika temporal waktu linier adalah logika modal (temporal) dengan modalitas yang mengacu pada garis waktu yang unik. LTL pertama kali diusulkan untuk verifikasi formal program komputer oleh Amir Pnueli pada tahun 1977. Dalam LTL, seseorang dapat mengkodekan rumus tentang masa depan jalur, misalnya, suatu kondisi

pada akhirnya akan benar, atau suatu kondisi akan benar sampai peristiwa lain menjadi benar, dll. Untuk kesederhanaan, pengantar kami terbatas pada perluasan logika proposisional.

Seperti dalam TL, dalam LTL, $\Box$ diinterpretasikan sebagai "selalu" dan $\Diamond$ sebagai "pada akhirnya." Selain dua operator modal umum ini, kita memiliki operator ketiga $\circ$ yang diinterpretasikan sebagai tanda (atau langkah) "berikutnya". Seperti $\Box$ dan $\Diamond$, $\circ$ mengambil rumus sebagai argumen: jika r berarti "hari ini hujan," maka $\circ r$ berarti "besok akan hujan," jika kita menginterpretasikan bahwa satu tanda adalah satu hari. Operator $\circ$ merupakan bagian penting dari LTL, yang diciptakan untuk verifikasi formal program konkuren.

Namun, $\circ$ jarang digunakan dalam spesifikasi program konkuren, karena tidak banyak yang dikatakan tentang eksekusi program pada langkah selanjutnya. Lebih lanjut, kita menginginkan pernyataan kebenaran tentang program konkuren berlaku terlepas dari bagaimana proses interleaving memilih operasi selanjutnya. Oleh karena itu, properti hampir selalu dinyatakan dalam bentuk $\Box$ dan $\Diamond$, bukan dalam bentuk $\circ$.

Misalkan V_p adalah himpunan semua variabel proposisional yang digunakan dalam aplikasi LTL. Definisi formal dari rumus LTL diberikan sebagai tata bahasa BNF berikut:

$$\begin{aligned} \langle V \rangle &::= p \text{ if } p \in V_p \\ \langle op \rangle &::= \wedge \mid \vee \mid \rightarrow \mid \oplus \mid \leftrightarrow \\ \langle Formulas \rangle &::= \top \mid \perp \mid \langle V \rangle \mid \neg \langle Formulas \rangle \mid (\langle Formulas \rangle \langle op \rangle \langle Formulas \rangle) \mid \\ &\quad \Box \langle Formulas \rangle \mid \Diamond \langle Formulas \rangle \mid \circ \langle Formulas \rangle \end{aligned}$$

Berikut beberapa contoh formula LTL:

$$(\circ p \wedge \Box \neg q), \Diamond(\circ r \rightarrow (\Diamond p \wedge \neg q)), \Box((p \wedge \neg q) \vee \circ \Diamond r)$$

Garis Waktu sebagai Urutan Interpretasi

Semantik LTL akan didefinisikan oleh urutan interpretasi proposisional tak terbatas sehingga variabel proposisional memiliki nilai kebenaran tetap pada posisi mana pun (disebut keadaan) dalam urutan tersebut. Secara formal, urutan interpretasi π adalah pemetaan dari himpunan N bilangan asli ke himpunan semua interpretasi proposisional All_p :

$$\pi : N \rightarrow All_p$$

Untuk setiap $i \in N$, misalkan $\pi(i) = s_i$. Maka, π dapat ditulis sebagai urutan interpretasi:

$$\pi = s_0 s_1 s_2 \cdots$$

di mana $s_i = \pi(i) \in All_p$ untuk setiap $i \in N$. i disebut keadaan dari π dan s_i adalah interpretasi pada keadaan i . Sekarang, kita melihat bahwa konsep keadaan diubah dari interpretasi dalam TL menjadi bilangan asli dalam LTL. Dengan demikian, kita memiliki jumlah

keadaan yang tak terbatas dalam LTL tetapi hanya sejumlah keadaan yang terbatas dalam TL, dan dua keadaan LTL dapat memiliki interpretasi yang sama, yaitu, $\pi(i) = \pi(j)$ untuk $i \neq j$.

Untuk $i \in N$, kita menggunakan $\pi_{\geq i}$ untuk menunjukkan sufiks dari π dengan menghapus i interpretasi pertama dari π :

$$\pi_{\geq i} = t_0 t_1 t_2 \cdots$$

di mana $t_0 = s_i, t_1 = s_{i+1}, t_2 = s_{i+2}, \dots$ Jelas, $\pi = \pi_{\geq 0} = \pi(0)\pi_{\geq 1}$, di mana $\pi(0) = s_0$. Kita katakan $\pi_{\geq 1}$ adalah sufiks langsung dari π .

Definisi 10.2.1 Diberikan urutan interpretasi π , sebuah rumus LTL C dikatakan benar dalam π secara rekursif, yang dilambangkan dengan $\pi \models C$, jika salah satu dari kondisi berikut benar:

1. C is T or 1.
2. C is $\neg A$ and $\pi \not\models A$
3. C is $A \wedge B$, $\pi \models A$, and $\pi \models B$
4. C is $A \vee B$ and either $\pi \models A$ or $\pi \models B$.
5. C is $A \rightarrow B$ and $\pi \models \neg A \vee B$.
6. C is $\circ A$ and $\pi_{\geq 1} \models A$
7. C is $\Box A$ and for all $i \geq 0, \pi_{\geq i} \models A$.
8. C is $\Diamond A$ and there exists $i \geq 0, \pi_{\geq i} \models A$

Jika tidak, kita katakan C salah di π .

Dari definisi rekursif di atas, kita melihat bahwa suatu formula $\circ A$ benar dalam urutan interpretasi π jika dan hanya jika A benar di $\pi_{\geq 1}$, urutan yang diperoleh dengan menghapus item pertama dari π . Suatu formula $\Box A$ benar di π jika dan hanya jika A selalu benar di setiap sufiks π ; $\Diamond A$ benar di π jika dan hanya jika A akhirnya benar di salah satu sufiks π . Sebaliknya, dalam semantik Kripke untuk waktu percabangan, $\Box A$ benar pada keadaan saat ini jika dan hanya jika A benar di setiap keadaan penerus. Perhatikan bahwa definisi kebenaran di atas dapat diperluas ke urutan dengan panjang terbatas. Jika π adalah urutan kosong, untuk setiap formula LTL A , $\Box A$ benar di π tetapi $\circ A$ dan $\Diamond A$ salah di π .

Agar $\Diamond \Box p$ benar di π , kita perlu menemukan suatu keadaan di π sedemikian sehingga p benar di setiap interpretasi π dari keadaan tersebut. Artinya, hanya suatu awalan dari π yang dapat memuat sejumlah interpretasi terbatas yang memalsukan p ; p harus benar di setiap interpretasi yang tidak ada di awalan tersebut. Artinya, terdapat $i \geq 0$, untuk semua $j \geq i, \pi(j)(p) = 1$.

Agar $\Box \Diamond p$ benar dalam π , kita hanya perlu p benar dalam salah satu interpretasi dari sufiks π mana pun: untuk semua $i \geq 0$, terdapat $j \geq i, \pi(j)(p) = 1$. Dengan demikian, dalam bahasa orde pertama, π untuk $\Diamond \Box p$ memenuhi $\exists i \forall j R(i, j)$, sedangkan π untuk $\Box \Diamond p$

memenuhi $\forall i \exists j R(i, j)$, di mana $R(i, j)$ menunjukkan rumus " $i \geq 0 \wedge j \geq i \wedge \pi(j)(p) = 1$." Dari sudut pandang rumus orde pertama, $\Box \Diamond p$ tidak setara dengan $\Diamond \Box p$.

Contoh 10.2.2 Untuk rumus $\Box \Diamond p \wedge \Box \Diamond \neg p$, kita memilih urutan π di mana nilai kebenaran p bersifat bergantian. $\Box \Diamond p$ benar di π karena $\Diamond p$ benar di setiap sufiks π . Perhatikan bahwa $\Box \Diamond p$ tidak benar di π , bukti lain bahwa $\Box \Diamond p$ dan $\Diamond \Box p$ tidak setara. $\Box \Diamond \neg p$ juga benar di π karena $\Diamond \neg p$ benar di setiap sufiks π . Jadi, π adalah model dari $\Box \Diamond p \wedge \Box \Diamond \neg p$. Baik $\Box \Diamond p$ maupun $\Box \Diamond \neg p$ tidak valid karena ada urutan di mana satu rumus salah dan yang lainnya benar.

Contoh 10.2.3 Rumus $\Diamond \Box p \wedge \Diamond \Box \neg p$ tidak dapat dipenuhi karena tidak benar di urutan π mana pun. Artinya, $\Diamond \Box p$ membutuhkan p benar di mana-mana dalam satu sufiks π . $\Diamond \Box \neg p$ mensyaratkan p bernilai salah di mana-mana dalam satu sufiks π . Kedua kondisi tersebut tidak dapat dipenuhi secara bersamaan.

Definisi 10.2.4 Suatu formula LTL A dapat dipenuhi jika terdapat urutan interpretasi π sedemikian sehingga $\pi \models A$ dan kita katakan π adalah model dari A . A valid dalam LTL, ditulis $\models A$, jika setiap urutan interpretasi adalah model dari A .

Baik $\Diamond \Box p$ dan $\Box \Diamond p$ dapat dipenuhi tetapi tidak valid. Dari Contoh 10.2.2, kita tahu $\Diamond \Box p \wedge \Box \Diamond \neg p$ dapat dipenuhi tetapi tidak valid. Dari Contoh 10.2.3, $\Diamond \Box p \wedge \Diamond \Box \neg p$ tidak dapat dipenuhi, dan negasinya valid.

Teorema 10.2.5 Untuk setiap formula LTL A

- (a) (**reflexivity**) $\models \Box A \rightarrow A$;
- (b) (**transitivity**) $\models \Box A \rightarrow \Box \Box A$.

Bukti (a): Untuk menunjukkan bahwa $\Box A \rightarrow A$ valid dalam LTL, pertimbangkan urutan interpretasi π apa pun untuk A . Jika $\Box A \rightarrow A$ salah dalam π , maka $\Box A$ benar dan A salah dalam π . $\Box A$ berarti A benar dalam $\pi_{\geq i}$ untuk setiap $i \in \mathbb{N}$, termasuk $i = 0$. Artinya, A benar dalam $\pi_{\geq 0} = \pi$. Itu merupakan kontradiksi dengan A yang salah dalam π . Oleh karena itu, $\Box A \rightarrow A$ benar dalam π .

Bukti (b) diserahkan sebagai latihan.

Perhatikan bahwa baik $\Box A \rightarrow A$ maupun $\Box A \rightarrow \Box \Box A$ tidak valid dalam TL. Teorema 10.1.12 menyatakan bahwa $\Box A \rightarrow A$ benar dalam setiap kerangka Kripke reflektif, dan Teorema 10.1.13 menyatakan bahwa $\Box A \rightarrow \Box \Box A$ benar dalam setiap kerangka Kripke transitif. Semantik dari urutan interpretasi bukanlah kasus khusus dari semantik Kripke karena kerangka Kripke tidak memiliki cara untuk menentukan jalur unik yang secara eksplisit didefinisikan oleh urutan interpretasi. Di sisi lain, urutan interpretasi tidak cukup ekspresif untuk menentukan berbagai relasi interpretasi proposisional yang disediakan oleh kerangka Kripke. Pada Bagian 10.2.3, kita

akan melihat bahwa kerangka Kripke masih sangat berguna untuk merepresentasikan kemungkinan model LTL dan dapat berfungsi sebagai dasar untuk pemeriksaan model dengan LTL. Diberikan sebuah formula LTL A , misalkan $M(A)$ menunjukkan himpunan semua model A .

Definisi 10.2.6 Diberikan dua formula LTL A dan B , A dan B setara, ditulis $A \equiv B$, jika $M(A) = M(B)$.

Kita katakan A menyiratkan B , atau B adalah konsekuensi logis dari A , ditulis $A \models B$, jika $M(A) \subseteq M(B)$.

Teorema 10.2.7 Untuk setiap formula LTL A

- (a) $\Box A \models \circ A$;
- (b) $\circ A \models \Diamond A$;
- (c) $\Box A \models \Diamond A$.

Bukti (a) dan (b) berasal dari definisi semantik operator modal ini; (c) berasal dari transitivitas implikasi.

Seperti TL, semua tautologi dalam logika proposisional berlaku di LTL dan semua hasil utama dari logika proposisional masih berguna di sini.

Teorema 10.2.8

- (a) **(Substitusi)** Setiap contoh formula LTL yang valid berlaku di LTL. Artinya, jika p adalah variabel proposisional dalam A dan B adalah formula apa pun, maka $A[p \sim B]$ valid setiap kali A valid.
- (b) **(Substitusi kesetaraan)** Untuk setiap formula LTL A, B , dan C , di mana B adalah subformula dari A , dan $B \equiv C$, maka $A \equiv A[B \sim C]$.
- (c) **(Kesetaraan Logis)** Diberikan dua rumus LTL A dan B , $\models A \leftrightarrow B$ jika dan hanya jika $A \equiv B$.
- (d) **(Keterkaitan)** Diberikan dua rumus LTL A dan B , $\models A \rightarrow B$ jika dan hanya jika $A \models B$.

Bukti teorema di atas analog dengan bukti untuk logika proposisional dan dihilangkan di sini.

Sifat-Sifat LTL

Bukti teorema berikut adalah contoh pembuktian bahwa suatu rumus valid atau setara dengan rumus lain. Pada bagian selanjutnya, kita akan memberikan beberapa teknik pembuktian berdasarkan tabel semantik. Operator $\circ$ bercommute dengan $\Box$, $\Diamond$, dan $\neg$, tetapi $\Box$ dan $\Diamond$ tidak bercommute satu sama lain, seperti yang ditunjukkan pada Contoh 10.2.3.

Teorema 10.2.9 (Komutativitas)

- (a) $\Box \circ p \equiv \circ \Box p$;

- (b) $\Diamond \circ p \equiv \circ \Diamond p$;
 (c) $\neg \circ p \equiv \circ \neg p$.

Bukti Bukti (a) dan (b) serupa dengan bukti (c): diberikan urutan interpretasi π apa pun, π adalah model dari $\neg \circ p$ jika dan hanya jika $\circ p$ salah di π ; $\circ p$ salah di π jika dan hanya jika p salah di $\pi_{\geq 1}$, atau secara ekuivalen, $\neg p$ benar di $\pi_{\geq 1}$; $\neg p$ benar di $\pi_{\geq 1}$ jika dan hanya jika $\circ \neg p$ benar di π .

Teorema 10.2.10 (Dualitas)

- (a) $\neg \Box p \equiv \Diamond \neg p$;
 (b) $\neg \Diamond p \equiv \Box \neg p$;

Bukti

- (a) Diberikan urutan interpretasi π apa pun, $\neg \Box p$ benar di π jika dan hanya jika $\Box p$ salah di π ; $\Box p$ salah di π jika dan hanya jika ada $i \geq 0, \pi(i)(p) = 0$, atau yang setara, $\pi(i)(\neg p) = 1$. Ada $i \geq 0, \pi(i)(\neg p) = 1$ jika dan hanya jika $\circ \neg p$ benar di π .
 (b) Ganti p dengan $\neg q$ dalam (a), dan terapkan $\neg \neg q \equiv q$, kita memiliki $\neg \Box \neg q \equiv \Diamond q$, atau $\neg \Diamond q \equiv \Box \neg q$.

Dalam Bab 2, kita memperkenalkan konsep yang disebut "bentuk normal negasi," yang merupakan rumus proposisional di mana setiap argumen dari $\neg$ adalah variabel proposisional. Konsep ini berlaku untuk rumus LTL maupun TL. Dengan menerapkan dua teorema di atas, kita memiliki teorema berikut.

Teorema 10.2.11 Untuk setiap rumus LTL A , terdapat rumus LTL B sedemikian sehingga $A \equiv B$ dan B berada dalam bentuk normal negasi.

Bukti Aturan tambahan yang dibutuhkan untuk LTL adalah Teorema 10.2.9(c) dan 10.2.10.

Kita akan menggunakan bentuk normal negasi sebagai metode pembuktian, seperti yang ditunjukkan oleh contoh berikutnya.

Contoh 10.2.12 Untuk menunjukkan bahwa $\Diamond \Box p \rightarrow \Box \Diamond p$ valid, kita mengubah $\neg(\Diamond \Box p \rightarrow \Box \Diamond p)$ menjadi bentuk normal negasi:

$$\begin{aligned} \neg(\Diamond \Box p \rightarrow \Box \Diamond p) &\equiv \Diamond \Box p \wedge \neg \Box \Diamond p \\ &\equiv \Diamond \Box p \wedge \Diamond \neg \Diamond p \\ &\equiv \Diamond \Box p \wedge \Diamond \Box \neg p \end{aligned}$$

Pada Contoh 10.2.3 ditunjukkan bahwa $\Diamond \Box p \wedge \Diamond \Box \neg p$ tidak dapat dipenuhi. Dengan demikian, $\Diamond \Box p \rightarrow \Box \Diamond p$ valid.

Di sisi lain, bentuk normal negasi dari $\neg(\Box \Diamond p \rightarrow \Diamond \Box p)$ adalah $\Box \Diamond p \wedge \Box \Diamond \neg p$, yang dapat dipenuhi (Contoh 10.2.2). Dengan demikian, $\Box \Diamond p \rightarrow \Diamond \Box p$ tidak valid.

Rumus dalam logika proposisional dapat ditransformasikan menjadi bentuk normal konjungtif (CNF) atau bentuk normal disjungtif (DNF). Namun, hal ini tidak mungkin dilakukan baik dalam LTL maupun TL. Seperti $\forall$ dalam logika orde pertama, $\Box$ tidak terdistribusi atas $\vee$; seperti $\exists$, $\Diamond$ tidak terdistribusi atas $\wedge$.

Teorema 10.2.13 (Distributivitas)

- (a) $\Box(p \wedge q) \equiv \Box p \wedge \Box q$;
- (b) $\Diamond(p \vee q) \equiv \Diamond p \vee \Diamond q$;
- (c) $\circ(p \wedge q) \equiv \circ p \wedge \circ q$;
- (d) $\circ(p \vee q) \equiv \circ p \vee \circ q$

Bukti

(a) Untuk setiap urutan interpretasi π , $(p \wedge q)$ benar di π jika dan hanya jika untuk setiap $i \geq 0$, $\pi(i)(p \wedge q) = 1$. Artinya, $\pi(i)(p) = 1$ dan $\pi(i)(q) = 1$ untuk setiap $i \geq 0$. Ini sama dengan p dan q , masing-masing.

(b) dapat diperoleh dari (a) dengan dualitas. Bukti (c) dan (d) diserahkan sebagai latihan.

Contoh 10.2.14 Untuk menunjukkan bahwa $\Diamond(p \vee q)$ tidak setara dengan $\Box p \vee \Box q$, pertimbangkan urutan interpretasi π di mana p hanya benar pada setiap keadaan bernomor ganjil dari π dan q hanya benar pada setiap keadaan bernomor genap dari π . Maka, $p \vee q$ benar pada setiap keadaan π ; oleh karena itu, $\Box(p \vee q)$ benar di π . Mudah dilihat bahwa baik $\Box p$ maupun $\Box q$ tidak benar di π . Perhatikan bahwa $\Box p \vee \Box q \not\models \Box(p \vee q)$.

Teorema 10.2.15 (Absorpsi)

- (a) $\Box \Box p \equiv \Box p$
- (b) $\Diamond \Diamond p \equiv \Diamond p$
- (c) $\Box \Diamond \Box p \equiv \Diamond \Box p$
- (d) $\Diamond \Box \Diamond p \equiv \Box \Diamond p$

Bukti

(a) berasal dari Teorema 10.2.5.

(b) adalah dual dari (a).

(c) dapat diperoleh dari

(d) dengan dualitas. Berikut adalah bukti (d): untuk setiap urutan interpretasi π , $\Box \Diamond \Box p$ benar di π jika dan hanya jika ada $i \geq 0$, $\Box \Diamond p$ benar di $\pi_{\geq i}$. $\Box \Diamond p$ benar di $\pi_{\geq i}$ jika dan hanya jika untuk semua $j \geq i$, $\Diamond p$ benar di $\pi_{\geq j}$. $\Diamond p$ benar di $\pi_{\geq j}$ untuk $j \geq i$ jika dan hanya jika $\Diamond p$ benar di $\pi_{\geq k}$ untuk setiap $k \geq 0$, yang setara dengan $\Box \Diamond p$ benar di π .

Teorema di atas memungkinkan kita untuk memampatkan rangkaian operator $\Box$ dan $\Diamond$ menjadi tidak lebih dari dua. Mereka berguna dalam penyederhanaan rumus LTL.

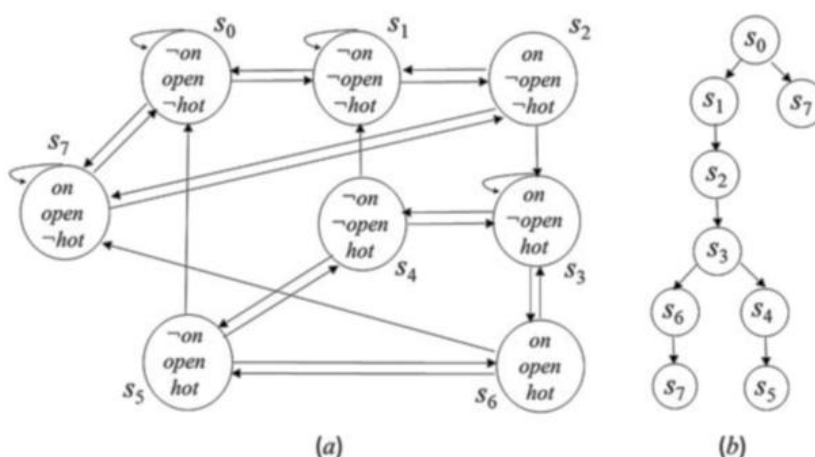
Pemeriksaan Model dengan Kerangka Kripke

Dalam banyak aplikasi LTL, relasi antar interpretasi variabel proposisional dapat ditetapkan, dan relasi ini dapat direpresentasikan oleh grafik berarah $K = (S, R)$, di mana S adalah himpunan interpretasi dan R adalah relasi biner atas S . K disebut kerangka Kripke untuk logika waktu percabangan, dan kita akan menggunakan K untuk LTL juga.

Setiap jalur dengan panjang tak terbatas dari K memberi kita urutan interpretasi, dan kita dapat memeriksa apakah urutan ini merupakan model LTL atau bukan. Ide ini disebut "pemeriksaan model" dalam komunitas metode formal.

Contoh 10.2.16 Misalkan kita ingin membangun model untuk mengoperasikan oven listrik dan mengharapkan bahwa setelah daya "on," oven akhirnya akan "hot." Dalam LTL, ini dinyatakan sebagai $\Box(on \rightarrow \Diamond hot)$. Kita juga meminta agar oven tidak akan *hot* jika pintu oven "open." Dalam LTL, kita menentukannya sebagai $\Box(open \rightarrow \Diamond \neg hot)$.

Hubungan antara ketiga variabel proposisional tersebut diberikan oleh kerangka Kripke pada Gambar 10.2a, yang menunjukkan hasil dari "daya oven menyala," "membuka atau menutup pintu oven," dan fungsi oven biasa. Tentu saja, ini adalah model oven yang disederhanakan untuk tujuan ilustrasi.



Gambar 10.2 (a) Kerangka Kripke untuk model oven, di mana terdapat hubungan antara dua interpretasi yang berbeda satu nilai kebenaran; (b) pohon pencarian parsial dari kerangka tersebut

Suatu *lintasan* (node yang berulang diperbolehkan) dalam kerangka Kripke $K = (S, R)$ adalah urutan keadaan $s_1, s_2, s_3, \dots, s_i, \dots$ di atas S , sedemikian sehingga $(s_i, s_{i+1}) \in R$ untuk semua $i \geq 1$. Karena K terbatas, lintasan tak terbatas dari kerangka tersebut dapat dengan mudah dinyatakan sebagai urutan keadaan yang diakhiri dengan suburutan yang dapat diulang

sejumlah kali. Misalnya, s_0^* , $s_0 s_1^*$, $(s_0 s_1)^*$, $s_0 (s_1 s_2 s_3 s_4)^*$, dan seterusnya adalah lintasan seperti itu dalam contoh oven, di mana x^* menunjukkan bahwa kita mengulangi x sejumlah kali.

Jika suatu barisan interpretasi tak hingga π direpresentasikan oleh $s_0 s_1 s_2 \cdots s_k (s_{k+1} s_{k+2} \cdots s_{k+m})^*$, kita katakan barisan tersebut teratur (disebut ekspresi teratur dalam teori komputasi).

Cukup mudah untuk memeriksa apakah $s_0 s_1 \cdots s_k (s_{k+1} s_{k+2} \cdots s_{k+m})^*$ merupakan model dari formula LTL C dengan mengikuti Definisi 10.2.1, di mana kondisi $i \geq 0$ pada dua kasus terakhir dapat diganti dengan $0 \leq i \leq k + m$:

7. C adalah $\Box A$ dan untuk semua $0 \leq i \leq k + m$, $\pi_{\geq i} \models A$.
8. C adalah $\Diamond A$ dan terdapat $0 \leq i \leq k + m$, $\pi_{\geq i} \models A$.

Contoh 10.2.17 Mari kita lanjutkan dengan Contoh 10.2.16 dan misalkan A adalah $\Box(on \rightarrow \Diamond hot)$. Kita tunjukkan bahwa A benar dalam $s_0 s_1 s_2 s_3^*$, suatu barisan reguler pada Gambar 10.2. Agar A benar, $on \rightarrow \Diamond hot$ harus benar di semua akhiran $s_0 s_1 s_2 s_3^*$: Yaitu $s_0 s_1 s_2 s_3^*$, $s_1 s_2 s_3^*$, $s_2 s_3^*$, dan s_3^* . Memang, $on \rightarrow \Diamond hot$ benar di $s_0 s_1 s_2 s_3^*$ dan $s_1 s_2 s_3^*$ karena on salah di s_0 dan s_1 . $on \rightarrow \Diamond hot$ juga benar di $s_2 s_3^*$ dan s_3^* karena $\Diamond hot$ benar di $s_2 s_3^*$ dan s_3^* (karena hot benar di s_3). Dengan demikian, A dapat dipenuhi.

Contoh di atas menunjukkan proses kerja tipikal dari pemeriksaan model:

1. Bangun model sistem dengan mengabstraksikan detail yang tidak relevan.
2. Tulis spesifikasi formal tentang properti yang diinginkan, seperti tidak adanya kebuntuan, masalah keterjangkauan, invarian sistem, dll.
3. Jalankan pemeriksa model yang melakukan beberapa variasi analisis keterjangkauan pada model dengan jawaban bahwa spesifikasi terpenuhi atau ditemukan contoh tandingan.

Untuk saat ini, kami menganggap kerangka Kripke sebagai model kami. Kerangka Kripke dapat diperluas ke mesin keadaan terbatas sehingga sistem yang lebih kompleks dapat dimodelkan. Bagaimana mengontrol kompleksitas pencarian untuk menemukan contoh tandingan sangat penting untuk keberhasilan pemeriksaan model, dan strategi yang efisien diperlukan untuk menghindari ledakan ruang keadaan. Misalnya, $s_0 s_1 s_2 \cdots s_k (s_{k+1} s_{k+2} \cdots s_{k+m})^*$ setara dengan

$$s_0 s_1 s_2 \cdots s_k s_{k+1} s_{k+2} \cdots s_{k+m} (s_{k+1} s_{k+2} \cdots s_{k+m})^*$$

Kita dapat menggunakan kesetaraan ini untuk mengurangi jumlah urutan reguler dalam pemeriksaan model.

Untuk kerangka Kripke $K = (S, R)$, pemeriksaan model akan memeriksa semua urutan reguler K sebagai berikut: pilih sebuah simpul di S sebagai akar dari pohon T dan perluas berulang kali setiap simpul daun T sesuai dengan R . Kriteria penghentian perluasan adalah

bahwa jalur dari akar ke daun tidak berisi lebih dari satu salinan dari simpul mana pun. Dengan cara ini, pohon akan terbatas untuk K , dan semua urutan reguler yang tidak setara yang diwakili oleh T juga akan terbatas. Untuk setiap urutan reguler yang diwakili oleh T , pemeriksaan model akan memeriksa apakah itu merupakan model dari rumus yang diberikan.

Untuk kerangka Kripke yang diberikan pada Gambar 10.2a, pohon pencarian ditunjukkan pada (b). Jalur paling kanan, yaitu s_0s_7 , mewakili urutan reguler berikut: $s_0^*, s_0s_7^*$, dan $(s_0s_7)^*$. Untuk menunjukkan bahwa “setelah daya dinyalakan, oven pada akhirnya akan panas,” kita harus menunjukkan bahwa $\neg A$, di mana A adalah $\Box(on \rightarrow \Diamond hot)$, tidak memiliki model. Namun, $s_0s_7^*$ adalah model dari $\neg A$, yang setara dengan $\Diamond(on \wedge \neg \Box hot)$; dengan demikian, A bukanlah teorema dari model oven. Kita perlu memodifikasi A atau model oven jika kita ingin A menjadi sebuah teorema.

10.3 TABEL SEMANTIK UNTUK LTL

Metode tabel semantik yang diperkenalkan dalam Bab 3 adalah prosedur pengambilan keputusan untuk kepuasan dalam logika proposisional. Pada bagian ini, kita memperluas metode ini untuk LTL dengan menambahkan aturan untuk operator modal. Perluasan untuk TL serupa.

Model LTL lebih kompleks daripada model proposisional karena model proposisional adalah interpretasi proposisional tunggal; model LTL adalah urutan interpretasi proposisional. Kita perlu menemukan interpretasi untuk setiap keadaan dalam urutan tersebut. Oleh karena itu, kita perlu mengelompokkan node dalam tabel semantik berdasarkan keadaannya. Misalnya, aturan β untuk $\vee$ akan menghasilkan dua anak untuk node yang berisi $p \vee q$, dan kedua anak ini termasuk dalam kelompok induknya. Di sisi lain, $\circ p$ akan menghasilkan node anak yang termasuk dalam kelompok berbeda yang terkait dengan keadaan selanjutnya dari keadaan saat ini.

Seperti logika proposisional, metode tableau yang diperluas mengambil formula masukan A , membuat simpul awal yang berisi A , dan kemudian menerapkan aturan berulang kali ke simpul daun mana pun untuk menghasilkan penerusnya. Aturan tableau untuk LTL terdiri dari aturan untuk logika proposisional yang ditunjukkan pada Bagian 3.1.2, ditambah aturan α baru untuk $\Box$, aturan β baru untuk $\Diamond$, jenis aturan baru yang disebut aturan X (aturan neXt) untuk $\circ$. Untuk menyederhanakan pembahasan, kita akan menerapkan beberapa aturan penyederhanaan pada formula awal. Aturan penyederhanaan mencakup aturan penulisan ulang berikut dan beberapa aturan yang melibatkan konstanta T dan $\perp$ (seperti $\perp \vee A \rightarrow A$ dan $\Box T \rightarrow T$):

- (1) $\neg \neg A \rightarrow A$
- (2) $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$
- (3) $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$
- (4) $\neg(A \rightarrow B) \rightarrow A \wedge \neg B$
- (5) $\neg(A \leftrightarrow B) \rightarrow (A \wedge \neg B) \vee (\neg A \wedge B)$
- (6) $\neg \Box A \rightarrow \Diamond \neg A$

- $$\begin{aligned}
(7) \quad & \neg \Diamond A \rightarrow \Box \neg A \\
(8) \quad & \neg \circ A \rightarrow \circ \neg A \\
(9) \quad & \Box \circ A \rightarrow \circ \Box A \\
(10) \quad & \Diamond \circ A \rightarrow \circ \Diamond A
\end{aligned}$$

Tiga aturan terakhir adalah aturan komutativitas yang menggerakkan $\circ$ ke luar; aturan lainnya menggerakkan $\neg$ ke dalam. Aturan penyederhanaan ini didasarkan pada kesetaraan logis dan mempertahankan relasi kesetaraan antara kedua rumus sebelum dan sesudah penerapan aturan penulisan ulang. Aturan-aturan tersebut akan mengubah rumus awal menjadi bentuk normal negasi di mana $\neg, \vee, \wedge, \circ, \Box, \Diamond$ adalah satu-satunya operator.

Aturan untuk Operator Modal

Pada Bab 3, kita telah menunjukkan bagaimana sebuah tabel dibuat dengan membuat simpul awal dari rumus masukan dan menerapkan aturan α dan β pada rumus dalam sebuah simpul untuk membuat simpul penerus. Aturan α membuat satu penerus, dan aturan β membuat dua penerus. Misalkan n adalah simpul saat ini dan $F(n)$ adalah himpunan rumus yang terdapat dalam n . Jika aturan α dapat mengganti A dengan $\{B, C\}$ untuk $A \in F(n)$, maka penerus n , yang dilambangkan dengan n' , akan tercipta dan $F(n') = F(n) - \{A\} \cup \{B, C\}$. Jika aturan β dapat mengganti A dengan B dan C untuk $A \in F(n)$, maka dua penerus n , yang dilambangkan dengan n' dan n'' , akan tercipta sedemikian sehingga $F(n') = F(n) - \{A\} \cup \{B\}$ dan $F(n'') = F(n) - \{A\} \cup \{C\}$.

Seperti pada Bab 3, diberikan sebuah tabel, kita katakan sebuah simpul tertutup jika simpul tersebut berisi sepasang literal komplementer. Sebuah simpul terbuka jika simpul tersebut tidak tertutup dan hanya berisi literal. Sebuah simpul dapat diperluas jika simpul tersebut adalah simpul daun yang tidak tertutup maupun terbuka. Simpul yang dapat diperluas selalu berisi rumus yang memungkinkan salah satu aturan α atau β untuk diterapkan. Untuk LTL, kami memiliki jenis node baru, yang disebut X -node.

Definisi 10.3.1 (Rumus X , Simpul X , dan Aturan X) Jika simbol akar dari rumus LTL A adalah $\circ$ dan tidak ada aturan yang dapat diterapkan pada A , maka A disebut rumus X atau rumus berikutnya.

Suatu simpul dari tabel semantik disebut simpul X jika simpul tersebut berisi setidaknya satu rumus X , dan setiap rumus dalam simpul tersebut adalah literal atau rumus X .

Misalkan $F(n)$ adalah rumus-rumus dalam simpul X n :

$$F(n) = \{A_1, A_2, \dots, A_k, \circ B_1, \circ B_2, \dots, \circ B_m\},$$

di mana $k \geq 0, m \geq 1$, dan $\{A_1, A_2, \dots, A_k\}$ adalah himpunan literal yang konsisten, maka aturan X akan menghasilkan interpretasi, yang membuat semua $A_1, A_2, \dots, A_n$ menjadi benar, untuk keadaan saat ini dan membuat simpul baru n' , di mana $F(n') = \{B_1, B_2, \dots, B_m\}$ untuk keadaan berikutnya.

Contoh 10.3.2 Asumsikan bahwa simpul awal berisi rumus masukan $\neg p \wedge \circ p$. Dengan menerapkan aturan α untuk $\wedge$, kita memperoleh simpul kedua yang berisi $\{\neg p, \circ p\}$. Di sini, $\neg p$ adalah literal, $\circ p$ adalah rumus X , dan tidak ada aturan yang dapat diterapkan kecuali aturan X . Aturan X akan menghasilkan simpul ketiga yang berisi p , dan simpul ini termasuk dalam keadaan yang berbeda. Dari simpul kedua, kita membuat $s_0 = \{\neg p\}$; Dari simpul ketiga, kita membuat $s_1 = \{p\}$. Urutan interpretasi yang dibuat dari tabel ini adalah $s_0, s_1, s_1, s_1, \dots$, yang merupakan model dari $\neg p \wedge \circ p$. Bahkan, setiap urutan yang dua keadaan pertamanya adalah s_0 dan s_1 akan menjadi model dari rumus ini.

Menurut definisi, aturan X hanya dapat diterapkan pada simpul X . Baik aturan penyederhanaan maupun aturan α dan β tidak dapat diterapkan pada rumus apa pun di simpul X . Lebih banyak contoh aplikasi aturan X akan diberikan segera, setelah menyajikan aturan untuk $\Box$ dan $\Diamond$.

Untuk membangun model untuk $\Box A$, kita perlu menemukan urutan interpretasi π sedemikian sehingga A benar di setiap sufiks π . Artinya, A benar di $\pi(0)$ dan $\Box A$ harus benar di semua keadaan selanjutnya. Artinya, alih-alih $\Box A$, kita akan membangun model untuk $A \wedge \circ \Box A$. Secara intuitif, jika $\circ$ mewakili “besok,” maka A benar setiap hari jika dan hanya jika A benar hari ini dan benar setiap besok.

Teorema 10.3.3 (Induksi)

- (a) $\Box A \equiv A \wedge \circ \Box A$
 (b) $\Diamond A \equiv A \vee \circ \Diamond A$

Bukti

- (a) Untuk setiap urutan interpretasi π , $A \wedge \circ \Box A$ benar di π jika dan hanya jika A benar di $\pi_{\geq 0}$ dan $\Box A$ benar di $\pi_{\geq 1}$. A benar di $\pi_{\geq 1}$ jika dan hanya jika A benar di $\pi_{\geq i}$ untuk setiap $i \geq 1$. Dengan demikian, A harus benar di $\pi_{\geq i}$ untuk setiap $i \geq 0$, yang persis sama dengan “ $\Box A$ benar di π ” menurut definisi.
 (b) adalah kasus dual dari (a).

Teorema di atas berfungsi sebagai alat induktif untuk membangun model A dan $\circ A$: dari (a), A adalah kasus dasar dan $\circ \Box A$ adalah kasus induktif. Setelah kedua kasus selesai, kita memperoleh model untuk $\Box A$. Dari (b), A adalah kasus dasar; jika model A ditemukan, kita menggunakannya sebagai model $\Diamond A$. Jika tidak, dari kasus induktif, kita mencoba menemukan model $\circ \Diamond A$.

Definisi 10.3.4 Aturan α untuk $\Box A$ dan aturan β untuk $\Diamond A$ diberikan di bawah ini:

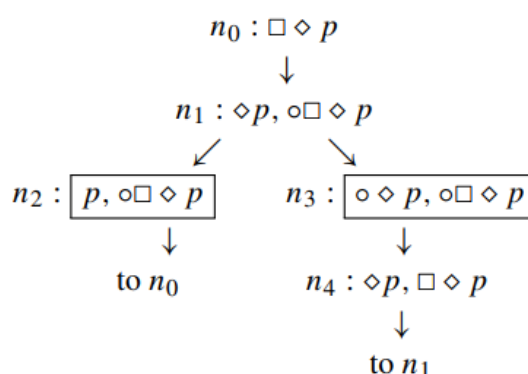
α	α_1, α_2	β	β_1	β_2
$\Box A$	$A, \circ \Box A$	$\Diamond A$	A	$\circ \Diamond A$

Keabsahan kedua aturan ini dipastikan oleh Teorema 10.3.3.

Konstruksi tabel semantik untuk LTL mirip dengan logika proposisional dengan mengulangi aturan $\alpha-$, $\beta-$, dan $X-$ pada node yang dapat diperluas, hingga tidak ada aturan yang dapat diterapkan. Namun, ada satu perbedaan kecil dan satu perbedaan besar. Perbedaan kecilnya adalah kita menerapkan aturan penyederhanaan terlebih dahulu pada formula masukan untuk mendapatkan bentuk normal negasi sebelum memulai konstruksi tabel. Perbedaan ini dapat dihilangkan jika kita memperlakukan aturan penyederhanaan sebagai aturan α .

Perbedaan besarnya adalah kita perlu memeriksa apakah node baru berisi himpunan formula yang sama dengan node yang sudah ada. Jika ya, alih-alih membuat node baru, kita membuat tautan ke node yang sudah ada. Karena tautan ini, tabel yang dibuat masih berupa grafik berarah tetapi bukan lagi pohon. Grafik ini dapat berisi siklus dan setiap node memiliki paling banyak dua penerus (yaitu, dua tautan keluar). Penghentian konstruksi tabel bergantung pada perbedaan utama ini karena tabel dapat berisi jalur dengan panjang tak terbatas.

Contoh 10.3.5 Tabel untuk $\Box \Diamond p$ ditunjukkan di bawah ini, di mana simpul X diberi kotak.



Tabel ini tidak memiliki simpul daun (baik simpul terbuka maupun tertutup). Simpul yang diberi kotak adalah simpul X . Kita akan menunjukkan nanti bagaimana cara menurunkan model dari tabel ini.

Diberikan tabel sebagai graf berarah, kita masih menyebut simpul tanpa tautan keluar sebagai daun, yang bisa berupa simpul tertutup, terbuka, atau dapat diperluas. Diberikan dua simpul n dan n' dari graf berarah, n' dikatakan dapat *dijangkau* dari n jika ada jalur dari n ke n' dalam graf tersebut.

Untuk setiap simpul n dari tabel, misalkan $F(n)$ menunjukkan himpunan rumus yang muncul di n . Seperti pada Bab 3, kita mengasumsikan bahwa $F(n) \equiv \bigwedge_{A \in F(n)} A$, yaitu, himpunan rumus setara dengan konjungsi rumus dalam himpunan tersebut.

Teorema 10.3.6

- (a) Konstruksi tabel untuk formula LTL selalu berakhir.
- (b) Jika node n' diturunkan dari n dengan aturan α , maka $F(n) \equiv F(n')$; jika n' dan n'' diturunkan dari n dengan aturan β , maka $F(n) \equiv F(n') \vee F(n'')$.

- (c) Jika node n' diturunkan dari n dengan aturan X , maka urutan interpretasi π ; adalah model dari $F(n')$ jika dan hanya jika $\pi = s_0\pi'$ adalah model dari $F(n)$, di mana $s_0 \in All_p$ adalah interpretasi yang membuat setiap literal $A \in F(n)$ benar.

Bukti

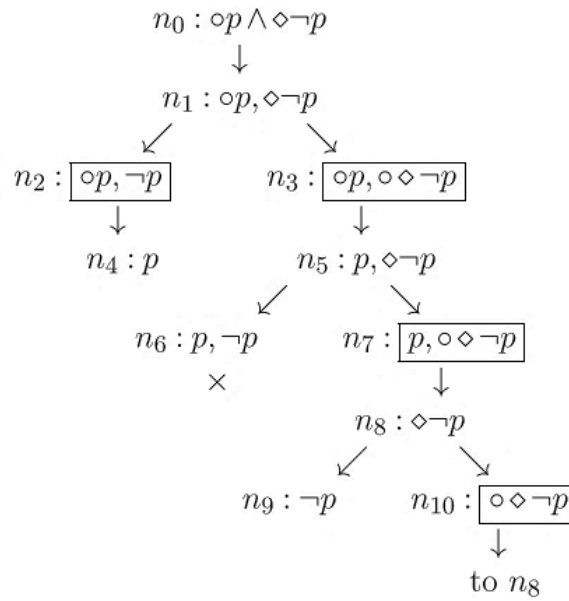
- (a) Konstruksi berhenti ketika tidak ada node yang dapat diperluas. Jika konstruksi tidak berhenti, maka akan ada jalur node tak terbatas dalam tabel sedemikian sehingga setiap node berisi himpunan formula yang berbeda. Oleh karena itu, himpunan rumus dari simpul-simpul ini tidak terbatas. Namun, aturan yang digunakan dalam konstruksi hanya dapat menghasilkan subrumus dari rumus asli atau menambahkan paling banyak satu operator $\circ$ ke dalamnya. Dengan kata lain, hanya sejumlah rumus terbatas yang dapat dihasilkan. Ini adalah kontradiksi, sehingga tabel tidak dapat memiliki jalur tak terbatas selain siklus.
- (b) Semua aturan α dan β yang digunakan dalam konstruksi mempertahankan relasi ekuivalensi.
- (c) Ketika aturan X diterapkan pada n untuk menghasilkan n' , di mana $F(n) = \{A_1, A_2, \dots, A_k, \circ B_1, \circ B_2, \dots, \circ B_m\}$, $F(n') = \{B_1, B_2, \dots, B_m\}$, di mana B_i adalah rumus apa pun. Dengan asumsi, $F(n) \equiv (\bigwedge_{1 \leq i \leq k} A_i) \wedge (\bigwedge_{1 \leq j \leq m} B_j)$. Misalkan π' adalah urutan interpretasi apa pun dan s_0 adalah interpretasi sedemikian sehingga $s_0(A_i) = 1$ untuk $1 \leq i \leq k$. Maka, π' adalah model dari $F(n')$ jika dan hanya jika $s\pi'$ adalah model dari $\circ (\bigwedge_{1 \leq j \leq m} B_j)$ untuk setiap $s \in All_p$. Ambil $s = s_0$, maka $\pi = s_0\pi'$ adalah model dari $(\bigwedge_{1 \leq i \leq k} A_i) \wedge \circ (\bigwedge_{1 \leq j \leq m} B_j)$, karena s_0 adalah model dari $\bigwedge_{1 \leq i \leq k} A_i$. Artinya, $\pi = s_0\pi'$ adalah model dari $F(n)$.

Bagian (a) dari teorema di atas menyatakan bahwa konstruksi tabel untuk formula LTL akan berakhir sedemikian sehingga setiap daun tertutup atau terbuka, tanpa simpul yang dapat diperluas. Jika tabel memiliki simpul terbuka, bagian (b) dan (c) memberi tahu kita cara membangun model (untuk rumus masukan di simpul awal) dari simpul terbuka ini dari bawah ke atas, sehingga telah menunjukkan bahwa rumus asli dapat dipenuhi. Apa yang terjadi ketika tabel tidak memiliki simpul terbuka? Kita akan membahas kasus ini selanjutnya.

Menentukan Kepuasan dengan Tabel

Kita telah melihat bagaimana tabel dapat dibangun untuk setiap rumus LTL. Untuk logika proposisional, suatu rumus dapat dipenuhi jika dan hanya jika tabelnya memiliki simpul terbuka. Untuk LTL, setidaknya satu sisi bernilai benar.

Teorema 10.3.7 Jika tabel untuk rumus LTL A memiliki simpul terbuka, maka A dapat dipenuhi.



Gambar 10.3 Tabel untuk $\odot p \wedge \odot \neg p$

Bukti Misalkan $(n_0, n_1, \dots, n_m)$ adalah jalur sederhana dari simpul awal n_0 ke simpul terbuka n_m dalam tabel. Kita tunjukkan bahwa $F(n_j)$ memiliki model π_j untuk $j = m, m-1, \dots, 1, 0$ dengan induksi.

Untuk kasus dasar ketika $j = m$, $F(n_m)$ adalah himpunan literal yang konsisten. Misalkan s_m adalah interpretasi yang membuat setiap literal dari $F(n_m)$ benar dan $\pi_m = s_m^+$, di mana s_m^+ berarti kita mengulangi s_m selamanya, maka π_m adalah model dari $F(n_m)$.

Sebagai hipotesis induksi, kita mengasumsikan bahwa π_j adalah model dari $F(n_j)$, $0 < j \leq m$. Jika n_j adalah penerus dari n_{j-1} dengan aturan α atau aturan β , menurut Teorema 10.3.6 (b), $\pi_{j-1} = \pi_j$ juga merupakan model dari $F(n_{j-1})$. Jika n_j adalah penerus dari n_{j-1} menurut aturan X , menurut Teorema 10.3.6 (c), $\pi_{j-1} = s_{j-1}\pi_j$ adalah model dari $F(n_{j-1})$, di mana s_{j-1} adalah interpretasi yang dibuat dari literal $A_i \in F(n_{j-1})$.

Terakhir, ketika $j = 0$, π_0 adalah model dari $F(n_0) = \{A\}$. Dengan demikian, A dapat dipenuhi. Untuk LTL, sebuah tabel mungkin tidak memiliki simpul terbuka, seperti yang terlihat pada Contoh 10.3.5. Tabel tersebut juga tidak memiliki simpul tertutup. Bagaimana kita dapat mengetahui apakah rumus masukan dari tabel tersebut dapat dipenuhi? Mari kita lihat contoh lain.

Contoh 10.3.8 Untuk menunjukkan bahwa $\odot p \wedge \odot \neg p$ dapat dipenuhi, kita dapat membangun model dengan tabel semantik. Dengan menggunakan aturan tabel, kita memperoleh tabel yang ditunjukkan pada Gambar 10.3, di mana simpul X diberi kotak.

Sebuah model adalah urutan interpretasi dan sebuah interpretasi dapat dibuat baik dari simpul terbuka atau simpul X . Misalnya, pertimbangkan jalur (n_0, n_1, n_2, n_4) : dari n_2 , kita

membuat $s_0 = \{\neg p\}$; dari n_4 , kita membuat $s_1 = \{p\}$. Dua interpretasi pertama dari model kita adalah s_0 dan s_1 , masing-masing. Dengan mengulang s_1 , kita memperoleh urutan interpretasi tak terbatas yang dilambangkan dengan $s_0 s_1^+$, di mana s_1^+ berarti kita mengulang s_1 satu kali atau lebih. Mudah untuk memeriksa bahwa $s_0 s_1^+$ adalah model dari $\circ p \wedge \Diamond \neg p$. Karena model interpretasi setelah s_1 tidak memengaruhi nilai kebenaran p , misalkan s_2 menunjukkan interpretasi di mana p mengambil nilai apa pun, maka representasi umum dari model dari jalur ini adalah $s_0 s_1 s_2^*$, di mana s_2^* berarti kita dapat mengulangi s_2 sebanyak berapa pun. Pertimbangkan jalur lain yang berakhir dengan simpul terbuka:

$$(n_0, n_1, n_3, n_5, n_7, n_8, n_9)$$

di mana n_3 dan n_7 adalah simpul X dan n_9 terbuka. n_3 tidak menentukan nilai kebenaran p , dan kita menamai interpretasinya sebagai s_2 di mana p dapat mengambil nilai kebenaran apa pun. Interpretasi yang dibuat dari n_7 dan n_9 masing-masing adalah $s_1 = \{p\}$ dan $s_0 = \{\neg p\}$. Model yang dibuat dari jalur ini adalah $s_2 s_1 s_0 s_2^*$.

Pertimbangkan jalur berikut:

$$(n_0, n_1, n_3, n_5, n_7, n_8, n_{10})$$

di mana n_3, n_7 , dan n_{10} adalah simpul X . Jika interpretasi yang dibuat dari mereka selalu membuat p benar, maka tidak ada model yang dapat ditemukan dari jalur ini. Di sisi lain, jika jalurnya adalah

$$(n_0, n_1, n_3, n_5, n_7, n_8, n_{10}, n_8, n_9),$$

Model dari jalur ini adalah $s_2 s_1 s_2 s_0 s_2^*$. Jika kita mengulangi n_8 dan n_{10} beberapa kali, maka semua model dari jalur ini dapat direpresentasikan oleh $s_2 s_1 s_2^* s_0 s_2^*$, di mana p dapat mengambil nilai apa pun di s_2 .

Faktanya, semua model dari $\circ p \wedge \Diamond \neg p$ dapat dilambangkan dengan $s_0 s_1 s_2^* \cup s_2 s_1 s_2^* s_0 s_2^*$. Setiap model sesuai dengan beberapa jalur dalam tabel. Contoh di atas memberi tahu kita bahwa ketika ada simpul terbuka, kita dapat membangun model untuk rumus masukan. Ketika jalur berakhir dengan siklus, model mungkin atau mungkin tidak dibangun.

Contoh 10.3.9 Lihat kembali tabel pada Contoh 10.3.5: tabel tersebut tidak memiliki simpul terbuka. Jika kita memilih siklus (n_0, n_1, n_2) , misalkan $s_1 = \{p\}$ adalah interpretasi yang diperoleh dari simpul X di n_2 , maka s_1^+ adalah model dari rumus masukan $\Box \Diamond p$.

Mari kita lihat jalur (n_0, n_1, n_3, n_4) , di mana tiga simpul terakhir berada dalam sebuah siklus. Interpretasi yang dibuat dari n_3 tidak memperhatikan nilai kebenaran p dan kita menunjuknya dengan s_2 , maka s_2^+ bukanlah sebuah model.

Di sisi lain, pertimbangkan jalur:

$$(n_0, n_1, n_2, n_0, n_1, n_3, n_4)$$

Urutan $(s_1 s_2)^+$ adalah model dari $\Box \Diamond p$, sedangkan urutan $s_1 s_2^+$ bukanlah model. Dengan kata lain, kita perlu mengulang semua interpretasi yang dihasilkan dari simpul-simpul dalam sebuah siklus. Perhatikan bahwa semua model dari $\Box \Diamond p$ dapat dilambangkan dengan urutan $(s_2 s_1)^+$.

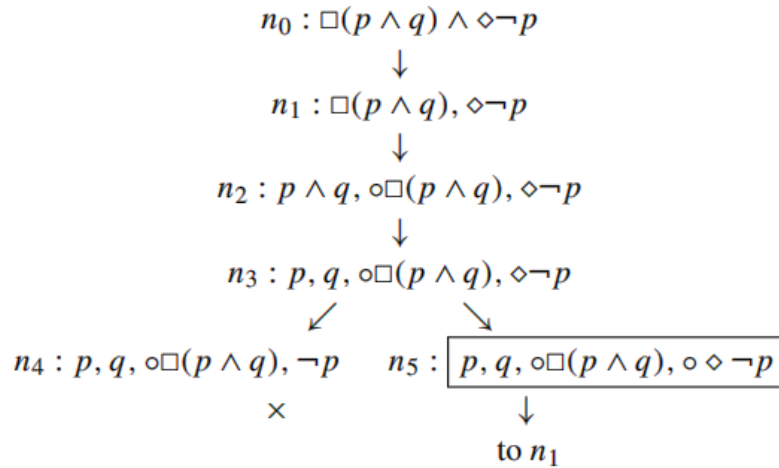
Contoh-contoh di atas menunjukkan bahwa beberapa jalur yang berakhir dengan siklus menghasilkan model dan beberapa tidak. Tujuan kita menggunakan tabel adalah untuk memutuskan apakah rumus masukan dapat dipenuhi atau tidak. Sebuah model hanyalah bukti bahwa rumus masukan dapat dipenuhi. Jika sebuah tabel memiliki simpul terbuka, kita tahu rumus masukan dapat dipenuhi karena sebuah model dapat dibangun dari jalur yang menghubungkan simpul awal dan simpul terbuka. Jika tabel tidak memiliki simpul terbuka, kita mendefinisikan di bawah ini mekanisme penandaan untuk memeriksa secara efektif apakah rumus masukan dapat dipenuhi atau tidak.

Definisi 10.3.10 Diberikan sebuah tabel yang tidak memiliki simpul yang dapat diperluas. Suatu simpul n ditandai mati jika salah satu dari kondisi berikut benar:

1. n tertutup.
2. Semua penerus n ditandai mati.
3. Terdapat suatu rumus $\Diamond A \in F(n)$ sedemikian sehingga tidak ada simpul n' yang dapat dijangkau dari n dan $A \in F(n')$.

Dalam kasus ketiga, kondisi jangkauan diperiksa di antara simpul-simpul yang tidak ditandai mati. Artinya, jika suatu simpul ditandai mati, simpul tersebut tidak dapat digunakan dalam jalur apa pun. Simpul n ditandai mati karena mengandung $\Diamond A$ yang tidak terpenuhi: $\Diamond A$ benar dalam suatu urutan π jika dan hanya jika A benar dalam suatu sufiks dari π . Jika A tidak muncul dalam simpul n' mana pun yang dapat dijangkau dari n , maka tidak mungkin untuk membangun sufiks π sedemikian sehingga π merupakan model dari $F(n)$.

Contoh 10.3.11 Untuk menunjukkan bahwa $\Box(p \wedge q) \rightarrow \Box p$ valid, kita menunjukkan bahwa negasinya tidak dapat dipenuhi oleh tabel semantik. Bentuk normal negasi dari $\neg(\Box(p \wedge q) \rightarrow \Box p)$ adalah $\Box(p \wedge q) \wedge \Diamond \neg p$, yang tabelnya diberikan di bawah ini:



Untuk $\Diamond \neg p \in F(n_1)$, tidak ada node n' yang dapat dijangkau dari n_1 dan $\neg p \in F(n')$. Dengan demikian, n_1 ditandai mati. n_0 ditandai mati karena penerusnya ditandai mati. Bahkan, setiap node dalam tabel ini dapat ditandai mati.

Contoh 10.3.12 Untuk tabel yang diberikan dalam Contoh 10.3.5, $\Diamond p$ muncul di n_4 . Karena n_2 dapat dijangkau dari n_4 melalui jalur (n_4, n_1, n_2) dan $p \in F(n_2)$, n_4 tidak dapat ditandai mati.

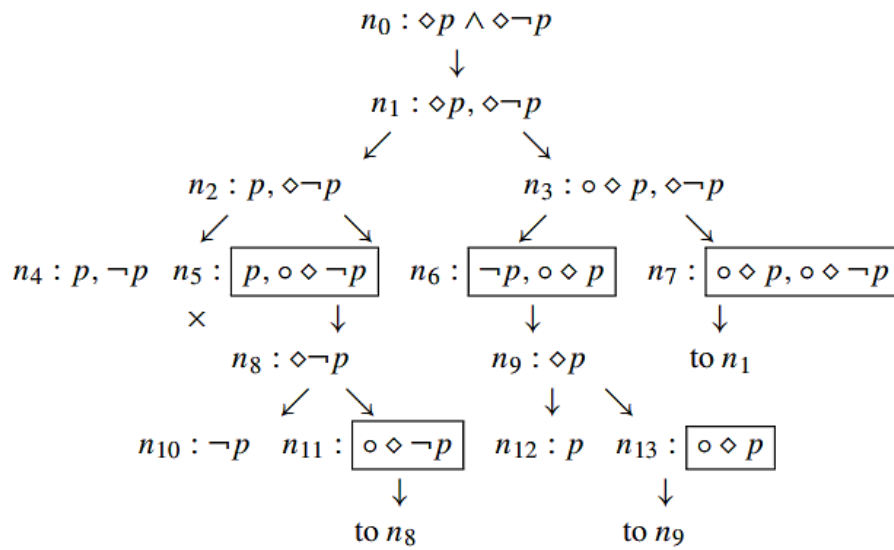
Algoritma 10.3.13 Algoritma `satisfiabilityByTableau` mengambil formula LTL A sebagai input dan mengembalikan "satisfiable" jika A dapat dipenuhi, jika tidak mengembalikan "unsatisfiable."

proc *satisfiabilityByTableau*(A)

1. Buat tableau untuk A :
 - 1.1. Buat node awal yang berisi bentuk normal negasi dari A .
 - 1.2. Terapkan aturan $\alpha-$, $\beta-$, dan $X-$ hingga tidak ada node yang dapat diperluas.
2. Tentukan apakah A dapat dipenuhi:
 - 2.1. Jika tableau memiliki node terbuka, kembalikan "dapat dipenuhi."
 - 2.2. Terapkan Definisi 10.3.10 hingga tidak ada node yang dapat ditandai mati.
 - 2.3. Jika node awal ditandai mati, kembalikan "tidak dapat dipenuhi."
 - 2.4. Jika tidak, kembalikan "dapat dipenuhi."

Meskipun algoritma hanya mengembalikan "dapat dipenuhi" atau "tidak dapat dipenuhi," kita masih dapat membangun model dari jalur tableau jika algoritma mengembalikan "dapat dipenuhi," seperti yang diilustrasikan oleh contoh-contoh berikut.

Contoh 10.3.14 Untuk menunjukkan bahwa $\Diamond p \wedge \Diamond \neg p$ dapat dipenuhi, kita membangun tabel semantiknya.



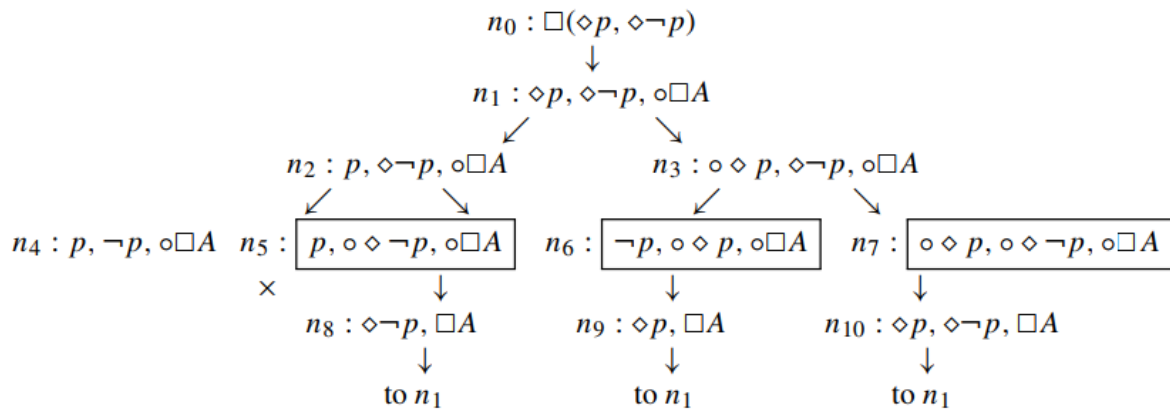
Untuk $\Diamond p \in F(n_1)$, n_5 atau n_{12} mengandung p ; untuk $\Diamond \neg p \in F(n_1)$, n_6 atau n_{10} mengandung $\neg p$. Keempat node tersebut dapat dijangkau dari n_1 . n_0 tidak dapat ditandai mati, sehingga algoritma akan mengembalikan "terpenuhi".

Berbagai model dapat dibangun dari tabel ini, tergantung pada jalur mana yang kita pilih dalam tabel tersebut. Misalkan $s_0 = \{\neg p\}$, $s_1 = \{p\}$, dan s_2 menunjukkan interpretasi di mana kita tidak peduli dengan nilai kebenaran p .

1. $(n_0, n_1, n_2, n_5, n_8, n_{10})$: Model yang diturunkan adalah $s_1 s_0 s_2^*$, di mana s_1 diturunkan dari n_5 dan s_0 dari n_{10} . s_2^* ditambahkan di akhir karena n_{10} terbuka.
2. $(n_0, n_1, n_3, n_6, n_9, n_{12})$: Model yang diturunkan adalah $s_0 s_1 s_2^*$, di mana s_0 diturunkan dari n_6 dan s_1 dari n_{12} .
3. $(n_0, n_1, n_2, n_5, n_8, n_{11})$: Dua node terakhir berada dalam siklus dan tidak ada model yang dapat ditemukan dari jalur ini. Untuk memenuhi $\Diamond \neg p \in F(n_1)$, kita membutuhkan n_6 atau n_{10} .
4. $(n_0, n_1, n_3, n_6, n_9, n_{13})$: Dua node terakhir berada dalam siklus dan tidak ada model yang dapat ditemukan dari jalur ini.
5. (n_0, n_1, n_3, n_7) : Tiga node terakhir berada dalam siklus dan tidak ada model yang dapat ditemukan dari jalur ini.

Semua model dari $\Diamond p \wedge \Diamond \neg p$ dapat dilambangkan dengan $s_2^* s_0 s_2^* s_1 s_2^2 \cup s_2^* s_1 s_2^* s_0 s_2^*$ dan setiap model sesuai dengan jalur dalam tabel.

Contoh 10.3.15 Untuk menunjukkan bahwa $\Box(\Diamond p \wedge \Diamond \neg p)$ dapat dipenuhi, kita membangun tabel semantik di mana kita mengganti $\wedge$ dengan “,” dan A mewakili $(\Diamond p, \Diamond \neg p)$.



Tabel ini memiliki satu simpul tertutup, yaitu n_4 , dan tidak ada simpul terbuka. n_4 adalah satu-satunya simpul yang ditandai mati. $\Diamond p$ dan $\Diamond \neg p$ di n_1 dipenuhi oleh n_5 dan n_6 , masing-masing. n_1 tidak dapat ditandai mati, begitu pula n_0 . Jadi, algoritma akan mengembalikan "dapat dipenuhi".

Jika kita ingin membangun model dari tabel tersebut, kita perlu mempertimbangkan jalur yang dimulai dari n_0 dan berisi n_5 dan n_6 . Misalkan $s_0 = \{p\}$ dan $s_1 = \{\neg p\}$ adalah interpretasi yang diperoleh dari simpul- X n_5 dan n_6 , masing-masing. Maka, $(s_0 s_1)^+$ adalah model dari rumus masukan. Setiap model membutuhkan pengulangan kedua interpretasi ini bersama-sama; mengulangi satu interpretasi tanpa batas tidak menghasilkan model. Misalkan s_2 adalah interpretasi apa pun di mana kita tidak peduli dengan nilai p . Semua model $\Box(\Diamond p \wedge \Diamond \neg p)$ dapat direpresentasikan oleh $(s_2^* s_0 s_2^* s_1 s_2^* \cup s_2^* s_1 s_2^* s_0 s_2^*)$, dan setiap model sesuai dengan beberapa jalur dalam tabel.

Teorema berikut memberikan kebenaran Algoritma 10.3.13.

Teorema 10.3.16 Suatu rumus LTL A dapat dipenuhi jika dan hanya jika Algoritma 10.3.13 mengembalikan "dapat dipenuhi."

Bukti Pada awalnya, algoritma akan berhenti karena (1) konstruksi tabel akan berhenti (Teorema 10.3.6 (a)), dan (2) memeriksa apakah simpul awal ditandai mati membutuhkan sejumlah langkah yang terbatas.

Jika tabel memiliki simpul terbuka, A dapat dipenuhi menurut Teorema 10.3.7.

Jika simpul awal tidak ditandai mati, kita menunjukkan bahwa ada model untuk A . Model ini dibangun dari jalur yang dimulai dengan simpul awal dan berisi semua simpul yang dibutuhkan oleh kondisi pemenuhan.

Jalur dipilih di antara simpul yang tidak ditandai mati sebagai berikut: (1) jalur dimulai dengan simpul awal; (2) jika simpul terakhir dari jalur saat ini hanya memiliki satu penerus yang belum ditandai, tambahkan penerus tersebut ke jalur; (3) jika jalur saat ini berisi simpul n dan $\Diamond A$ di n belum terpenuhi, temukan simpul n' yang dapat dijangkau dari n dan berisi A . Jika n' tidak muncul di jalur, tambahkan jalur dari n ke n' ke jalur saat ini. Jika n' sudah muncul di jalur, tambahkan hanya simpul yang diperlukan sehingga semua simpul dari siklus yang berisi

n dan n' ada di jalur. Artinya, bagian terakhir dari jalur berisi daftar simpul dari siklus yang sama, dan mereka dapat dijangkau satu sama lain.

Perhatikan bahwa jika $\Diamond A$ dan $\Diamond B$ sama-sama muncul di $F(n)$, kita akan mengerjakannya satu per satu: ketika $\Diamond A$ terpenuhi di n' , maka $\Diamond B$ telah terpenuhi di jalur hingga n' atau $\Diamond B$ masih ada di $F(n')$. Dalam kasus terakhir, ada simpul lain n'' yang dapat dijangkau dari n' dan berisi B (jika tidak, n' akan ditandai mati).

Dengan mengulangi seleksi, diperoleh jalur yang berakhir dengan siklus dan semua rumus bentuk $\Diamond C$ terpenuhi. Misalkan $n_{k_0}, n_{k_1}, \dots, n_{k_m}$ adalah simpul X di jalur dan $s_0 s_1 \dots, s_m$ adalah interpretasi dari simpul X ini (dengan tetap mempertahankan urutan yang sama), kita mendefinisikan urutan interpretasi π sebagai berikut:

$$\pi = s_0 s_1 s_2 \dots (s_j \dots s_m)^+,$$

di mana $s_j, \dots, s_m$ adalah interpretasi yang dihasilkan dari simpul X dalam siklus. Siap untuk memeriksa bahwa π adalah model dari rumus awal dengan induksi struktural pada rumus di setiap simpul.

Kami mengklaim bahwa $\pi_{\geq i}$ adalah model dari $F(n_{k_i})$ untuk $0 \leq i \leq m$. Untuk setiap simpul n selain simpul X , jika n_{k_i} adalah simpul X pertama yang mengikuti n dalam jalur untuk beberapa i , maka $F(n) \equiv F(n_{k_i})$ dan $\pi_{\geq i}$ juga merupakan model dari $F(n)$. Untuk setiap rumus $A \in F(n)$, pernyataan berikut ini benar:

- Jika A tidak memiliki operator temporal, maka A benar dalam $\pi_{\geq i}$.
- Jika A adalah $\circ B$, maka A benar dalam $\pi_{\geq i}$ karena B muncul di simpul n' setelah n_k menurut aturan X dan B benar dalam model n' menurut hipotesis induksi (karena B lebih kecil dari $\circ B$ dan model ini adalah sufiks langsung dari $\pi_{\geq i}$).
- Jika A adalah $\Diamond B$, maka A benar dalam $\pi_{\geq i}$ karena B muncul di simpul yang dapat dijangkau n' dari n , B benar dalam model untuk n' menurut hipotesis induksi (B lebih kecil dari A), dan model ini adalah sufiks dari $\pi_{\geq i}$.
- Jika A adalah $\Box B$, maka A benar dalam $\pi_{\geq i}$ karena B benar dalam $\pi_{\geq i}$ (menurut aturan α untuk $\Box$) dan $\Box B$ juga benar dalam sufiks langsung dari $\pi_{\geq i}$ (menurut aturan X).

Di sisi lain, jika simpul awal ditandai mati, maka A tidak dapat dipenuhi karena untuk setiap simpul n yang ditandai mati, $F(n)$ tidak dapat dipenuhi. Pernyataan ini dapat dibuktikan dengan induksi sesuai dengan definisi rekursif bahwa " n ditandai mati." Ada tiga kasus yang perlu dipertimbangkan:

1. n tertutup: $F(n)$ berisi sepasang literal komplementer dan tidak dapat dipenuhi.
2. semua penerus n ditandai mati: Dengan menggunakan hipotesis induksi, rumus-rumus dalam penerus tidak dapat dipenuhi. $F(n)$ tidak dapat dipenuhi berdasarkan Teorema 10.3.6 (b) dan (c).
3. $\Diamond A \in F(n)$ dan tidak ada simpul yang dapat dijangkau dari n dan mengandung A : $\Diamond A$ tidak dapat benar dalam urutan interpretasi π apa pun, karena setiap model sesuai dengan beberapa jalur dalam tabel. Jika A tidak muncul dalam jalur apa pun yang

dimulai dari n , A tidak dapat benar dalam sufiks apa pun dari π . Oleh karena itu, $\Diamond A$ salah di setiap π .

Teorema di atas memastikan bahwa Algoritma 10.3.13 adalah prosedur pengambilan keputusan untuk LTL. Dalam literatur, terdapat prosedur pengambilan keputusan lain berdasarkan automata Büchi (BA), yang merupakan perluasan dari automata keadaan terbatas untuk string input dengan panjang tak terbatas. Untuk LTL, string input ini mewakili urutan interpretasi. Diberikan sebuah tabel, automata Büchi dapat dengan mudah dibangun dari tabel tersebut dengan hanya menyimpan simpul X dan simpul terbuka (yang tidak ditandai mati) sebagai keadaan. Transisi dari satu keadaan ke keadaan lain ditambahkan jika ada jalur dari simpul yang sesuai ke simpul lain dalam tabel. Bahasa yang dikenali oleh BA harus berupa himpunan model dari formula LTL.

Terdapat algoritma yang menerjemahkan formula LTL ke automata Büchi tanpa tabel. Algoritma-algoritma ini berbeda dalam strategi konstruksinya, tetapi semuanya memiliki prinsip dasar yang sama, yaitu, setiap keadaan dalam automata yang dibangun mewakili sekumpulan rumus LTL yang diharapkan akan dipenuhi oleh sisa string input setelah terjadinya keadaan tersebut selama eksekusi BA. Automata Büchi memiliki aplikasi penting dalam pemeriksaan model keadaan terbatas untuk verifikasi formal program konkuren, dan presentasinya berada di luar cakupan buku ini.

10.4 OPERATOR TEMPORAL BINER

Dalam logika temporal, terkadang kita ingin mengekspresikan hubungan dua pernyataan dalam hal waktu. Misalnya, setelah dimulai, komputer akan terus berjalan sampai dihentikan. Untuk mengekspresikan kalimat ini, kita tulis $start \rightarrow \circ (running \ U \ halted)$ di mana operasi temporal biner U memiliki arti *sampai*.

Operator until dan release

Arti intuitif dari $(A \ U \ B)$ adalah bahwa A harus benar setidaknya sampai B menjadi benar dan B harus berlaku pada keadaan saat ini atau keadaan mendatang.

Operator U memiliki operator ganda yang disebut pelepasan, yang dilambangkan dengan $\mathcal{R}$. Arti intuitif dari $A \ \mathcal{R} \ B$ adalah bahwa B harus benar sampai (dan termasuk) keadaan di mana A pertama kali menjadi benar; jika A tidak pernah menjadi benar, B harus tetap benar selamanya.

Kita akan memperluas LTL dengan menambahkan U dan $\mathcal{R}$ pada definisi rumus LTL sehingga jika A dan B adalah rumus LTL, maka $(A \ U \ B)$ dan $(A \ \mathcal{R} \ B)$ juga merupakan rumus LTL. Kita juga perlu memperluas Definisi 10.2.1 dengan kondisi berikut:

Diberikan urutan interpretasi π , sebuah rumus LTL C benar dalam π jika

- C adalah $A \ U \ B$ dan terdapat $i \geq 0, \pi_{\geq i} \models B$ dan untuk semua $0 \leq j < i, \pi_{\geq j} \models A$.
- C adalah $A \ \mathcal{R} \ B$ dan baik $\Box B$ atau terdapat $i \geq 0, \pi_{\geq i} \models A$ dan untuk semua $0 \leq j \leq i, \pi_{\geq j} \models B$.

Dalam definisi di atas, jika π adalah urutan reguler, yaitu, π adalah $s_0 s_1 s_2 \cdots s_k (s_{k+1} s_{k+2} \cdots s_{k+m})^*$, maka kita dapat memeriksa apakah π adalah model dari C sebagai berikut:

- C adalah $A \mathcal{U} B$ dan terdapat $0 \leq i \leq k + m, \pi_{\geq i} \models B$ dan untuk semua $0 \leq j < i, \pi_{\geq j} \models A$.
- C adalah $A \mathcal{R} B$ dan baik $\Box B$ atau terdapat $0 \leq i \leq k + m, \pi_{\geq i} \models A$ dan untuk semua $0 \leq j \leq i, \pi_{\geq j} \models B$

Aturan-aturan di atas berguna dalam pendekatan pengecekan model.

Contoh 10.4.1 Menurut definisi, $p \mathcal{U} q$ benar dalam π berarti q pada akhirnya akan benar dan p harus benar sebelum q menjadi benar. Misalkan $s_0 = \{p, q\}, s_1 = \{p, \neg q\}, s_2 = \{\neg p, q\}$, dan $s_3 = \{\neg p, \neg q\}$. Kemudian $p \mathcal{U} q$ benar di $s_0^+, s_1 s_0^+, s_1^k s_0^+$ untuk $k \geq 0, s_1^k s_2^+$ untuk $k \geq 0$, dan $s_2 s_3^+$; untuk pemeriksaan model, k dapat memilih berbagai nilai kecil untuk memastikan keandalan output. $p \mathcal{U} q$ salah di s_1^+, s_3^+ , dan $s_1^k s_3^+$ untuk $k \geq 0$.

Teorema berikut menampilkan beberapa sifat $\mathcal{U}$.

Teorema 10.4.2 Untuk setiap rumus LTL A dan B

(a) (idempotency)	$A \mathcal{U} A \equiv A$
(b) (abbreviation)	$\Diamond A \equiv \top \mathcal{U} A$
(c) (absorption)	$A \mathcal{U} (A \mathcal{U} B) \equiv A \mathcal{U} B$
(d) (absorption)	$(A \mathcal{U} B) \mathcal{U} B \equiv A \mathcal{U} B$
(e) (distributivity)	$\circ (A \mathcal{U} B) \equiv (\circ A) \mathcal{U} (\circ B)$
(f) (induction)	$A \mathcal{U} B \equiv B \vee (A \wedge \circ (A \mathcal{U} B))$

Bukti

- Untuk setiap urutan interpretasi $\pi, \pi_{\geq 0} = \pi$. Jika A benar pada $\pi_{\geq 0}$, maka $i = 0$, dan untuk $0 \leq j < i, A$ benar pada $\pi_{\geq j}$ secara hampa. Jadi, $A \mathcal{U} A$ benar pada $\pi_{\geq 0}$. Jika A salah pada $\pi_{\geq 0}$, maka $A \mathcal{U} A$ salah pada $\pi_{\geq 0}$ menurut definisi.
- Di sini, B adalah $\top$ dan C adalah A . Kebenaran $B \mathcal{U} C$ direduksi menjadi kondisi pertama bahwa terdapat $i \geq 0, \pi_{\geq i} \models C$, identik dengan $\Diamond C$.
- Untuk setiap urutan interpretasi π , jika $\Diamond B$ salah, maka baik $A \mathcal{U} (A \mathcal{U} B)$ dan $A \mathcal{U} B$ salah pada π . Jika $\Diamond B$ benar, misalkan k adalah keadaan sedemikian sehingga B benar pada $\pi_{\geq k}$ tetapi salah pada $\pi_{\geq j}$ untuk $0 \leq j < k$. Karena B benar pada $\pi_{\geq k}, A \mathcal{U} B$ secara trivial benar pada $\pi_{\geq k}$. $A \mathcal{U} B$ benar pada π jika dan hanya jika A benar pada $\pi_{\geq j}$ untuk $0 \leq j < k$. $A \mathcal{U} (A \mathcal{U} B)$ benar pada π jika dan hanya jika A benar pada $\pi_{\geq j}$ untuk $0 \leq j < k$ juga.

Bukti (d) dan (e) diserahkan sebagai latihan.

- Agar $A \mathcal{U} B$ benar, B harus benar sekarang, atau kita menunda persyaratan untuk memenuhi $A \mathcal{U} B$ ke langkah berikutnya sambil mensyaratkan bahwa A benar sekarang.

Contoh 10.4.3 Menurut definisi, $p \mathcal{R} q$ benar di π berarti q benar selamanya, atau setelah p menjadi benar, kondisi q menjadi benar dilepaskan. Misalkan $s_0 = \{p, q\}$, $s_1 = \{p, \neg q\}$, $s_2 = \{\neg p, q\}$, dan $s_3 = \{\neg p, \neg q\}$. Maka $p \mathcal{R} q$ benar di $s_0^+, s_2^+, s_2 s_0^+$, dan $s_2 s_0 s_3^+$.
 $p \mathcal{R} q$ salah di $s_1^+, s_1^k s_0^+$ untuk $k \geq 1$, $s_1^k s_2^+$ untuk $k \geq 1$, $s_2 s_1^+$, $s_2 s_4^+$, dan s_3^+ .

Teorema berikut menampilkan beberapa sifat R .

Teorema 10.4.4 Untuk setiap rumus LTL A dan B

(a) (idempotency)	$A \mathcal{R} A \equiv A$
(b) (abbreviation)	$\perp \mathcal{R} A \equiv \Box A$
(c) (absorption)	$A \mathcal{R} (A \mathcal{R} B) \equiv A \mathcal{R} B$
(d) (absorption)	$(A \mathcal{R} B) \mathcal{R} B \equiv A \mathcal{R} B$
(e) (distributivity)	$\circ (A \mathcal{R} B) \equiv (\circ A) \mathcal{R} (\circ B)$
(f) (induction)	$A \mathcal{R} B \equiv B \vee (A \wedge \circ (A \mathcal{R} B))$
(g) (duality)	$A \mathcal{R} B \equiv \neg(\neg A \mathcal{U} \neg B)$

Bukti (a)–(f) dapat disimpulkan dari (g) dan Teorema 10.4.2. Untuk menyederhanakan bukti (g), kita buktikan $p \mathcal{R} q \equiv \neg(\neg p \mathcal{U} \neg q)$, di mana p dan q adalah variabel proposisional. Mengganti $\pi(i)$ dengan $\pi_{\geq i}$ dalam bukti berikut akan memberi kita bukti $A \mathcal{R} B \equiv \neg(\neg A \mathcal{U} \neg B)$ dengan teorema substitusi.

Asumsikan $p \mathcal{R} q$ benar di π . Jika q selalu benar, yaitu $\Box q$ benar, maka $\Diamond \neg q$ dan $\neg p \mathcal{U} \neg q$ salah. Dengan demikian, $\neg(\neg p \mathcal{U} \neg q)$ benar di π . Jika q tidak selalu benar, misalkan k minimal sedemikian sehingga q salah di $\pi(k)$. Karena $p \mathcal{R} q$ bernilai benar di π dan q bernilai salah di $\pi(k)$, berdasarkan semantik $\mathcal{R}$, harus ada $i < k$ sedemikian sehingga p bernilai benar di $\pi(i)$ dan untuk semua $0 \leq j \leq i$, q bernilai benar di $\pi(j)$. Dalam hal ini, mari kita periksa nilai kebenaran $\neg p \mathcal{U} \neg q$: karena $\pi(k)$ adalah interpretasi pertama di mana $\neg q$ bernilai benar dan $\neg p$ bernilai salah di $\pi(i)$ untuk $i < k$, $\neg p \mathcal{U} \neg q$ harus bernilai salah di π berdasarkan semantik $\mathcal{U}$. Oleh karena itu, $\neg(\neg p \mathcal{U} \neg q)$ harus bernilai benar di π .

Sekarang, anggap $p \mathcal{R} q$ bernilai salah di π . Berdasarkan semantik $\mathcal{R}$, $\Box q$ tidak mungkin bernilai benar dan misalkan k minimal sedemikian sehingga q bernilai salah di $\pi(k)$. Oleh karena itu, q bernilai benar di $\pi(j)$ untuk $0 \leq j < k$. Karena $p \mathcal{R} q$ bernilai salah di π , berdasarkan semantik $\mathcal{R}$, p harus bernilai salah di $\pi(j)$ untuk $0 \leq j < k$. Secara ekuivalen, $\neg p$ harus bernilai benar di $\pi(j)$ untuk $0 \leq j < k$. Oleh karena itu, $\neg p \mathcal{U} \neg q$ bernilai benar di π berdasarkan semantik $\mathcal{U}$. Jadi, kita simpulkan bahwa $\neg(\neg p \mathcal{U} \neg q)$ bernilai salah di π . Kedua kasus di atas melengkapi bukti bahwa $p \mathcal{R} q$ bernilai benar di π jika dan hanya jika $\neg(\neg p \mathcal{U} \neg q)$ bernilai benar di π untuk sembarang π .

Operator "until" lemah dan operator "lepas" kuat

Beberapa peneliti juga mendefinisikan operator biner "until" lemah, yang dilambangkan dengan $\mathcal{W}$, dengan semantik yang mirip dengan operator "until", tetapi kondisi

berhenti tidak diharuskan terjadi (mirip dengan pelepasan). Operator biner "lepas" kuat, yang dilambangkan dengan $\mathcal{S}$, adalah dual dari "until" lemah. Operator ini didefinisikan mirip dengan operator "until", sehingga kondisi pelepasan harus benar pada suatu titik. Oleh karena itu, operator ini lebih kuat daripada operator pelepasan.

Definisi 10.4.5 Operator temporal tambahan didefinisikan sebagai berikut:

- "**weak until**" $A \mathcal{W} B$: A harus benar setidaknya sampai B menjadi benar; jika B tidak pernah menjadi benar, A harus tetap benar selamanya.
- "**strong release**" $A \mathcal{S} B$: B harus benar sampai dan termasuk keadaan di mana A pertama kali menjadi benar, dan A harus benar pada keadaan saat ini atau keadaan di masa mendatang.

Alih-alih memperluas Definisi 10.2.1 untuk memasukkan makna formal dari $A \mathcal{W} B$ dan $A \mathcal{S} B$, kami memberikan hubungan yang setara di bawah ini sebagai definisi alternatif:

Teorema 10.4.6 Untuk setiap rumus LTL A dan B

$$\begin{aligned} (a) \quad (\text{weak until}) \quad A \mathcal{W} B &\equiv \Box A \vee (A \mathcal{U} B) \\ (b) \quad (\text{strong release}) \quad A \mathcal{S} B &\equiv \Diamond A \wedge (A \mathcal{R} B) \end{aligned}$$

Teorema di atas menunjukkan bahwa $\mathcal{W}$ dan $\mathcal{S}$ dapat didefinisikan dalam istilah $\mathcal{U}$ dan $\mathcal{R}$, sama seperti $\rightarrow$ dan $\leftrightarrow$ dapat didefinisikan dalam istilah $\vee$ dan $\wedge$. Kita tahu bahwa $\mathcal{R}$ dapat didefinisikan dalam istilah $\mathcal{U}$ melalui dualitas. Bahkan, salah satu dari empat operator temporal biner dapat digunakan untuk mendefinisikan tiga operator lainnya. Kita telah melihat dalam Teorema 10.4.2 dan 10.4.4 bahwa $\Diamond$ dan $\Box$ dapat didefinisikan dalam istilah $\mathcal{U}$ dan $\mathcal{R}$. Dengan demikian, himpunan minimal LTL hanya membutuhkan dua operator temporal: $\circ$ dan salah satu dari empat operator biner.

Teorema 10.4.7 Untuk setiap rumus LTL A dan B

$$\begin{aligned} (a) \quad A \mathcal{U} B &\equiv \Diamond B \wedge (A \mathcal{W} B) \\ (b) \quad A \mathcal{R} B &\equiv \Box B \vee (A \mathcal{S} B) \\ (c) \quad A \mathcal{U} B &\equiv B \mathcal{S} (A \vee B) \\ (d) \quad A \mathcal{R} B &\equiv B \mathcal{W} (A \wedge B) \\ (e) \quad A \mathcal{W} B &\equiv B \mathcal{R} (A \vee B) \\ (f) \quad A \mathcal{S} B &\equiv B \mathcal{U} (A \wedge B) \\ (g) \quad A \mathcal{W} B &\equiv A \mathcal{U} (A \vee B) \\ (h) \quad A \mathcal{S} B &\equiv A \mathcal{R} (A \wedge B) \\ (i) \quad \Box A &\equiv A \mathcal{W} \perp \\ (j) \quad \Diamond A &\equiv A \mathcal{S} \top \\ (k) \quad \neg(A \mathcal{S} B) &\equiv \neg A \mathcal{W} \neg B. \end{aligned}$$

Kesetaraan terakhir adalah dualitas $\mathcal{W}$ dan $\mathcal{S}$. Bukti teorema ini diserahkan sebagai latihan.

Perluasan Tabel Semantik

Dengan menggunakan dualitas dan aturan penulisan ulang $A \mathcal{W} B \rightarrow \Box A \vee (A \mathcal{U} B)$ dan $A \mathcal{S} B \rightarrow \Diamond A \wedge (A \mathcal{R} B)$, negasi dapat didorong ke bawah dan semua rumus LTL dapat ditransformasikan ke dalam bentuk normal negasi, di mana semua negasi hanya muncul di depan variabel proposisional, dan hanya operator temporal $\Box, \Diamond, \circ, \mathcal{U}$, dan $\mathcal{R}$ yang dapat muncul. Perhatikan bahwa transformasi ke bentuk normal negasi tidak memperbesar ukuran rumus.

Membangun tabel semantik untuk rumus yang menggunakan operator $\mathcal{U}$ dan $\mathcal{R}$ cukup mudah. Aturan β untuk $A \mathcal{U} B$ dan aturan α untuk $A \mathcal{R} B$ diberikan di bawah ini:

α	$\alpha1, \alpha2$	β	$\beta1$	$\beta2$
$A \mathcal{R} B$	$B, A \vee \circ (A \mathcal{R} B)$	$A \mathcal{U} B$	B	$A, \circ (A \mathcal{U} B)$

Aturan α didasarkan pada Teorema 10.4.4 (f), dan aturan β didasarkan pada Teorema 10.4.2 (f).

Setelah tabel dibangun menggunakan aturan-aturan ini, kita perlu memutuskan apakah sebuah model dapat dihasilkan dari tabel tersebut. Dalam hal ini, definisi node mati (Definisi 10.3.10) perlu diperluas untuk mempertimbangkan terjadinya $(A \mathcal{U} B)$ dalam sebuah node tabel: sebuah node n ditandai mati jika terdapat rumus $(A \mathcal{U} B) \in F(n)$ sedemikian sehingga tidak ada node n' yang dapat dijangkau dari n dan $B \in F(n')$.

10.5 VERIFIKASI PROGRAM KONKUREN

Bab ini dimulai dengan contoh bagaimana mengakses rekening bank secara konkuren dan benar. Contoh ini berfungsi sebagai motivasi untuk penerapan logika temporal untuk verifikasi formal program konkuren. Kita akan mengakhiri bab ini dengan contoh verifikasi formal, yang diambil dari tutorial sistem perangkat lunak **STeP** (**Stanford Temporal Prover**). STeP adalah sistem yang dikembangkan untuk mendukung verifikasi formal berbantuan komputer dari sistem konkuren dan reaktif berdasarkan logika Hoare dan logika temporal. Alat berdasarkan pemeriksaan model (MC) untuk tujuan yang sama. Teknik penalaran otomatis yang efisien seperti metode penyederhanaan, prosedur pengambilan keputusan, logika temporal, dll. digunakan dalam STeP dan beberapa pemeriksa model. Karena keterbatasan ruang, kita akan memperkenalkan pemeriksaan model secara singkat dan menyerahkan STeP kepada pembaca yang tertarik untuk menjelajahnya.

Pemeriksaan Model dengan Mesin Keadaan Terbatas

Pemeriksaan model adalah paradigma verifikasi formal umum untuk program imperatif dan fungsional. Dalam pemeriksaan model, desain perangkat keras/perangkat lunak/algoritma dimodelkan oleh mesin keadaan terbatas, properti ditulis dalam logika temporal proposisional, dan prosedur verifikasi adalah pencarian menyeluruh dalam ruang keadaan desain. Untuk menghindari ledakan ruang eksponensial, batasan digunakan untuk membatasi panjang jalur dalam mesin keadaan terbatas, sehingga disebut pemeriksaan model terbatas. Pemeriksaan model bertujuan untuk menemukan contoh tandingan diagnostik, seperti perilaku tak terduga

dari suatu rangkaian atau kemungkinan pelanggaran properti keselamatan. Jika tidak ditemukan contoh tandingan, verifikasi dianggap "lengkap."

Ketika LTL (logika temporal linier) digunakan dengan pemeriksaan model, mesin keadaan terbatas M adalah kerangka Kripke $K = (S, R)$, di mana S adalah himpunan keadaan, yang masing-masing merupakan interpretasi variabel proposisional dalam LTL, dan R adalah relasi biner atas S , ditambah informasi berikut:

- Himpunan bagian I yang tidak kosong dari S diidentifikasi sebagai keadaan awal M .
- Secara opsional, himpunan bagian F dari S diidentifikasi sebagai keadaan akhir M .
- Secara opsional, label diberikan kepada setiap transisi dalam R .

Contoh 10.5.1 Misalkan Anda diminta untuk mengimplementasikan loket tiket daring untuk menjual tiket suatu acara. Anda membuat prosedur berikut, di mana k adalah jumlah total tiket yang akan dijual dan *reserved* adalah variabel Boolean untuk "permintaan reservasi sedang diproses."

```

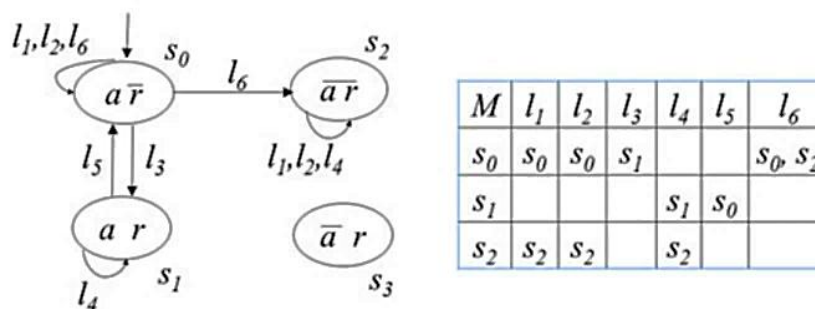
proc ticketing()
    precondition:  $nTickets = k > 0 \wedge reserved = false$ 
 $l_1$  while ( $true$ ) do
 $l_2$      if ( $nTickets > 0 \wedge \neg reserved$ )
 $l_3$           $reserved := true$ 
 $l_4$      if ( $nTickets > 0 \wedge reserved$ )
 $l_5$           $reserved := false$ 
 $l_6$           $nTickets := nTickets - 1$  // critical area
    od

```

Untuk mendeskripsikan program di atas dalam pemeriksaan model, kita memperkenalkan dua variabel proposisional: a untuk " $nTickets > 0$," yaitu, a bernilai benar jika dan hanya jika tiket tersedia dan r untuk reserved. Dengan demikian, untuk kerangka Kripke, $S = \{ar, a\bar{r}, \bar{a}r, \bar{a}\bar{r}\}$. Relasi R atas S ditentukan oleh eksekusi prosedur.

- Dengan prasyarat yang diberikan, satu-satunya keadaan awal adalah $a\bar{r}$.
- Setelah " $reserved := true$ " pada baris l_3 , keadaan berubah menjadi ar .
- Setelah " $reserved := false$ " pada baris l_5 , keadaan kembali ke $a\bar{r}$.
- Setelah " $nTickets := nTickets - 1$ " pada baris l_6 , keadaan baru memiliki dua pilihan: tetap di $a\bar{r}$ jika $nTickets > 0$ masih benar atau pergi ke $\bar{a}\bar{r}$ jika $nTickets > 0$ salah.
- Baris kode lainnya tidak mengubah keadaan, dan keadaan $\bar{a}r$ tidak dapat diakses dari keadaan awal $a\bar{r}$.

Berdasarkan analisis di atas, kami membuat mesin keadaan terbatas M yang ditunjukkan pada Gambar 10.4 untuk prosedur *ticketing*. Mesin ini memiliki tiga keadaan yang berguna: s_0, s_1 , dan s_2 , dan s_0 adalah keadaan awal. Label kode dilampirkan pada setiap transisi untuk membantu memahami bagaimana keadaan berubah selama eksekusi.



Gambar 10.4 Mesin keadaan terbatas M untuk *ticketing()*

Bagi pembaca yang sudah familiar dengan mesin keadaan terbatas, mesin M pada Gambar 10.4 adalah mesin nondeterministik, karena dari keadaan s_0 , transisi dapat menuju ke s_0 atau s_2 dengan label yang sama l_6 . Hal ini jelas dari representasi tabel M di sisi kanan gambar.

Di Bagian... 10.2, kami telah menunjukkan bahwa jalur tak terbatas dalam kerangka Kripke dapat dengan mudah direpresentasikan oleh urutan keadaan reguler, yaitu, $s_0 s_1 \dots s_k (s_{k+1} s_{k+2} \dots s_{k+m})^* \cdot$. Kami akan menggunakannya dalam mesin keadaan terbatas sebagai kandidat model rumus LTL.

Jika kita menggabungkan label yang diberikan pada setiap transisi (s_i, s_{i+1}) dari jalur tertentu, kita memperoleh string label, yang akan disebut jejak jalur *eksekusi*. Sebagai contoh, berikut ini adalah jalur M pada Gambar 10.5.1 dengan label eksplisit pada setiap transisi.

$$s_0 \xrightarrow{l_1} s_0 \xrightarrow{l_2} s_0 \xrightarrow{l_3} s_1 \xrightarrow{l_4} s_1 \xrightarrow{l_5} s_0 \xrightarrow{l_6} s_0 \xrightarrow{l_1} s_0 \xrightarrow{l_2} s_0 \xrightarrow{l_3} s_1 \xrightarrow{l_4} s_1 \xrightarrow{l_5} s_0 \xrightarrow{l_6} s_2$$

Jejak jalur ini adalah $l_1 l_2 l_3 l_4 l_5 l_6 l_1 l_2 l_3 l_4 l_5 l_6$, yang mewakili eksekusi prosedur *ticketing()* dengan $k = 2$. Dalam hal ini, kita mengatakan jalur tersebut adalah eksekusi *ticketing()*. Tidak semua jalur adalah eksekusi karena tidak semua jejak menunjukkan eksekusi program dengan benar.

Untuk program sekuensial, jejak harus mengikuti urutan sekuensial perintah. Untuk program konkuren, jejak harus berupa pengacakan atau penggabungan dua atau lebih jejak sekuensial. Mudah untuk memeriksa apakah suatu jalur adalah eksekusi. Untuk *ticketing*, eksekusi harus dimulai dengan l_1 , diikuti oleh l_2 , kemudian diikuti oleh l_3 atau l_4 , dan seterusnya. Untuk tujuan verifikasi, kita hanya mempertimbangkan eksekusi program. Untuk menjadi kandidat model LTL, suatu jalur harus tak terbatas, dan kita akan menggunakan urutan reguler untuk tujuan ini. Untuk pemeriksaan model terbatas, jalur dengan panjang terbatas sudah cukup.

Sifat-Sifat Program Konkuren

Misalkan prosedur *ticketing* dipanggil setiap kali pelanggan melakukan pemesanan online. Ini berarti bahwa beberapa salinan prosedur akan berjalan secara konkuren. Komputasi program konkuren dipandang sebagai penggabungan operasi atomik dari prosesnya, di mana setiap proses adalah program sekuensial dan memiliki variabel kontrol untuk menemukan kode eksekusi saat ini.

Prosedur konkuren menimbulkan banyak masalah yang tidak terlihat pada prosedur sekuensial. Misalnya, jika dua salinan ticketing menjalankan baris l_6 , yaitu $nTickets := nTickets - 1$ secara bersamaan, maka hasil akhirnya akan salah. Misalnya, jika $nTickets = 5$ sebelum eksekusi, dan kedua salinan akan menulis 4, bukan 3, ke dalam $nTickets$. Jika *reserved* adalah variabel global dan salinan 1 menjalankan l_5 tepat setelah salinan 2 menjalankan l_3 , maka pelanggan yang menjalankan salinan 2 tidak akan mendapatkan tiket. Prosedur konkuren harus memiliki properti berikut:

- *Saling eksklusif*: Eksekusi tidak pernah berada di bagian kritis (baris l_6 dari ticketing) pada waktu yang sama untuk beberapa proses.
- *Aksesibilitas*: Setelah suatu proses menyatakan minat untuk memasuki bagian kritis, proses tersebut pada akhirnya akan melakukannya. Untuk ticketing, jika suatu proses berada di l_3 , proses tersebut pada akhirnya akan selalu mencapai l_6 .
- *Pengambilalihan satu batas*: Jika satu proses ingin memasuki bagian kritis, proses lain dapat memasuki bagian kritis paling banyak satu kali sebelum proses pertama melakukannya.

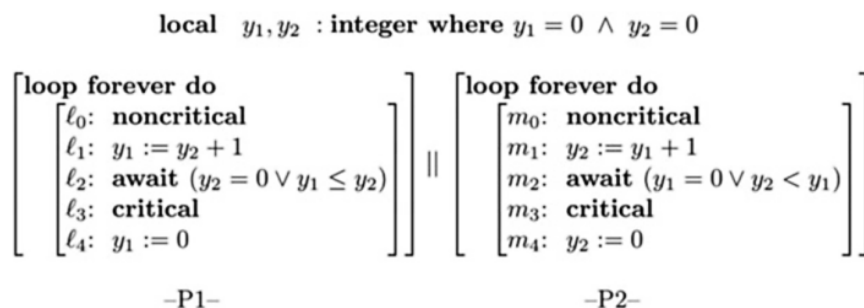
Saling eksklusif dan pengambilalihan satu batas termasuk dalam kelas properti keamanan temporal. Secara informal, properti tersebut menyatakan bahwa sesuatu yang "buruk" tidak akan pernah terjadi dan akan dipalsukan jika keadaan buruk pernah tercapai: jika rumus keamanan A salah dalam suatu model, maka ada awalan terbatas dari model tersebut sehingga A juga salah dalam setiap perluasan awalan ini.

Aksesibilitas, di sisi lain, adalah properti respons dan termasuk dalam kelas properti kemajuan yang lebih besar. Sifat-sifat ini menyatakan bahwa sesuatu yang "baik" dijamin akan terjadi pada akhirnya. Verifikasi setiap kelas sifat memiliki persyaratan khusus. Misalnya, sifat keamanan tidak bergantung pada persyaratan penghentian program yang diberikan, sedangkan verifikasi sifat respons bergantung pada penghentian.

Merancang program konkuren dengan sifat-sifat ini merupakan tantangan. Tampaknya, *ticketing()* tidak dapat digunakan sebagai program konkuren karena tidak memiliki salah satu dari tiga sifat di atas.

Program BAKERY(2)

Solusi awal untuk eksklusif bersama mengalami kelaparan, di mana suatu proses dapat selamanya ditolak aksesnya ke bagian kritis, dan beberapa solusi selanjutnya tidak memenuhi satu-bounded overtaking. Algoritma bakery Lamport, yang ditemukan oleh Leslie Lamport pada tahun 1974, adalah solusi pertama yang memenuhi tiga sifat di atas untuk proses N umum (dalam contoh kita, $N = 2$).



Gambar 10.5 Program BAKERY(2)

Gambar 10.5 menunjukkan program yang diambil dari tutorial STeP yang mengimplementasikan algoritma roti Lamport untuk eksklusivitas bersama. Dua proses, P_1 dan P_2 , mengkoordinasikan akses ke bagian kritis, di mana paling banyak satu proses harus berada pada waktu tertentu. Misalnya, memodifikasi saldo rekening bank harus dilakukan di bagian kritis. Setiap proses memilih "tiket" yang merupakan angka di y_1 dan y_2 , masing-masing, seperti halnya pelanggan di bank, dan proses dengan tiket yang lebih rendah diizinkan untuk memasuki bagian kritis. Awalnya, $y_1 = y_2 = 0$. Tiket dengan angka 0 menunjukkan bahwa proses tersebut tidak tertarik untuk mengakses bagian kritis. Karena hanya ada dua proses, dapat dibuktikan bahwa $y_1, y_2 \in \{0, 1, 2\}$. Deskripsi informal algoritma ini dapat dibuat lebih tepat jika sistem dan propertinya dimodelkan secara formal.

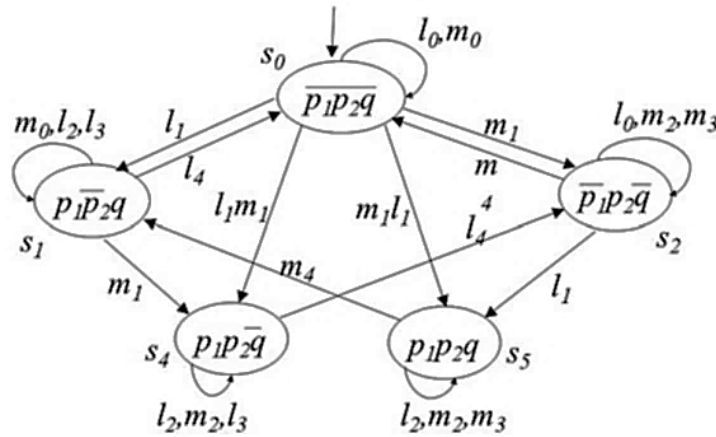
Arti dari "**await** ($y_2 = 0 \vee y_1 \leq y_2$)" adalah bahwa eksekusi berhenti sementara hingga kondisi ($y_2 = 0 \vee y_1 \leq y_2$) benar. Ini sama dengan **while** $\neg(y_2 = 0 \vee y_1 \leq y_2)$ /* tidak melakukan apa pun */

Untuk menggambarkan eksekusi BAKERY(2) dalam LTL, kita memperkenalkan tiga variabel proposisional, yaitu p_1, p_2 , dan q , dengan arti bahwa p_1 adalah $y_1 > 0$, p_2 adalah $y_2 > 0$, dan q adalah $y_1 > y_2$. Kemudian kita membangun mesin keadaan terbatas berdasarkan variabel-variabel tersebut. Tiga variabel dapat menciptakan jumlah maksimum delapan keadaan. Namun, arti dari tiga variabel tersebut mengesampingkan tiga keadaan: $\overline{p_1} \overline{p_2} q$, $p_1 \overline{p_2} \overline{q}$, $\overline{p_1} p_2 q$. Oleh karena itu, mesin keadaan terbatas kita memiliki lima keadaan. Awalnya, $y_1 = y_2 = 0$; Oleh karena itu, p_1, p_2, q bernilai salah, yang mewakili keadaan awal. Setelah eksekusi BAKERY(2) baris demi baris, kita membuat transisi antar keadaan, seperti yang ditunjukkan pada Gambar 10.6. Secara tegas, ini bukanlah mesin keadaan terbatas yang kita lihat dalam teori komputasi, karena transisi dari s_0 ke s_4 dan s_5 memodelkan eksekusi baris ℓ_1 dan m_1 secara bersamaan dengan dua kemungkinan hasil ($y_1 > y_2$ bernilai benar atau salah).

Mengikuti konvensi dari Bagian 10.5, kita menyebut jalur dalam mesin keadaan terbatas sebagai sebuah run, jika jejak jalur tersebut merepresentasikan eksekusi program. Untuk BAKERY(2), jejak sebuah run harus berupa "shuffle" atau "interleaving" dari dua string dari $(l_0^+ l_1 l_2^+ l_3 l_4)^*$ dan $(m_0^+ m_1 m_2^+ m_3 m_4)^*$, di mana l_0^+ (masing-masing l_2^+, m_0^+, m_2^+) berarti l_0 dapat diulang satu kali atau lebih, untuk memodelkan efek bekerja di bagian nonkritis atau perintah **await**. Misalnya, urutan $l_0 l_1 l_2 l_3 m_0 m_1 l_4 m_2 m_3 m_4$ menunjukkan bahwa P_1 memasuki bagian kritis sekali, sementara P_2 memulai bagian kritisnya sendiri pada saat itu.

Run yang terkait dengan jejak ini adalah $s_0s_0s_1s_1s_1s_1s_4s_2s_2s_0$, yang dapat diverifikasi di bawah ini:

$$s_0 \xrightarrow{l_0} s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_1 \xrightarrow{l_3} s_1 \xrightarrow{m_0} s_1 \xrightarrow{m_1} s_4 \xrightarrow{l_4} s_2 \xrightarrow{m_2} s_2 \xrightarrow{m_3} s_2 \xrightarrow{m_4} s_0$$



Gambar 10.6 Mesin keadaan terbatas M untuk BAKERY(2) (Gambar 1 dari [2])

Salah satu keunggulan mesin keadaan terbatas seperti yang ditunjukkan pada Gambar 10.6 adalah eksekusi program dapat dijelaskan baik oleh jejak atau transisi keadaan (yang akan dibahas dalam LTL). Sekarang, mudah untuk mengekspresikan tiga properti yang diinginkan dalam LTL:

- *Saling eksklusif*: Baris l_3 dan m_3 tidak dapat dieksekusi secara bersamaan. Untuk M pada Gambar 10.6, ini meminta agar l_3m_3 atau m_3l_3 tidak dapat muncul dalam string label dari setiap eksekusi (ini dapat diperiksa dengan pencarian graf yang menyeluruh). Untuk membuktikan properti ini dengan LTL, kita melihat bahwa satu-satunya tempat di mana label l_3 muncul adalah pada keadaan s_1 dan s_4 dan kondisi keberadaannya adalah $p_1\bar{p}_2q \vee p_1p_2\bar{q}$ atau, secara ekuivalen, $p_1(\bar{p}_2q \vee p_2\bar{q})$. Demikian pula, satu-satunya tempat di mana label m_3 muncul adalah pada keadaan s_2 dan s_5 dan kondisi keberadaannya adalah $\bar{p}_1p_2\bar{q} \vee p_1p_2q$ atau, secara ekuivalen, $p_2(\bar{p}_1\bar{q} \vee p_1q)$. Untuk pengecualian bersama, kedua kondisi eksistensi tidak dapat benar secara bersamaan. Dalam LTL, kita mensyaratkan bahwa

$$\Box (\overline{p_1(\bar{p}_2q \vee p_2\bar{q})p_2(\bar{p}_1\bar{q} \vee p_1q)})$$

yang dapat dengan mudah dibuktikan kebenarannya di LTL.

- *Aksesibilitas*: Jika kontrol berada di l_1 (atau m_1), kontrol tersebut pada akhirnya akan mencapai l_3 (atau m_3). Satu-satunya hambatan bagi kontrol untuk berpindah dari l_1 ke l_3 adalah l_2 : **await**($y_2 = 0 \vee y_1 \leq y_2$). Setelah l_1 dieksekusi, p_1 menjadi benar. Untuk melewati l_2 , kita membutuhkan ($y_2 = 0 \vee y_1 \leq y_2$), atau setara dengan, $p_2 \vee q$, menjadi benar. Jadi, di LTL, aksesibilitas BAKERY(2) dinyatakan sebagai

$$\Box(p_1 \rightarrow \Diamond(\overline{p_2} \vee \overline{q})) \wedge \Box(p_2 \rightarrow \Diamond(\overline{p_1} \vee q)),$$

Kita tidak dapat membuktikan validitas rumus di atas karena kita tidak memiliki aksioma tentang p_1, p_2 , dan q . Namun, kita memiliki mesin keadaan terbatas pada Gambar 10.6 yang memodelkan eksekusi program. Untuk menunjukkan aksesibilitas, kita menunjukkan bahwa negasi dari rumus di atas tidak memiliki model yang dapat diterima dalam mesin keadaan terbatas.

Mari kita pertimbangkan $\bar{A}$ terlebih dahulu, di mana A adalah $\Box(p_1 \rightarrow \Diamond(\overline{p_2} \vee \overline{q}))$. $\bar{A}$ dapat disederhanakan menjadi $\Diamond(p_1 \wedge \Box(p_2 \wedge q))$. Satu-satunya model $\bar{A}$ dalam M adalah urutan reguler yang diakhiri dengan s_5^* . Namun, mereka dibuang karena bukan run. Demikian pula, $\bar{B}$, di mana B adalah $\Box(p_2 \rightarrow \Diamond(\overline{p_1} \vee q))$, hanya memiliki model yang diakhiri dengan s_4^* , dan mereka dibuang karena alasan yang sama.

Sifat aksesibilitas mengesampingkan kasus kelaparan, yang menghasilkan jejak dalam bentuk $\dots l_0 l_1 (m_0 m_1 m_2 m_3 m_4)^*$. Sifat selanjutnya meniadakan kasus di mana m_3 muncul lebih dari sekali antara l_1 dan l_4 dalam jejak apa pun.

- *Menyalip satu kali*: Sifat ini mudah diungkapkan oleh sifat jejak BAKERY(2): dalam jejak apa pun dari suatu run, antara l_1 ($P1$ membuat permintaan untuk memasuki bagian kritis) dan l_4 ($P1$ meninggalkan bagian kritis), m_3 ($P2$ memasuki bagian kritis) dapat muncul paling banyak satu kali. Sifat dual diperoleh dengan menukar $P1$ dan $P2$. Karena l_1 (masing-masing m_1) dikaitkan dengan p_1 (masing-masing p_2) menjadi benar dan l_4 (masing-masing m_4) dikaitkan dengan p_1 (masing-masing p_2) menjadi salah, kita dapat menggunakan rumus berikut untuk menyatakan sifat menyalip satu kali:

$$\Box (p_1 \rightarrow (\overline{p_2} \mathcal{W} (p_2 \mathcal{W} (\overline{p_2} \mathcal{W} \overline{p_1}))))).$$

Rumus ini menyatakan bahwa setiap kali kontrol berada di l_2 (p_1 benar), yang berarti $P1$ ingin memasuki bagian kritis, maka hal berikut harus terjadi: mungkin ada interval di mana $P2$ tidak berada di bagian kritis (sehingga semua keadaan dalam interval tersebut memenuhi $\overline{p_2}$), diikuti oleh interval di mana $P2$ berada di bagian kritis (keadaan yang memenuhi p_2), diikuti oleh interval di mana $P2$ kembali tidak berada di bagian kritis (keadaan yang memenuhi $\overline{p_2}$), dan akhirnya diikuti oleh keadaan di mana $P1$ meninggalkan bagian kritis (yaitu, l_4 , dan $\overline{p_1}$ salah). Dengan demikian, $P2$ dapat memasuki bagian kritis (m_3) paling banyak sekali sebelum $P1$ meninggalkan bagian kritis (p_1 menjadi salah).

Sekali lagi, kita dapat memeriksa bahwa negasi dari rumus di atas tidak memiliki model yang dapat diterima untuk mesin keadaan terbatas BAKERY(2).

Versi umum BAKERY(N), yang diberikan pada Gambar 10.7, adalah penyederhanaan berlebihan dari algoritma bakery Lamport karena ketika dua atau lebih proses memiliki nilai y positif yang sama, yaitu $y_i = y_j > 0$ untuk $i \neq j$ (artinya mereka membuat permintaan pada waktu yang sama), algoritma Lamport akan menggunakan $i < j$ untuk memecahkan

kebuntuan, sedangkan BAKERY(N) akan memungkinkan mereka untuk menyelesaikan penantian pada waktu yang sama. Akibatnya, properti eksklusivitas bersama dari BAKERY(N) tidak akan berlaku; dua properti lainnya dari BAKERY(2) dapat dibawa ke BAKERY(N). Sebagai latihan, Anda diminta untuk memodifikasi BAKERY(N) sehingga properti eksklusivitas bersama berlaku untuk BAKERY(N) yang dimodifikasi.

```

in      N : integer where N > 0
local  y : array [1..N] of integer where  $\forall i : [1..N]. y[i] = 0$ 
value  max : array [1..N] of (integer  $\rightarrow$  integer)


$$\begin{array}{l} N \\ \parallel \\ i = 1 \end{array} :: \left[ \begin{array}{l} \text{loop forever do} \\ \quad \ell_0: \text{noncritical} \\ \quad \ell_1: y[i] := 1 + \max(y) \\ \quad \ell_2: \text{await } \forall j : [1..N]. (i \neq j \rightarrow y[j] = 0 \vee y[i] \leq y[j]) \\ \quad \ell_3: \text{critical} \\ \quad \ell_4: y[i] := 0 \end{array} \right]$$


```

Gambar 10.7 Program BAKERY(N)

Terdapat sejumlah besar urutan eksekusi ketika $N > 2$ dalam BAKERY(N), karena komputasi program konkuren merupakan interleaving dari operasi atomik dari prosesnya. Seperti yang ditunjukkan oleh contoh di atas, sangat sulit secara manual untuk memverifikasi program konkuren secara formal dengan menghabiskan semua urutan interleaving, untuk memastikan bahwa program tersebut benar untuk setiap kemungkinan komputasi.

Untuk memodelkan BAKERY(N) untuk $N > 2$ dalam LTL, kita membutuhkan lebih banyak variabel proposisional. Misalnya, untuk $N = 3$, kita membutuhkan sembilan variabel: p_1 (menunjukkan $y_1 > 0$), $p_2(y_2 > 0)$, $p_3(y_3 > 0)$, $q_1(y_1 > y_2)$, $q_2(y_2 > y_3)$, $q_3(y_3 > y_1)$, $r_1(y_1 = y_2)$, $r_2(y_2 = y_3)$, dan $r_3(y_1 = y_3)$. Untuk membantu memahami kode ketika $N = 3$, kita mengganti kondisi perintah await pada baris ℓ_2 untuk ketiga proses, $P1$, $P2$, dan $P3$, sebagai berikut:

$$\begin{aligned} P1: & (\overline{p_2} \vee \overline{q_1}) \wedge (\overline{p_3} \vee q_3 \vee r_3) \\ P2: & (\overline{p_1} \vee q_1 \vee r_1) \wedge (\overline{p_3} \vee \overline{q_2}) \\ P3: & (\overline{p_1} \vee \overline{q_3}) \wedge (\overline{p_2} \vee q_2 \vee r_2) \end{aligned}$$

Mudah untuk memeriksa bahwa ketika $y_1 = y_2 = y_3 = 1$, semua kondisi tunggu di $P1$, $P2$, dan $P3$ bernilai benar, sehingga memasuki area kritis pada saat yang bersamaan. Bug ini dapat diperbaiki dan ini diserahkan sebagai latihan.

Perhatikan bahwa jumlah interpretasi atas sembilan variabel adalah $2^9 = 512$. Mesin keadaan terbatas yang memodelkan eksekusi BAKERY(3) akan jauh lebih kompleks daripada yang ada pada Gambar 10.6. Jelas, alat perangkat lunak sangat diperlukan untuk keberhasilan verifikasi formal. Pembaca tidak boleh memiliki pandangan pesimistis terhadap verifikasi formal karena fokus verifikasi formal adalah algoritmanya. Setelah verifikasi formal selesai, verifikasi tersebut dapat digunakan dengan aman dalam banyak aplikasi. Memverifikasi bahwa

suatu implementasi mengikuti algoritmanya dengan benar adalah tugas yang relatif mudah, karena dapat diuraikan menjadi verifikasi baris demi baris.

Singkatnya, verifikasi formal menyediakan alat yang layak untuk memverifikasi kebenaran program konkuren. Presentasi kami terbatas pada LTL, dan ada pendekatan yang didasarkan pada teori automata dan pengecekan model, yang berada di luar cakupan buku ini.

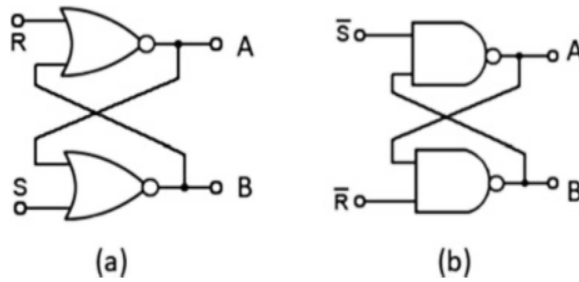
10.6 LATIHAN SOAL

1. Untuk rumus TL $A = p \rightarrow \Box p$, temukan dua kerangka Kripke sedemikian sehingga A bernilai benar di satu kerangka dan bernilai salah di kerangka lainnya.
2. Untuk rumus TL $A = \Box(p \vee q) \rightarrow (\Box p \vee \Box q)$, temukan dua kerangka Kripke sedemikian sehingga A bernilai benar di satu kerangka dan bernilai salah di kerangka lainnya.
3. Buktikan bahwa rumus TL berikut ini valid:

$$\Box(p \rightarrow q) \rightarrow (\Diamond p \rightarrow \Diamond q).$$

4. Buktikan bahwa kesetaraan berikut ini benar di TL:
 - (a) $\Box\Box p \equiv \Box p$
 - (b) $\Diamond\Diamond p \equiv \Diamond p$.
5. Jelaskan pernyataan-pernyataan berikut dalam LTL, menggunakan jumlah variabel proposisional seminimal mungkin:
 - (a) Jika pintu terkunci, pintu tidak akan terbuka sampai seseorang membukanya.
 - (b) Jika Anda menekan Ctrl-C, Anda akan mendapatkan prompt baris perintah.
 - (c) Gergaji tidak akan beroperasi kecuali pengaman diaktifkan.
6. Untuk rumus LTL $A = p \rightarrow \Box p$, temukan dua urutan interpretasi sedemikian sehingga A benar dalam satu dan salah dalam yang lain.
7. Untuk rumus LTL $A = \Box(p \vee q) \rightarrow (\Box p \vee \Box q)$, temukan dua urutan interpretasi sedemikian sehingga A benar dalam satu dan salah dalam yang lain.
8. Tidak seperti rangkaian kombinasi, rangkaian sekuensial memiliki "memori" dan merupakan blok bangunan dasar komputer. Rangkaian (a) pada Gambar 10.8 disebut *SR latch*, yang merupakan salah satu rangkaian sekuensial paling sederhana. Sebuah latch SR memiliki dua input Boolean, S dan R , dan dua output, A dan B . Nilai A dan B ditentukan tidak hanya oleh S dan R tetapi juga oleh nilai A dan B sebelumnya. Artinya, $A = \overline{R} \vee B'$ dan $B = \overline{S} \vee A'$, di mana A' dan B' adalah nilai "lama". Suatu keadaan latch *SR* ditentukan oleh nilai S, R, A , dan B . Dengan demikian, terdapat paling banyak 16 keadaan latch *SR*. Transisi antar keadaan ditentukan oleh perubahan S dan R . Dengan demikian, suatu keadaan memiliki empat penerus, termasuk dirinya sendiri. Misalnya, ketika $S = R = 1, A = B = 0$, keadaannya adalah $SR\overline{A}\overline{B}$ atau 1100 (keadaan ini tidak diperbolehkan dalam aplikasi latch *SR*). Ketika S berubah dari 1 menjadi 0, transisi terjadi dari 1100 ke 0101. Ketika kedua input berubah dari 11 menjadi 00, nilai output akan tidak pasti, dan Anda memerlukan beberapa transisi ke semua kemungkinan

keadaan. Harap berikan kerangka Kripke untuk memodelkan transisi dari latch SR pada Gambar 10.8a.



Gambar 10.8 (a) Latch SR dan (b) Flip-flop SR

9. Silakan ulangi latihan sebelumnya untuk rangkaian pada Gambar 10.8b, yang disebut flip-flop SR . Perhatikan bahwa input S dan R dinegasikan sebelum dimasukkan ke rangkaian, $A = \bar{S} \wedge B'$, dan $B = \bar{R} \wedge A'$, di mana A' dan B' adalah nilai "lama".
10. Tulis program Prolog `nnf(X,Y)`, yang mengambil rumus LTL X sebagai input dan menghasilkan Y sebagai bentuk normal negasi dari X .
11. Buktikan bahwa rumus-rumus berikut ini berlaku di LTL:
 - (a) $\Box p \rightarrow \Box \Box p$
 - (b) $\Diamond \Diamond p \rightarrow \Diamond p$
 - (c) $\Box \Diamond \Box p \rightarrow \Diamond \Box p$
 - (d) $\Box (p \rightarrow \Diamond q) \wedge \Box (q \rightarrow \Diamond r) \rightarrow \Box (p \rightarrow \Diamond r)$
12. Buktikan bahwa kesetaraan berikut ini berlaku di LTL:
 - (a) $\circ (p \wedge q) \equiv \circ p \wedge \circ q$
 - (b) $\circ (p \vee q) \equiv \circ p \vee \circ q$.
13. Buktikan dengan tableau semantik bahwa rumus LTL valid:
 - (a) $(\Box p \vee \Box q) \rightarrow (\Box p \vee \Box)$
 - (b) $\Diamond \Diamond p \rightarrow \Diamond p$
 - (c) $\Box \Diamond p \rightarrow \Diamond \Box p$
 - (d) $\Diamond \Box \Diamond p \rightarrow \Box \Diamond p$.
14. Temukan bentuk formal negasi yang setara untuk rumus-rumus berikut:
 - (a) $(A \mathcal{U} B) \leftrightarrow \Diamond B \wedge (A \mathcal{W} B)$
 - (b) $(A \mathcal{R} B) \leftrightarrow \Box B \vee (A \mathcal{S} B)$
 - (c) $(A \mathcal{U} B) \leftrightarrow B \mathcal{S} (A \vee B)$
 - (d) $(A \mathcal{R} B) \leftrightarrow B \mathcal{W} (A \wedge B)$
 - (e) $(A \mathcal{W} B) \leftrightarrow B \mathcal{R} (A \vee B)$
 - (f) $(A \mathcal{S} B) \leftrightarrow B \mathcal{U} (A \wedge B)$
 - (g) $(A \mathcal{W} B) \leftrightarrow A \mathcal{U} (A \vee B)$
 - (h) $(A \mathcal{S} B) \leftrightarrow A \mathcal{R} (\Diamond A \wedge B)$
 - (i) $\Box A \leftrightarrow A \mathcal{W} \perp$
 - (j) $\Diamond A \leftrightarrow A \mathcal{S} \top$
 - (k) $\neg(A \mathcal{S} B) \leftrightarrow \neg A \mathcal{W} \neg B$

15. Buktikan dengan menggunakan tableau semantik bahwa rumus LTL valid:

- (a) $p \mathcal{R} p \rightarrow p$
- (b) $\perp \mathcal{R} p \rightarrow \Box p$
- (c) $p \mathcal{R} (p \mathcal{R} q) \rightarrow p \mathcal{R} q$
- (d) $(p \mathcal{R} q) \mathcal{R} q \rightarrow p \mathcal{R} q$
- (e) $p \mathcal{R} q \rightarrow q \wedge (p \vee \circ (p \mathcal{R} q))$
- (f) $p \mathcal{R} q \rightarrow \neg(\neg p \mathcal{U} \neg q)$

16. Buktikan dengan tableau semantik bahwa rumus LTL valid:

- (a) $(p \mathcal{U} q) \leftrightarrow \Diamond q \wedge (p \mathcal{W} q)$
- (b) $(p \mathcal{R} q) \leftrightarrow \Box q \vee (p \mathcal{S} q)$
- (c) $(p \mathcal{U} q) \leftrightarrow q \mathcal{S} (p \vee q)$
- (d) $(p \mathcal{R} q) \leftrightarrow q \mathcal{W} (p \wedge q)$
- (e) $(p \mathcal{W} q) \leftrightarrow q \mathcal{R} (p \vee q)$
- (f) $(p \mathcal{S} q) \leftrightarrow q \mathcal{U} (p \wedge q)$
- (g) $(p \mathcal{W} q) \leftrightarrow p \mathcal{U} (p \vee q)$
- (h) $(p \mathcal{S} q) \leftrightarrow p \mathcal{R} (\Diamond p \wedge q)$
- (i) $\Box p \leftrightarrow p \mathcal{W} \perp$
- (j) $\Diamond p \leftrightarrow p \mathcal{S} \top$
- (k) $\neg(p \mathcal{S} q) \leftrightarrow \neg p \mathcal{W} \neg q$

17. Buktikan pernyataan berikut ini benar dalam LTL: (a) $(A \mathcal{U} B) \mathcal{U} B \equiv A \mathcal{U} B$ dan (b) $\circ (A \mathcal{U} B) \equiv (\circ A) \mathcal{U} (\circ B)$.

18. Tentukan berapa banyak variabel proposisional yang dibutuhkan untuk memodelkan BAKERY(N) (Gambar 10.7) dengan $N = 4$. Nyatakan kondisi baris l_2 untuk keempat proses dalam variabel-variabel ini.

19. (a) Modifikasi algoritma bakery BAKERY(N) pada Gambar 10.7 untuk memberikan deskripsi yang benar tentang algoritma bakery Lamport.

(b) Untuk $N = 3$, nyatakan kondisi perintah await dalam BAKERY(N) yang dimodifikasi sebagai ekspresi Boolean dari bahasa pemrograman di P1, P2, dan P3, masing-masing.

(c) Untuk $N = 3$, kita mendefinisikan variabel proposisional berikut: p_1 (menunjukkan $y_1 > 0$), p_2 ($y_2 > 0$), p_3 ($y_3 > 0$), q_1 ($y_1 > y_2$), q_2 ($y_2 > y_3$), q_3 ($y_3 > y_1$), r_1 ($y_1 = y_2$), r_2 ($y_2 = y_3$), dan r_3 ($y_1 = y_3$). Apakah kita membutuhkan kesembilan variabel tersebut untuk BAKERY(N) yang dimodifikasi? Jawablah pertanyaan ini dengan menyatakan kondisi tunggu P1, P2, dan P3 sebagai rumus dari kesembilan variabel tersebut.

20. Buktikan sifat eksklusif bersama dan aksesibilitas dari program yang diberikan di bawah ini:

$$\begin{array}{c}
\text{local } b_1, b_2, b_3 : \text{boolean where } b_1, b_2, b_3 \\
\left[\begin{array}{l} \text{loop forever do} \\ \ell_0: \text{noncritical} \\ \ell_1: (b_1, b_3) := (false, false) \\ \ell_2: \text{await } (b_2 \vee b_3) \\ \ell_3: \text{critical} \\ \ell_4: (b_1, b_3) := (true, true) \end{array} \right] \parallel \left[\begin{array}{l} \text{loop forever do} \\ m_0: \text{noncritical} \\ m_1: (b_2, b_3) := (false, true) \\ m_2: \text{await } (b_1 \vee \neg b_3) \\ m_3: \text{critical} \\ m_4: (b_2, b_3) := (true, b_1) \end{array} \right] \\
\text{--P1--} \qquad \qquad \qquad \text{--P2--}
\end{array}$$

BAB 11

MASALAH YANG DAPAT DIPUTUSKAN DAN TIDAK DAPAT DIPUTUSKAN

Dalam ilmu komputer, teori ketidakpastian mempelajari masalah-masalah yang berada di luar kemampuan komputer dan merupakan bagian dari teori komputabilitas. Dalam logika, ketidakpastian berkaitan dengan masalah apa yang dapat dihitung atau tidak dapat dihitung, atau rumus mana yang dapat dibuktikan, disangkal, atau tidak dapat dibuktikan. Misalnya, semua masalah logika proposisional yang dibahas dalam buku ini dapat dihitung. Di sisi lain, beberapa masalah dalam logika orde pertama tidak dapat dihitung, dan kita akan melihat beberapa di antaranya sepanjang bab ini.

Tujuan bab ini adalah untuk memperkenalkan konsep formal masalah keputusan yang dapat diputuskan dan tidak dapat diputuskan, serta fungsi yang dapat dihitung dan tidak dapat dihitung, dan menunjukkan bahwa beberapa masalah dalam logika orde pertama tidak dapat diputuskan atau tidak dapat dihitung. Kita akan membahas masalah yang dapat diputuskan dalam logika orde pertama di Bab 12, di mana kita akan menyajikan beberapa fragmen logika orde pertama yang dapat diputuskan.

11.1 SEJARAH TEORI KETIDAKPASTIAN

Teorema Ketidaklengkapan Gödel

Landasan teori ketidakpastian berasal dari dua teorema ketidaklengkapan Kurt Gödel:

1. Dalam sistem formal konsisten S apa pun yang dapat melakukan operasi aritmatika umum pada bilangan asli, beberapa rumus S tidak dapat dibuktikan benar atau salah di S .
2. Jika S konsisten, maka kita tidak dapat membuktikan di S bahwa S itu sendiri konsisten.

Sistem formal mirip dengan teori pembuktian yang diperkenalkan pada Bab 1 (Bagian 1.3).

Teori pembuktian terdiri dari dua komponen:

1. Sistem aksioma, yang mendefinisikan rumus mana yang benar (yaitu, teorema)
2. Prosedur pembuktian yang memberi tahu kita cara membuktikan teorema

Kita telah memperkenalkan empat sifat untuk prosedur pembuktian: konsistensi, kebenaran, kelengkapan, dan penghentian. Prosedur keputusan adalah prosedur pembuktian yang memiliki kebenaran dan penghentian. Teorema ketidaklengkapan Gödel pada dasarnya menyatakan bahwa tidak ada prosedur keputusan yang ada untuk setiap masalah keputusan dari sistem formal yang dapat melakukan operasi aritmatika umum.

Pada tahun 1931, pada usia 25 tahun, Gödel secara formal menerbitkan teorema ketidaklengkapan berdasarkan konsep konsistensi- ω . Penerimaan hasil Gödel beragam. Beberapa tokoh penting di bidang logika dengan cepat mengasimilasi hasilnya, tetapi juga terdapat banyak kesalahpahaman dan penolakan. Teorema ketidaklengkapan Gödel tidak hanya menandai tonggak sejarah dalam teori pembuktian dan teori komputabilitas, tetapi juga

memiliki dampak besar dalam studi teori himpunan dan logika modern (logika modal, logika intuisisionistik, logika kategorikal, dll.). Stanford Encyclopedia of Philosophy memiliki entri panjang tentang "teorema ketidaklengkapan Gödel" dengan informasi historis yang terperinci. Alih-alih memperkenalkan sistem formal Gödel, kami memilih untuk menyajikan mesin Turing, yang merupakan model komputasi formal, karena alasan berikut:

- Mesin Turing adalah model komputasi formal yang sangat sederhana, dan setiap langkah mesin Turing menyerupai langkah komputasi oleh manusia. Gödel menganggap pendekatan Turing lebih unggul daripada semua karya sebelumnya dan berkomentar "bahwa ini benar-benar definisi yang tepat tentang komputabilitas mekanis telah ditetapkan tanpa keraguan oleh Turing."
- Mesin Turing mudah dijelaskan dan mudah dipahami oleh orang-orang yang terbiasa dengan komputer. Komputer masa kini dapat dilihat sebagai implementasi dari mesin Turing universal, sebuah mesin Turing khusus yang diusulkan oleh Turing, yang menerima data dan kode program sebagai input, mengeksekusi kode pada data, dan melaporkan outputnya.
- Mesin Turing penting dalam teori komputabilitas karena tesis Church-Turing, yang menyatakan bahwa setiap model komputasi, yang ditemukan di masa lalu atau yang akan ditemukan di masa depan, dapat disimulasikan oleh mesin Turing.
- Sebagai hasil dari tesis Church-Turing, sistem formal Gödel dapat dengan mudah disimulasikan oleh mesin Turing. Teorema ketidaklengkapan pertama Gödel diterjemahkan menjadi teorema mesin Turing sebagai berikut: untuk setiap mesin Turing M dan setiap input w , tidak ada perangkat komputasi yang dapat memutuskan apakah M menerima w sebagai input dan berhenti dengan "berhasil" (terbukti), berhenti dengan "gagal" (tidak terbukti), atau berulang selamanya (tidak dapat dibuktikan). Artinya, penghentian M pada w tidak dapat diputuskan, yang merupakan masalah penghentian yang terkenal.
- Meskipun sifat inkonsistensi tidak didefinisikan untuk mesin Turing, teorema ketidaklengkapan kedua Gödel sesuai dengan teorema umum mesin Turing (teorema Rice): sifat-sifat nontrivial dan terkait bahasa dari mesin Turing tidak dapat diputuskan.

Tiga Model Komputasi Terkenal

Mesin Turing hanyalah salah satu dari banyak model komputasi formal yang pernah diusulkan. Pada tahun 1930-an, beberapa upaya independen dilakukan untuk memformalkan gagasan komputabilitas:

- Pada tahun 1931, Gödel membuktikan teorema ketidaklengkapan pertama untuk kelas yang disebut fungsi rekursif primitif. Pada tahun 1934, Gödel menciptakan kelas yang lebih umum yang disebut fungsi rekursif umum, juga disebut fungsi rekursif parsial. Kelas fungsi rekursif umum adalah kelas fungsi terkecil (mungkin dengan lebih dari satu argumen) yang mencakup semua fungsi konstan, proyeksi, dan fungsi penerus dan tertutup terhadap komposisi fungsi, rekursi, dan minimisasi. Jika minimisasi dikecualikan, maka itu menjadi kelas fungsi rekursif primitif. Pada tahun 1936, Stephen

Kleene memperkenalkan varian definisi Gödel dan mendefinisikan teori rekursi yang setara dengan fungsi rekursif umum.

- Pada tahun 1932, Alonzo Church menciptakan metode untuk mendefinisikan fungsi yang disebut kalkulus lambda, yang sering ditulis sebagai kalkulus- λ . Kemudian, Church mendefinisikan pengkodean bilangan asli yang disebut angka Church. Suatu fungsi pada bilangan asli disebut λ -komputabel jika fungsi yang sesuai pada angka Church dapat direpresentasikan oleh suatu suku dari kalkulus- λ .
- Pada tahun 1936, sebelum mengetahui kalkulus- λ Church, Alan Turing menciptakan model komputasi teoretis, yang sekarang disebut mesin Turing, yang dapat melakukan perhitungan dari input dengan memanipulasi simbol pada pita. Dengan diberikan pengkodean bilangan asli yang sesuai sebagai urutan simbol, suatu fungsi pada bilangan asli disebut Turing-komputabel jika terdapat mesin Turing yang dimulai dengan bilangan asli yang dikodekan sebagai input dan berhenti dengan hasil fungsi yang sesuai sebagai output.

Church dan Turing membuktikan bahwa ketiga kelas fungsi komputasi yang didefinisikan secara formal ini bertepatan: suatu fungsi adalah λ -komputabel jika dan hanya jika fungsi tersebut dapat dikomputasi Turing dan jika dan hanya jika fungsi tersebut merupakan fungsi rekursif umum. Hal ini telah membuat para matematikawan dan ilmuwan komputer percaya bahwa konsep komputabilitas secara akurat dicirikan oleh ketiga model komputasi yang setara ini. Upaya formal lainnya untuk mengkarakterisasi komputabilitas, termasuk teori rekursi Kleene dan sistem kanonik Post, kemudian memperkuat keyakinan ini. Karena kita tidak dapat mencakup semua kemungkinan model komputasi, tesis Church-Turing, meskipun memiliki penerimaan yang hampir universal, tidak dapat dibuktikan secara formal.

Saat ini, bahasa pemrograman yang didasarkan pada fungsi rekursif umum (atau kalkulus lambda atau teori rekursi Kleene) disebut bahasa pemrograman fungsional. Di sisi lain, bahasa pemrograman yang didasarkan pada mesin program tersimpan akses acak (RASP) disebut bahasa pemrograman imperatif. RASP adalah model komputasi yang setara kekuatannya dengan tiga model komputasi di atas dan merupakan contoh dari apa yang disebut arsitektur von Neumann, yang menggunakan memori akses acak dan program tersimpan. Lisp, ML, Clojure, dan Haskell adalah contoh dari yang pertama; C, C++, Java, dan Python adalah contoh dari yang terakhir.

Masalah Penghentian

Dalam teori komputabilitas, masalah penghentian adalah masalah menentukan, dari deskripsi program komputer dan input yang sembarang, apakah program tersebut akan selesai berjalan atau terus berjalan selamanya.

Turing membuktikan pada tahun 1936 bahwa algoritma umum untuk menyelesaikan masalah penghentian untuk semua pasangan (M, w) yang mungkin, di mana M adalah mesin Turing dan w adalah input ke M , tidak mungkin ada. Hasil ini merupakan gema dari teorema ketidaklengkapan pertama Gödel untuk mesin Turing dan merupakan salah satu kasus pertama dari masalah keputusan yang tidak dapat diputuskan. Ketidakpastian masalah

penghentian sangat penting bagi upaya komputasi praktis dan mendefinisikan kelas masalah yang tidak mungkin dapat diselesaikan secara pasti oleh perangkat komputasi mana pun.

Masalah penghentian bukan hanya masalah teoretis. Ini adalah masalah praktis bagi banyak peneliti. Dalam Contoh 7.3.13, dugaan Collatz yang belum terpecahkan terkait dengan apakah program di bawah ini berakhir untuk setiap bilangan bulat positif x :

```

proc Collatz( $x$ )
  while  $x > 1$  do
    if even( $x$ ) then  $x := x/2$  else  $x := (3 * x + 1)/2$ 
  return 1

```

Kode yang sama dapat diekspresikan sebagai sistem penulisan ulang:

$$\begin{aligned}
 f(1) &\rightarrow 1 \\
 f(2x) &\rightarrow f(x) \\
 f(2x + 1) &\rightarrow f(3x + 2)
 \end{aligned}$$

Penghentian sistem penulisan ulang di atas akan menyiratkan bahwa dugaan Collatz terpecahkan secara positif.

Dalam bab ini, kita akan memformalkan konsep-konsep yang disebutkan di atas seperti komputabilitas, decidabilitas, masalah yang tidak dapat diputuskan, dll., berdasarkan mesin Turing, dan menunjukkan bahwa masalah-masalah tertentu dalam logika tidak dapat diputuskan.

11.2 MESIN TURING

Mesin Turing adalah model matematika komputasi yang pertama kali diusulkan oleh Alan Turing pada tahun 1936. Mesin Turing memanipulasi simbol pada pita simbol sesuai dengan seperangkat aturan. Menurut tesis Church-Turing, setiap fungsi pada bilangan asli dapat dihitung dengan metode efektif jika dan hanya jika dihitung oleh mesin Turing. Dengan kata lain, meskipun modelnya sederhana, mesin Turing mampu mensimulasikan komputasi apa pun pada komputer saat ini, baik itu superkomputer atau ponsel pintar.

Turing berpendapat bahwa ketika manusia menghitung dengan pena dan kertas, pada setiap momen perhitungan, pikiran hanya dapat berada dalam salah satu dari sejumlah keadaan terbatas dan bahwa dalam setiap keadaan, dengan mempertimbangkan hasil sementara yang telah diperoleh sejauh ini, langkah perhitungan selanjutnya sepenuhnya ditentukan. Dan mesin Turing menangkap semua fitur penting dari komputer manusia.

Mesin Turing beroperasi pada pita yang berisi sejumlah sel tak terbatas dan setiap sel berisi satu simbol. Mesin memposisikan "kepalanya" di atas sebuah sel dan "membaca" simbol di sana. Sesuai dengan simbol dan keadaan mesin saat ini dalam serangkaian instruksi yang ditentukan pengguna, mesin beroperasi sebagai berikut:

- (a) Membaca simbol pita yang ditunjuk oleh kepala pita.

- (b) Mendapatkan instruksi yang ditentukan pengguna sesuai dengan simbol dari (a) dan keadaan saat ini. Jika tidak ada instruksi yang tersedia, berhenti dengan "kegagalan." Jika ada, lakukan (c)–(e) sesuai dengan instruksi.
- (c) Menulis simbol ke dalam sel saat ini.
- (d) Memindahkan kepala pita satu sel ke kiri atau ke kanan.
- (e) Ubah keadaan saat ini. Jika keadaan baru adalah “berhasil,” berhenti dengan “berhasil.”
- (f) Lanjutkan ke (a).

Pita awalnya diisi dengan string input simbol dan kosong di tempat lain. Mesin memposisikan kepalanya ke simbol pertama dengan keadaan yang ditentukan yang disebut “keadaan awal.” Mesin bergerak seperti yang dijelaskan di atas melalui langkah-langkah (a)–(f). Ada tiga kemungkinan hasil: (1) berhenti dengan “berhasil”; (2) berhenti dengan “ditolak”; atau (3) berulang selamanya.

Turing mampu menjawab pertanyaan mendasar secara negatif: apakah ada algoritma yang dapat menentukan apakah mesin Turing berhenti pada string input tertentu? Ini disebut “masalah penghentian” mesin Turing.

Mesin Turing dapat melakukan semua yang dapat dilakukan komputer sungguhan. Namun, desain minimalisnya membuatnya tidak cocok untuk komputasi dalam praktik: komputer dunia nyata didasarkan pada desain yang berbeda yang, tidak seperti mesin Turing, menggunakan memori akses acak. Bagi orang yang memahami pemrograman, Anda mungkin melihat pita mesin Turing sebagai susunan simbol yang sangat panjang dengan satu variabel indeks dari susunan tersebut (kepala pita). Indeks dapat bertambah atau berkurang 1 untuk mensimulasikan pergerakan kepala pita. Instruksi pengguna dapat disimpan dalam susunan dua dimensi yang diindeks oleh simbol pita dan simbol keadaan.

Meskipun mesin Turing dapat mengekspresikan komputasi sembarang, namun, bahkan mesin Turing pun tidak dapat menyelesaikan masalah tertentu. Dalam arti yang sangat nyata, masalah-masalah ini berada di luar batas teoritis komputasi dan menunjukkan adanya keterbatasan mendasar pada kekuatan komputasi mekanis.

Definisi Formal Mesin Turing

Mesin Turing (TM) menggunakan pita dengan jumlah sel yang tak terhingga, yang menyediakan memori tak terbatas untuk komputasi. Himpunan simbol yang muncul dalam pita dilambangkan dengan Γ , yang disebut alfabet pita. Secara default, Γ adalah himpunan bagian dari Σ , yaitu himpunan simbol yang digunakan dalam semua string input. Simbol kosong, $\sqcup$, berada di $\Gamma - \Sigma$. Σ^* menunjukkan himpunan semua string input (lihat Contoh 1.3.9) dan Γ^* menunjukkan himpunan semua isi pita tidak termasuk simbol kosong di akhir.

Untuk setiap string $w \in \Sigma^*$, misalkan $w_0 = \epsilon$ dan $w^{i+1} = ww^i$, di mana $i \geq 0$. Misalnya, $a^3 = aaa$ dan $(ab)^2 = abab$.

Contoh 11.2.1 Misalkan $\Sigma = \{a, b\}$ dan kita ingin merancang mesin Turing yang memeriksa apakah string masukan $w \in A_1 = \{a^j b^k \mid 0 \leq j \leq k\}$. Merancang mesin Turing sama dengan merancang algoritma yang bekerja pada array simbol. Jika mesin membaca a , kita dapat menghapusnya (dengan menulis simbol

lain ke dalam sel), yang sesuai dengan $j := j - 1$, dan kemudian mencari simbol b dan menghapusnya juga, yang sesuai dengan $k := k - 1$. Kode semu algoritma tersebut dapat ditulis sebagai berikut:

```

while ( $j > 0 \wedge k > 0$ ) do
     $j := j - 1$ ;  $k := k - 1$ 
if ( $j = 0$ ) return 1 else return 0

```

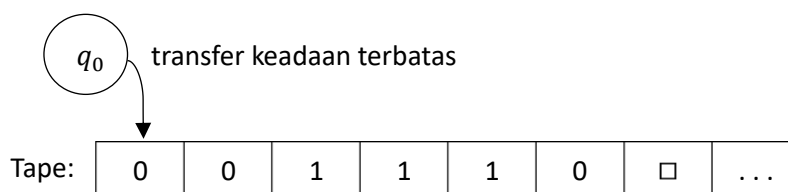
Berdasarkan algoritma di atas, kita dapat menuliskan detail mesin Turing yang sedang dirancang.

Mesin ini memiliki keadaan dan kepala pita (penunjuk pita), dapat membaca dan menulis simbol di sel yang ditunjuk oleh kepala pita, dan bergerak maju atau mundur pada pita, satu sel pada satu waktu. Awalnya, pita hanya berisi string input di bagian awal pita dan kosong di tempat lain. Gambar 11.1 mengilustrasikan mesin Turing dengan keadaan awal q_0 dan string input 001110. Jika mesin perlu menyimpan informasi, ia dapat menulis informasi ini pada pita. Untuk membaca informasi yang telah dituliskannya, mesin dapat menggerakkan kepala pita kembali ke atasnya.

Mesin terus bergerak hingga memutuskan untuk menghasilkan output. Output 1 (atau "terima") dan 0 (atau "tolak") diperoleh dengan memasuki keadaan penerimaan dan penolakan yang ditentukan. Jika tidak memasuki keadaan menerima atau menolak, mesin akan terus berjalan selamanya, tidak pernah berhenti, kecuali jika pergerakannya diblokir karena tidak ada definisi (kasus ini setara dengan "menolak").

Untuk menghasilkan output 1, mesin Turing menggunakan keadaan q_a (atau q_{accept}); untuk menghasilkan output 0, mesin menggunakan keadaan q_r (atau q_{reject}). Karena setiap pergerakan ditentukan oleh simbol yang ditunjuk oleh kepala dan keadaan saat ini, kita secara formal mendefinisikan pergerakan sebagai fungsi umum δ yang mengambil keadaan dan simbol pita sebagai input dan menghasilkan (i) keadaan berikutnya, (ii) simbol yang akan ditulis ke dalam sel saat ini, dan (iii) arah pergerakan kepala pita. Definisi δ menentukan fitur mesin Turing. Jika pengguna (yaitu, perancang mesin Turing) tidak memberikan definisi δ untuk suatu keadaan (selain q_a) dan simbol pita, maka dianggap sebagai "menolak" (dengan memblokir pergerakan berikutnya), yang setara dengan memasuki q_r .

Berikut adalah definisi formal dari mesin Turing.



Gambar 11.1 Ilustrasi mesin Turing dengan keadaan awal q_0 dan string masukan 001110

Definisi 11.2.2 Mesin Turing adalah 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$, di mana Q adalah himpunan keadaan yang terbatas, Σ adalah alfabet string masukan yang

terbatas, Γ adalah himpunan simbol pita yang terbatas, $\Sigma \subset \Gamma$, $\sqcup \in \Gamma - \Sigma$, keadaan awal $q_0 \in Q$, keadaan penerimaan $q_a \in Q$, dan fungsi parsial δ .

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

mendefinisikan setiap gerakan M , di mana L dan R menunjukkan pergerakan kepala pita ke kiri atau ke kanan, masing-masing. Jika kepala pita menunjuk ke simbol pertama pada pita, ia tidak dapat bergerak ke kiri (yaitu, "bergerak keluar dari pita" diblokir).

Contoh 11.2.3 Misalkan $A_0 = \{0^i 1^j \mid i, j > 0\}$. Kita ingin membangun $M_0 = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$ untuk menerima string di A_0 , di mana $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \sqcup\}$, dan $Q = \{q_0, q_1, q_2, q_a\}$. Tugas-tugas yang telah dilakukan sebelum memasuki suatu keadaan atau yang akan dilakukan dalam keadaan ini adalah sebagai berikut:

- q_0 : periksa apakah input dimulai dengan 0.
- q_1 : jika 0 pertama ditemukan, lewati angka 0 lainnya, dan cari angka 1 pertama.
- q_2 : jika 1 pertama ditemukan, lewati angka 1 lainnya, dan cari $\sqcup$, akhir dari input.
- q_a : laporkan "terima."

Fungsi δ didefinisikan sebagai berikut:

1. $\delta(q_0, 0) = (q_1, 0, R)$ // $i > 0$, pergi ke q_1
2. $\delta(q_1, 0) = (q_1, 0, R)$ // lewati 0 dan lanjutkan
3. $\delta(q_1, 1) = (q_2, 1, R)$ // $j > 0$, pergi ke q_2
4. $\delta(q_2, 1) = (q_2, 1, R)$ // lewati 1 dan lanjutkan
5. $\delta(q_2, \sqcup) = (q_a, \sqcup, R)$ // mencapai akhir dan menerima

M_0 memindai pita sekali dan tidak mengubah isi pita. Setelah memasuki q_a , input harus berisi string berbentuk $0^i 1^j$, di mana $i > 0$ dan $j > 0$. Untuk string input lainnya, M_0 berhenti pada titik di mana $\delta(q, a)$ tidak terdefinisi, dan hasilnya adalah "tolak". Alternatifnya, kita dapat memasuki q_r , keadaan penolakan yang ditentukan, ketika salah satu dari kasus yang tidak terdefinisi ini terjadi.

(q, a)	Alasan
$(q_0, \sqcup)$	Inputnya adalah string kosong
$(q_0, 1)$	Inputnya diawali dengan 1
$(q_1, \sqcup)$	Inputnya kehilangan 1
$(q_2, 0)$	0 muncul setelah 1

Contoh 11.2.4 Melanjutkan dari Contoh 11.2.1, berdasarkan algoritma pada Contoh 11.2.1, kita ingin membangun $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$ untuk menerima string dalam $A_1 = \{a^j b^k \mid 0 \leq j \leq k\}$, di mana $\Sigma = \{a, b\}$, $\Gamma = \{a, b, x, y, \sqcup\}$, dan

$Q = \{q_0, q_1, q_2, q_3, q_a\}$. Simbol x dan y dalam Γ masing-masing mewakili a dan b yang telah dihapus. Tugas-tugas yang telah dilakukan sebelum memasuki suatu keadaan atau yang akan dilakukan dalam keadaan ini adalah sebagai berikut:

- q_0 : di awal loop while, cari a , yaitu, periksa $j > 0$.
- q_1 : $j := j - 1$ selesai, cari b , yaitu, periksa $k > 0$.
- q_2 : $k := k - 1$ selesai, kembali ke x terakhir, dan pergi ke q_0 .
- q_3 : tidak ditemukan a , yaitu, $j = 0$, periksa $k \geq 0$ dan tidak ada a setelah b .

Fungsi δ didefinisikan sebagai berikut.

1. $\delta(q_0, \sqcup) = (q_a, \sqcup, R) // j = k = 0, \epsilon \in A_1$
2. $\delta(q_0, a) = (q_1, x, R) // j > 0, j := j - 1$, pergi ke q_1
3. $\delta(q_0, b) = (q_3, b, R) // j = 0, k > 0$, pergi ke q_3
4. $\delta(q_0, y) = (q_3, y, R) // j = 0$, lewati y , pergi ke q_3
5. $\delta(q_1, a) = (q_1, a, R) //$ lewati a , lanjutkan ke kanan
6. $\delta(q_1, y) = (q_1, y, R) //$ lewati y , lanjutkan ke kanan
7. $\delta(q_1, b) = (q_2, y, L) // k := k - 1$, mundur dan pergi ke q_2
8. $\delta(q_2, y) = (q_2, y, L) //$ lewati y , lanjutkan ke kiri
9. $\delta(q_2, a) = (q_2, a, L) //$ lewati a , lanjutkan ke kiri
10. $\delta(q_2, x) = (q_0, x, R) // x$ ditemukan, pergi ke q_0 dan mulai putaran baru
11. $\delta(q_3, y) = (q_3, y, R) //$ lewati y , lanjutkan ke kanan
12. $\delta(q_3, b) = (q_3, b, R) //$ lewati b , lanjutkan ke kanan
13. $\delta(q_3, \sqcup) = (q_a, \sqcup, L) // j = 0, k \geq 0, b^k \in A_1$

Dari instruksi di atas, kita melihat bahwa M_1 memasuki q_a ("keadaan sukses") pada baris 1 ($\epsilon \in A_1$) dan baris 13 ($j = 0, k \geq 0, b^k \in A_1$, setelah (jumlah salinan a dan b yang dihapus sama). M_1 diblokir ketika keadaannya adalah q_1 dan simbol saat ini adalah $\sqcup$ (lebih banyak simbol a daripada simbol b dalam input), atau keadaannya adalah q_3 dan simbol saat ini adalah a (a muncul setelah simbol b).

Definisi 11.2.5 Diberikan string input w ke mesin Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$, misalkan $q \in Q$ adalah keadaan saat ini, $a \in \Gamma$ adalah simbol yang ditunjuk oleh kepala pita, dan α dan β adalah string simbol pita sebelum dan sesudah kepala pita, masing-masing. Tiga serangkai $(\alpha, q, a\beta)$, atau disingkat $\alpha q a\beta$, disebut konfigurasi M . Konfigurasi awal adalah $q_0 w$, di mana w adalah string input. Sebuah langkah yang sah dari M adalah sepasang konfigurasi (C_1, C_2) , ditulis $C_1 \vdash C_2$, sedemikian sehingga $C_1 = \alpha q a\beta$, $\delta(q, a) = (p, b, R)$, dan $C_2 = \alpha b p \beta$ atau $C_1 = \alpha q a\beta$, $\delta(q, a) = (p, b, L)$, dan $C_2 = \alpha p c b \beta$.

Dalam definisi di atas, kita mengasumsikan bahwa β tidak mengandung simbol kosong setelah simbol non-kosong terakhir pada pita.

Contoh 11.2.6 Melanjutkan dari contoh sebelumnya, urutan gerakan M_1 pada string input $aabb$ adalah $q_0aabb \vdash xq_1abb \vdash aq_1bb \vdash xq_2ayb \vdash q_2xayb \vdash xq_0ayb \vdash xxq_1yb \vdash xxyq_1b \vdash xxq_2yy \vdash xq_2xyy \vdash xxq_0yy \vdash xxyq_3y \vdash xxyyq_3a \vdash xxyqay$. Jadi, $aabb$ diterima oleh M_1 .

Di sisi lain, $q_0aba \vdash xq_1ba \vdash q_2xya \vdash xq_0ya \vdash xq_3a$. Karena $\delta(q_3, a)$ tidak terdefinisi, langkah tersebut diblokir, dan input aba ditolak oleh M_1 .

Definisi 11.2.7 Misalkan $M = (Q, \Sigma, \Gamma, \delta, q_0, qa)$ adalah mesin Turing, dan $w \in \Sigma^*$, w dikatakan diterima oleh M , ditulis $M(w) = 1$, jika

$$q_0w \vdash^* \alpha q_a \beta$$

di mana $\vdash^*$ menunjukkan urutan nol atau lebih gerakan berurutan (secara formal, $\vdash^*$ adalah penutupan refleksif dan transitif dari $\vdash$), dan $\alpha, \beta \in \Gamma^*$.

Definisi 11.2.8 Bahasa yang dikenali oleh mesin Turing M adalah himpunan $L(M)$ dari semua string masukan yang diterima oleh M , secara formal

$$L(M) = \{w \mid w \in \Sigma^*, M(w) = 1\}.$$

$L \subseteq \Sigma^*$ dapat dikenali jika ada mesin Turing M sedemikian sehingga $L = L(M)$.

Mudah untuk melihat bahwa $a^2b^2 \in L(M_1)$ dan $aba \in L(M_1)$, seperti yang ditunjukkan oleh Contoh 11.2.6. Bukan tugas yang mudah untuk menunjukkan bahwa M_1 mengenali $A_1 = \{a^j b^k \mid 0 \leq j \leq k\}$, yaitu, $L(M_1) = A_1$. Untuk menunjukkan $L(M_1) = A_1$, pembaca yang tertarik dapat merujuk pada buku teks teori komputasi apa pun.

Kami ingin menunjukkan bahwa M_1 juga dapat dianggap menghitung $k - j$, karena jumlah b yang tersisa pada pita ketika M_1 menerima adalah tepat $k - j$. Inilah cara fungsi pada bilangan asli dihitung oleh mesin Turing. Dalam hal ini, mesin Turing dianggap sebagai "komputer fungsi" atau prosedur, dan fungsi yang dihitung disebut dapat dihitung. Jika fungsi yang dihitung bersifat total, maka mesin Turing yang digunakan disebut algoritma.

Definisi 11.2.9 Suatu himpunan $S = \{x \mid p(x)\}$ dikatakan dapat dihitung jika $p(x)$ dapat dihitung; S dapat diputuskan jika $p(x)$ dapat dihitung secara total.

Menurut definisi, jika L adalah bahasa formal, maka L dapat dihitung jika dan hanya jika L dapat dikenali. Misalkan $\gamma(L) = \{\gamma(w) \mid w \in L\}$, di mana γ adalah bijeksi dari string ke bilangan asli, yang didefinisikan dalam Definisi 1.3.5, maka $\gamma(L)$ dapat dihitung jika dan hanya jika L dapat dikenali.

Deskripsi Tingkat Tinggi Mesin Turing

Secara umum, deskripsi rinci mesin Turing sangat membosankan. Kita sering lebih memilih untuk memberikan deskripsi tingkat tinggi sehingga setiap baris dalam deskripsi dapat diimplementasikan oleh sejumlah langkah mesin yang terbatas.

Contoh 11.2.10 Jika kita diminta untuk merancang mesin Turing M_2 untuk mengenali $A_2 = \{b^i a^j b^k \mid i * j = k, i, j, k \geq 0\}$, kita dapat menggunakan algoritma berikut untuk mempermudah perancangan mesin Turing.

```

Check if  $i = 0$  or  $j = 0$ , then  $k$  must be 0
while ( $i > 0 \wedge k \geq j$ ) do
     $i := i - 1$ ;  $k := k - j$ 
if ( $k = 0$ ) return 1 else return 0

```

Dari Contoh 11.2.6, kita tahu cara memeriksa $i = 0$ dan cara melakukan $i := i - 1$ dan $k := k - j$. Jadi, tidak ada kesulitan teknis untuk mendesain M_2 untuk A_2 .

$M_2 = \text{"On input } w \in \{a, b\}^*$

1. Baca simbol pertama, jika itu adalah a dan input tidak mengandung b (yaitu, $i = k = 0$), atau itu adalah b dan input tidak mengandung a (yaitu, $j = k = 0$), maka, **return** 1.
2. Periksa apakah w berbentuk $b^i a^j b^k$; jika tidak, **return** 0, jika ya, kembali ke simbol pertama.
3. Kurangi i sebanyak satu dengan menghapus b pertama.
4. Lakukan zigzag antara a^j dan b^k untuk memeriksa apakah $k < j$; jika ya, **return** 0, jika tidak lakukan $k := k - j$ seperti pada M_1 .
5. Kembali ke b pertama dan kembalikan a .
6. Jika $i > 0$, lanjutkan ke 3.
7. Sekarang, $i = 0$. Periksa apakah $k = 0$; jika ya, **return** 1, jika tidak, **return** 0."

Cara mengimplementasikan setiap langkah di atas dengan serangkaian gerakan mesin Turing diserahkan sebagai latihan.

Setelah kita mengetahui cara memeriksa $i * j = k$, kita dapat memodifikasi kode mesin untuk menghitung $i * j, k/i$, atau j^k . Kita juga dapat membandingkan dua bilangan asli atau memeriksa dua string untuk melihat apakah keduanya identik atau menyalin satu string dari satu tempat ke tempat lain. Jenis operasi ini dapat diasumsikan dalam deskripsi tingkat tinggi dari mesin Turing. Dalam banyak kasus, deskripsi tingkat tinggi adalah satu-satunya cara untuk merancang mesin Turing. Namun, algoritma atau prosedur dasar, yang dapat diekspresikan oleh deskripsi tingkat tinggi, adalah suatu keharusan dalam desain beberapa mesin Turing.

Dapat Dikenali Versus Dapat Diputuskan

Diberikan mesin Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$, jika kita memasukkan $w \in \Sigma^*$ ke M , akan ada tiga kemungkinan hasil: (1) M berhenti dengan "terima" ($M(w) = 1$); (2) M berhenti

dengan "tolak" ($M(w) = 0$); dan (3) M berulang pada w ($M(w)$ tidak terdefinisi). Dengan kata lain, M membagi Σ^* menjadi tiga himpunan yang terpisah:

- $L(M) = \{w \mid w \in \Sigma^*, M(w) = 1\}$.
- $reject(M) = \{w \mid w \in \Sigma^*, M(w) = 0\}$.
- $loop(M) = \{w \mid w \in \Sigma^*, M(w) \text{ tidak terdefinisi}\}$.

Jelas, $L(M) \cup reject(M) \cup loop(M) = \Sigma^*$. Berdasarkan konvensi, $\overline{L(M)} = \Sigma^* - L(M) = reject(M) \cup loop(M)$.

Dalam teori komputabilitas, setiap $A \subseteq \Sigma^*$ dikatakan sebagai bahasa formal. A dapat berupa $\emptyset$ atau Σ^* . Mesin Turing dapat digunakan untuk mengklasifikasikan bahasa formal.

Definisi 11.2.11 (Dapat Dikenali dan Dapat Diputuskan) Suatu bahasa formal $A \subseteq \Sigma^*$ dapat dikenali jika terdapat mesin Turing M sedemikian sehingga $A = L(M)$ dan M dikatakan sebagai pengenalan L ; jika tidak, A tidak dapat dikenali.

Suatu bahasa formal dapat diputuskan jika terdapat mesin Turing M sedemikian sehingga $A = L(M)$ dan $loop(M) = \emptyset$. Dalam hal ini, M dikatakan sebagai penentu dari A .

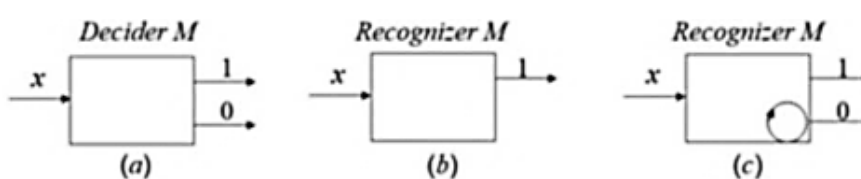
Jika A dapat diputuskan dan $L(M) = A$ untuk beberapa mesin Turing M , itu tidak berarti bahwa $loop(M) = \emptyset$. Definisi tersebut hanya memastikan bahwa terdapat penentu M' sedemikian sehingga $L = L(M')$ dan $loop(M') = \emptyset$.

Jadi, semua bahasa formal dibagi menjadi tiga kelompok yang terpisah: (a) dapat diputuskan, (b) dapat dikenali tetapi tidak dapat diputuskan, dan (c) tidak dapat dikenali. Beberapa masalah diketahui dalam salah satu dari tiga kelompok; beberapa masalah tidak diketahui termasuk dalam kelompok mana, dan ini disebut "masalah terbuka."

Dalam literatur, bahasa yang dapat dikenali juga disebut dapat dihitung secara rekursif, dapat diputuskan sebagian, dapat diputuskan sebagian, dapat dihitung sebagian, atau dapat dihitung. Bahasa yang dapat diputuskan juga disebut rekursif, dapat dikenali secara total, atau dapat dihitung secara total.

Untuk setiap bahasa formal A , terdapat fungsi unik $f_A : \Sigma^* \rightarrow \{0, 1\}$: $f_A(w) = 1$ jika dan hanya jika $w \in A$. f_A adalah fungsi karakteristik dari A . Karena terdapat bijeksi antara N dan Σ^* (Proposisi 1.3.4), f_A dapat dipandang sebagai fungsi keputusan.

Pengambil keputusan selalu berhenti pada setiap input dengan output 0 atau 1, yang sesuai dengan fungsi keputusan total (Gambar 11.2a). Pengenal berhenti dengan output 1 dan dapat berulang ketika output bukan 1, yang sesuai dengan fungsi keputusan parsial. Jika kita ingin membedakan antara "menolak" dan "berulang," kita menggunakan Gambar 11.2c sebagai pengganti (b).



Gambar 11.2 Ilustrasi pengambil keputusan dan pengenalan

11.3 KETERPUTUSAN MASALAH

Dalam teori komputabilitas, suatu masalah biasanya merupakan masalah keputusan yang memiliki jawaban ya/tidak yang sederhana. Masalah optimasi seperti “menemukan siklus sederhana terpanjang dalam graf G ” biasanya diubah menjadi pertanyaan seperti “apakah graf G memiliki siklus sederhana dengan panjang minimal m ?” dengan memperkenalkan angka m , sehingga menjadi masalah keputusan. Biasanya, masalah optimasi dan masalah keputusannya termasuk dalam kelas kompleksitas yang sama.

Pengkodean Masalah Keputusan

Praktik umum dalam teori komputabilitas adalah mengkodekan setiap masalah keputusan dengan sekumpulan string. Misalnya, masalah “apakah G memiliki siklus sederhana dengan panjang minimal m ?” dapat dikodekan sebagai himpunan B_1 :

$$B_1 = \{(G, m) \mid \text{graf } G \text{ memiliki siklus sederhana dengan panjang minimal } m\},$$

di mana (G, m) adalah string yang mewakili G dan m ; kita dapat menganggap $(G, m) = (G)(m)$. Jika G berisi n simpul dan $m \leq n$, maka alfabet untuk (G, m) dapat berupa $\{1, 2, \dots, n, (,), \cdot\}$, di mana setiap simpul diwakili oleh angka i , $1 \leq i \leq n$ dan setiap sisi diwakili oleh $(i \cdot j)$, $1 \leq i, j \leq n$. Alfabet juga dapat berupa $\{0, 1\}$ jika setiap simbol dalam $\{1, 2, \dots, n, (,), \cdot\}$ dikodekan oleh string biner dengan panjang yang sama, seperti kita menggunakan kode ASCII di komputer. Sekarang, sebuah graf G memiliki siklus sederhana dengan panjang minimal m jika dan hanya jika $(G, m) \in B_1$. B_1 menangkap esensi dari masalah keputusan ini.

“Apakah ada algoritma A yang menemukan siklus sederhana di G dengan panjang minimal m ?” Ini juga merupakan masalah keputusan dan dapat dikodekan sebagai $B_2 = \{(A, G, m) \mid \text{Algoritma } A \text{ menemukan dalam } G \text{ sebuah siklus sederhana dengan panjang minimal } m\}$ di mana (A) dapat berupa kode untuk mengimplementasikan A dalam bahasa pemrograman.

Mesin Turing juga dapat dikodekan sebagai string. Misalnya, definisi M_1 dalam Contoh 11.2.1 membutuhkan kurang dari 20 baris buku ini, dan setiap baris memiliki kurang dari 100 karakter, sehingga string 2000 karakter akan lebih dari cukup untuk mengkodekan M_1 . Kita juga dapat melihat pengkodean mesin Turing apa pun sebagai string biner karena setiap file disimpan dalam kode biner di komputer. Singkatnya, ada banyak cara untuk mengkodekan mesin Turing atau input apa pun yang diterima oleh mesin Turing. Metode pengkodean apa pun akan berfungsi bagi kita selama asumsi berikut benar:

- Setiap pengkodean memiliki alfabet yang sama. Secara teori, alfabet biner sudah cukup untuk semuanya.
- Objek yang berbeda memiliki kode yang berbeda. Artinya, pengkodean adalah fungsi injektif dari objek ke string.
- Komputasi pengkodean dan dekode efisien, membutuhkan waktu linier atau waktu kuadratik dalam kasus terburuk. Dekode dapat memberi tahu apakah mesin Turing yang dikodekan baik (terdefinisi dengan baik) atau tidak.

Ada banyak masalah keputusan tentang mesin Turing (TM) dan berikut adalah beberapa di antaranya:

- $A_{TM} = \{(M, w) \mid \text{TM } M \text{ menerima input } w\}$
- $H_{TM} = \{(M, w) \mid \text{TM } M \text{ berhenti pada input } w\}$
- $D_{TM} = \{(M) \mid \text{TM } M \text{ berhenti pada setiap input}\}$
- $E_{TM} = \{(M) \mid L(M) = \emptyset \text{ untuk TM } M\}$
- $One_{TM} = \{(M) \mid M \text{ adalah TM, } |L(M)| = 1\}$
- $Fin_{TM} = \{(M) \mid L(M) \text{ terbatas untuk TM } M\}$

Bahasa A_{TM} mengkodekan masalah penerimaan mesin Turing: diberikan mesin Turing M dan input w apa pun, apakah M menerima w ? Ini adalah salah satu masalah pertama mesin Turing yang terbukti tidak dapat diputuskan, yaitu, tidak ada algoritma untuk menyelesaikan masalah penerimaan.

Bahasa H_{TM} mengkodekan masalah penghentian mesin Turing yang terkenal. Alan Turing membuktikan pada tahun 1936 bahwa algoritma umum untuk menyelesaikan masalah penghentian untuk semua pasangan (M, w) yang mungkin tidak dapat ada. Kita akan menunjukkan bahwa A_{TM} dan H_{TM} dapat dikenali tetapi tidak dapat diputuskan.

Bahasa D_{TM} adalah pengkodean semua mesin Turing yang berhenti pada setiap input. Mesin Turing ini disebut penentu atau algoritma. D_{TM} mengkodekan masalah yang disebut masalah penghentian universal, yang lebih sulit daripada masalah yang dikodekan oleh H_{TM} .

Bahasa E_{TM} mengkodekan masalah kekosongan mesin Turing. Tidak seperti A_{TM} dan H_{TM} , E_{TM} tidak dapat dikenali.

One_{TM} (atau **Fin_{TM}**) mengkodekan masalah yang kurang populer: mesin Turing yang menerima tepat satu string (atau sejumlah string terbatas). Anda dapat membuat bahasa Anda sendiri untuk masalah pengambilan keputusan apa pun.

Untuk kemudahan diskusi, kita mengasumsikan lebih lanjut bahwa (M) dapat berupa string apa pun, termasuk ϵ (string kosong). Beberapa string adalah kode dari mesin Turing "baik" (yaitu, terdefinisi dengan baik); beberapa adalah kode dari mesin Turing "buruk", yaitu, yang sedang dalam pembangunan atau dengan karakter yang kacau. Kita mengasumsikan bahwa mesin Turing buruk memiliki nol keadaan, bergerak nol langkah pada input apa pun, dan tidak menerima apa pun, sama seperti program "buruk" yang tidak dapat dieksekusi.

Untuk memahami $x = (M, w)$, kita dapat mengasumsikan bahwa $x = (M)\#w$ jika M terdefinisi dengan baik, di mana $\#$ adalah karakter khusus yang tidak digunakan oleh M dan w . Jika x berisi lebih dari satu salinan $\#$, maka (M) adalah kode dari mesin Turing buruk dan w adalah sufiks setelah $\#$ terakhir. Jika x tidak mengandung $\#$, kita mengasumsikan $(M) = \epsilon$ dan $x = w$.

Berdasarkan asumsi di atas, (M) atau (M, w) dapat berupa string apa pun dari Σ^* . Sekarang, komplemen dari bahasa apa pun dalam daftar di atas masuk akal. Misalkan $\bar{L} = \Sigma^* - L$:

$$\begin{aligned}\overline{A_{TM}} &= \{(M, w) \mid \text{TM } M \text{ tidak menerima input } w\} \\ \overline{H_{TM}} &= \{(M, w) \mid \text{TM } M \text{ tidak berhenti pada input } w\}\end{aligned}$$

$$\begin{aligned}\overline{D_{TM}} &= \{(M) \mid \text{TM } M \text{ berulang pada beberapa input}\} \\ \overline{E_{TM}} &= \{(M) \mid L(M) \neq \emptyset \text{ untuk TM } M\} \\ &\dots\end{aligned}$$

Berdasarkan asumsi tersebut, kode mesin Turing yang buruk tidak muncul di A_{TM} , $\overline{H_{TM}}$, $\overline{E_{TM}}$, dan $\overline{D_{TM}}$.

Masalah yang Dapat Diputuskan

Karena setiap masalah keputusan dapat dikodekan sebagai bahasa formal $L \subseteq \Sigma^*$, kita ingin merancang mesin Turing M yang dapat menjawab pertanyaan apakah $w \in L$ atau tidak, untuk setiap $w \in \Sigma^*$. Hal ini hanya mungkin terjadi jika L dapat diputuskan; jika L dapat dikenali tetapi tidak dapat diputuskan, kita hanya dapat menjawab pertanyaan tersebut ketika M berhenti pada w ; jika L tidak dapat dikenali, maka tidak ada mesin Turing seperti itu yang dapat ada.

Definisi 11.3.1 (Masalah yang Dapat Diputuskan dan Dapat Dihitung) Suatu masalah keputusan dikatakan dapat diputuskan jika bahasa pengkodeannya dapat diputuskan. Suatu masalah keputusan dikatakan dapat dihitung jika pengkodeannya dapat dikenali.

Definisi formal di atas konsisten dengan definisi yang diberikan pada Bagian 1.3, di mana kita membagi semua masalah keputusan menjadi tiga kelompok yang terpisah: (a) dapat diputuskan, (b) dapat dihitung tetapi tidak dapat diputuskan, dan (c) tidak dapat dihitung. Bahasa yang dapat dikenali sesuai dengan masalah keputusan yang dapat dihitung. Tabel berikut merangkum terminologi yang digunakan untuk tiga domain yang dipelajari: bahasa formal, masalah keputusan, dan fungsi umum, beserta nama khusus mesin Turing untuk setiap domain.

Domain	Perbedaan	Nama mesin Turing
<i>Bahasa formal</i>	Dapat dipilih	Pengambil keputusan
	Dapat dikenali	Pengenali
<i>Masalah keputusan</i>	Dapat dipilih	Pengambil keputusan
	Dapat dihitung	Prosedur
<i>Fungsi umum</i>	Total dapat dihitung	Algoritma
	Dapat dihitung	Prosedur

Semua masalah keputusan yang telah kita temui dalam mata kuliah struktur data dan algoritma, di mana kompleksitasnya dibatasi oleh fungsi Big O, termasuk dalam kelas masalah yang dapat diputuskan. Dalam teori komputabilitas, penentu untuk masalah keputusan juga disebut prosedur keputusan, atau algoritma, yang bertepatan dengan definisi algoritma dalam mata kuliah struktur data dan algoritma mana pun: urutan instruksi terbatas yang dapat dieksekusi dalam sejumlah langkah terbatas untuk input apa pun.

Ingatlah bahwa fungsi keputusan (Proposisi 1.3.7) adalah fungsi yang mengambil bilangan asli sebagai input dan mengembalikan 0 atau 1 sebagai output. Setiap fungsi keputusan $f: N \rightarrow \{0, 1\}$ adalah masalah keputusan tentang bilangan asli: apakah $f(n) = 1$ untuk input $n \in N$? Tampaknya, fungsi keputusan adalah masalah keputusan sekaligus fungsi umum.

Oleh karena itu, fungsi keputusan dapat diputuskan (oleh penentu) dan dapat dihitung secara total (oleh algoritma). Perhatikan bahwa fungsi komputasi total kita disebut komputasi oleh Michael Sipser. Kami menghindari penggunaan "komputasi" untuk "komputasi total," karena tesis Turing berlaku tidak hanya untuk fungsi total tetapi juga untuk fungsi parsial.

Beberapa masalah keputusan dapat diputuskan; beberapa dapat dihitung tetapi tidak dapat diputuskan; beberapa tidak dapat dihitung. Menurut Proposisi 1.3.7, himpunan semua fungsi keputusan tidak terhingga. Artinya, ada lebih banyak fungsi keputusan daripada mesin Turing yang tersedia, yang dapat dihitung. Untuk setiap fungsi keputusan $f: N \rightarrow \{0, 1\}$, tidak selalu mungkin untuk memiliki mesin Turing M_f sedemikian sehingga M menerima i jika dan hanya jika $f(i) = 1$ untuk setiap bilangan asli i . Ada banyak fungsi keputusan yang tidak dapat dihitung oleh perangkat komputasi apa pun mengingat tesis Church-Turing.

Di sini, kita tertarik pada masalah keputusan tentang mesin Turing. Ketika mesin Turing dikodekan sebagai string atau sebagai bilangan asli, masalah keputusan ini juga merupakan fungsi keputusan.

Contoh 11.3.2 Misalkan $S10_{TM} = \{(M) \mid M \text{ memiliki setidaknya sepuluh keadaan}\}$. Maka, $S10_{TM}$ dapat diputuskan karena kita dapat merancang mesin Turing yang menghitung jumlah keadaan berbeda dalam (M) . Penghitungan tidak memakan waktu lebih dari waktu linier, dan jika hitungan melebihi 10, berhenti dengan "terima"; jika tidak, berhenti dengan "tolak."

Contoh 11.3.3 Misalkan $B10_{TM} = \{(M) \mid M \text{ bergerak setidaknya sepuluh langkah pada setiap input}\}$.

$B10_{TM}$ dapat diputuskan karena kita hanya perlu mempertimbangkan string dengan panjang kurang dari atau sama dengan 10 sebagai input, dan untuk setiap input, kita perlu mensimulasikan M paling banyak sepuluh langkah. Jadi, total upaya yang dibutuhkan terbatas. Artinya, kita tidak perlu mempertimbangkan string dengan panjang lebih dari 10 karena Anda membutuhkan setidaknya sepuluh langkah untuk membaca simbol ke-11.

Misalkan M_B adalah penentu yang akan kita buat untuk $B10_{TM}$. Algoritma dasar M_B bekerja sebagai berikut: pita M_B dibagi menjadi tiga bagian yang dipisahkan oleh simbol khusus # dan \$. Bagian 1 (dari sel pertama hingga #) menyimpan string input (M) , di mana $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$, dan tidak akan pernah berubah. Jika (M) adalah kode mesin Turing yang buruk, berhenti dengan "tolak."

Bagian 2 (antara # dan \$) memiliki 11 sel dan menyimpan secara berurutan string input $w \in \Sigma^*$ dari M untuk setiap kemungkinan w sedemikian sehingga $|w| \leq 10$. Kita

mengasumsikan bahwa string dalam Σ^* dihasilkan oleh urutan kanonik (Definisi 1.3.5). Jadi, jika $\Sigma = \{0, 1\}$, maka string berada dalam urutan ini: $\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots$. Dengan asumsi urutan ini, kita mengatakan *penerus* dari ϵ adalah 0, *penerus* dari 0 adalah 1, dan seterusnya.

Bagian 3 (setelah \$) akan menjadi ruang untuk mensimulasikan pergerakan M .

M_B = "Pada input (M), M adalah mesin Turing.

1. Pindai input; jika (M) adalah kode yang salah, **return** 0.
Write# di akhir (M); pindah ke kanan 11 langkah dan write\$.
Perlakukan string di bagian 2 (antara # dan \$, tidak termasuk simbol $\sqcup$) sebagai w .
Awalnya, $w = \epsilon$, string kosong.
2. Pindah ke awal bagian 3 dan tulis q_0 dari M setelah \$.
3. Salin string w di bagian 2 ke bagian 3, setelah q_0 .
Sekarang, bagian 3 berisi q_0w , konfigurasi awal M .
4. Berzigzag antara bagian 1 dan bagian 3, simulasikan M pada w :
 - 4.1. Gunakan keadaan M_B untuk mengingat pasangan (q, a) di bagian 3.
 - 4.2. Temukan definisi delta untuk (q, a) di bagian 1.
 - 4.3. Jika tidak ada $\delta(q, a)$ ditemukan, **return** 0. // M belum bergerak sepuluh kali.
 - 4.4. Gunakan keadaan M_B untuk mengingat $\delta(q, a) = (p, b, D), D \in \{L, R\}$.
 - 4.5. Bergerak ke kanan untuk menemukan keadaan M di bagian 3 dan perbarui konfigurasi M sesuai dengan $\delta(q, a) = (p, b, D)$.
 - 4.6. Jika 4.5 telah dilakukan kurang dari sepuluh kali, lanjutkan ke 4.1.
5. Hapus isi bagian 3, pindah ke bagian 2, dan ganti w dengan penerusnya.
6. Jika w melebihi 10 simbol, kembalikan 1, jika tidak, lanjutkan ke 2.

Baris 2–3 menulis konfigurasi awal q_0w dari M di bagian 3, mempersiapkan simulasi M pada w . Baris 4.1–4.6 mensimulasikan pergerakan M selama sepuluh langkah. Jika ini memungkinkan, M_B melanjutkan ke baris 5 untuk w baru: M_B menghapus konfigurasi M saat ini di bagian 3 dan mengganti w dengan penerus w di bagian 2 (baris 5). Jika pergerakan M terblokir sebelum bergerak 10 langkah, M_B menolak (baris 4.3). Pada baris 6, M_B memeriksa apakah $|w| > 10$. Jika ya, itu berarti M telah menghabiskan semua string $x, |x| \leq 10$, dan belum menemukan satu pun yang tidak dapat dilalui M setidaknya sepuluh langkah. Jadi, M_B berhenti dengan "terima"; jika tidak, ia mulai mensimulasikan M pada w baru dengan pergi ke baris 2 (baris 6).

Kami ingin menunjukkan bahwa baris 4.1–4.5 dari contoh di atas menjelaskan bagaimana pergerakan M disimulasikan oleh M_B . Nantinya, ketika kita mengatakan " M' mensimulasikan M ," kita merujuk pada contoh ini untuk detail simulasinya. Anda tidak dapat menggunakan status mesin Turing apa pun untuk mengingat informasi seperti a^i , di mana i adalah angka apa pun. Namun, kita dapat menggunakan keadaan M_B untuk mengingat fungsi δ dari M karena setiap $\delta(q, a) = (p, b, D)$ hanya berisi lima simbol dan hanya ada sejumlah simbol yang terbatas (baik himpunan keadaan maupun himpunan simbol pita terbatas).

Masalah yang Tidak Dapat Diputuskan

Sesuai dengan teorema ketidaklengkapan pertama Gödel, teorema berikut memberikan bukti formal pertama dari masalah yang tidak dapat diputuskan dalam buku ini.

Teorema 11.3.4 Masalah penerimaan mesin Turing, A_{TM} , tidak dapat diputuskan.

Bukti: Asumsikan bahwa A_{TM} dapat diputuskan, maka ada penentu hipotetis H untuk A_{TM} . H menerima (M, w) jika dan hanya jika M menerima w . Secara khusus, H menerima $(M, (M))$ jika dan hanya jika M menerima (M) .

Menggunakan H , kita membangun penentu lain Di , yang memeriksa apakah mesin Turing M tidak menerima kodenya sendiri:

$Di =$ "Pada input (M) , M adalah mesin Turing.
 1. Simulasikan H pada $(M, (M))$ sampai H berhenti.
 2. $return\ 1 - H((M, (M)))$."

$Di((M)) = 1 - H((M, (M)))$ berarti $Di((M)) = \neg H((M, (M)))$ jika kita menganggap 1 sebagai benar dan 0 sebagai salah. Artinya, jika H menerima $(M, (M))$, Di menolak (M) ; jika H menolak $(M, (M))$, Di menerima (M) . Oleh karena itu, Di menerima (M) jika dan hanya jika H menolak $(M, (M))$, jika dan hanya jika M menolak (M) , jika dan hanya jika $M((M)) = 0$. Oleh karena itu, Di adalah penentu untuk

$$X = \{(M) \mid M \text{ adalah mesin Turing, } M((M)) = 0\}$$

yang dapat diputuskan, karena Di berhenti pada input apa pun.

Mari kita periksa apakah $(Di) \in X$ dan hanya ada dua kasus:

- $(Di) \in X$: Setiap anggota (M) dari X , termasuk (Di) , memenuhi $M((M)) = 0$. Mengganti M dengan Di , kita memiliki $Di((Di)) = 0$, yaitu, Di menolak (Di) . Di sisi lain, dari $Di((M)) = 1 - H((M, (M)))$, kita memiliki $H((Di, (Di))) = 1 - Di((Di)) = 1 - 0 = 1$, yaitu, Di menerima (Di) , sebuah kontradiksi.
- $(Di) \notin X$: Itu berarti $Di((Di)) = 1$ menurut definisi X ; oleh karena itu, $H((Di, (Di))) = 1$. Dari $Di((M)) = 1 - H((M, (M)))$, kita memiliki $Di((Di)) = 1 - H((Di, (Di))) = 1 - 1 = 0$, sebuah kontradiksi.

Dalam kedua kasus, kita memiliki kontradiksi, jadi Di tidak dapat ada. Di dibangun dari H ; oleh karena itu, H tidak dapat ada. H berasal dari asumsi bahwa A_{TM} dapat diputuskan. Dengan demikian, A_{TM} tidak dapat diputuskan.

Bukti di atas analog dengan "paradoks Barber" di Bagian 1.3. Mesin Turing Di , yang menerima kode dari mesin Turing yang tidak menerima pengkodean mereka sendiri, sesuai dengan "tukang cukur yang mencukur semua orang yang tidak mencukur diri mereka sendiri." Kedua bukti tersebut berasal dari metode diagonal Cantor, yang menggunakan matriks. Dalam

pembuktian kami, baris matriks diindeks oleh (M_i) , kolom diindeks oleh w_i , dan baik (M_i) maupun w_i adalah anggota Σ^* . Berdasarkan asumsi kami, $w_i = (M_i)$. Pembuktian di atas analog dengan pembuktian teorema ketidaklengkapan pertama Gödel karena pembuktian Gödel juga didasarkan pada metode diagonal Cantor.

Reduksi

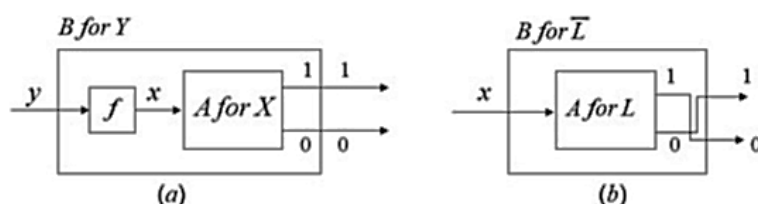
Setelah kita menemukan satu bahasa yang tidak dapat diputuskan, bahasa-bahasa yang tidak dapat diputuskan lainnya dapat ditemukan melalui reduksi.

Definisi 11.3.5 Misalkan X dan Y adalah dua bahasa formal. Kita katakan Y direduksi menjadi X jika kita dapat membangun penentu atau pengenali B untuk Y , menggunakan penentu atau pengenali A untuk X sebagai komponen B , sedemikian sehingga salah satu dari berikut ini benar:

- Jika X dapat diputuskan, maka Y dapat diputuskan.
- Jika X dapat dikenali, maka Y dapat dikenali.

Konsep reduksi dapat digeneralisasikan ke masalah komputasi X dan Y : ketika kita ingin membangun algoritma (atau prosedur) B untuk Y , kita dapat menggunakan algoritma (atau prosedur) A untuk X sebagai komponen B . Secara intuitif, reduksi menyerupai gagasan "panggilan prosedur" dalam pemrograman komputer. Sebagai programmer berpengalaman, seseorang telah melihat banyak panggilan prosedur dalam pemrograman berorientasi objek sebagai cara pemecahan masalah. Artinya, jika kita memiliki metode A untuk masalah X , untuk menyelesaikan masalah Y , kita dapat membangun metode B dari A untuk Y . Misalnya, misalkan X adalah "mengukur diameter lingkaran" dan Y adalah "mengukur luas lingkaran," maka metode A dari X dapat digunakan untuk membangun metode B dari Y . Itulah tepatnya arti dari mereduksi Y ke X .

Misalkan kita memiliki penentu (pengenali) A untuk menjawab pertanyaan apakah $x \in X$ atau tidak. Dengan menggunakan A , kita kemudian membangun penentu (pengenali) B untuk menjawab pertanyaan $y \in Y$ untuk setiap y . Di dalam B , kita dapat membuat input x dan memanggil A pada x . Output $A(x)$ digunakan untuk membuat output B . Jika penentu (pengenali) B bekerja dengan benar untuk Y , yaitu, $B(y)$ mengembalikan nilai benar jika dan hanya jika $y \in Y$, maka Y berhasil direduksi menjadi X .



Gambar 11.3 Ilustrasi dua kasus khusus reduksi: (a) reduksi pemetaan; (b) negasi

Ada dua kasus khusus reduksi. Pada kasus pertama, penentu (pengenali) B dengan input y menggunakan algoritma f yang menerima input y dan menghasilkan $x = f(y)$. B

memanggil A dengan input x dan mengembalikan $A(x)$ sebagai output dari $B(y)$, seperti yang diilustrasikan pada Gambar 11.3a. Dalam kasus ini, $y \in Y$ jika dan hanya jika $x = f(y) \in X$, dan reduksi ini disebut reduksi pemetaan, karena f memetakan instance "ya" dari Y ke instance "ya" dari X dan instance "tidak" dari Y ke instance "tidak" dari X .

Proposisi 11.3.6 Jika Y direduksi menjadi X dengan reduksi pemetaan, maka $\bar{Y}$ direduksi menjadi $\bar{X}$.

Bukti Algoritma f yang sama dapat digunakan untuk reduksi dari Y ke X .

Dalam kasus kedua reduksi khusus, input yang sama digunakan oleh A dan B . Misalnya, untuk menunjukkan bahwa bahasa yang dapat diputuskan tertutup terhadap komplement, jika kita memiliki penentu A untuk bahasa formal L , kita dapat membangun B untuk $\bar{L} = \Sigma^* - L$ sedemikian sehingga $B(x) = \neg A(x)$, seperti yang diilustrasikan pada Gambar 11.3b.

Penerapan reduksi dalam bab ini berbeda dan dapat dinyatakan sebagai berikut: jika Y direduksi ke X dengan membangun B dari A dan kita tahu B tidak ada untuk Y , maka A tidak dapat ada untuk X . Secara formal, kita memiliki proposisi berikut, yang berasal langsung dari definisi reduksi.

Proposisi 11.3.7 Jika Y direduksi ke X dan Y tidak dapat diputuskan, maka X tidak dapat diputuskan.

Dalam bukti berikut tentang masalah penghentian mesin Turing, X adalah pengkodean dari masalah penghentian dan Y adalah A_{TM} , bahasa tak terputuskan pertama yang diketahui dalam buku ini.

Teorema 11.3.8 Masalah penghentian mesin Turing, H_{TM} , tidak dapat diputuskan.

Bukti Asumsikan H_{TM} dapat diputuskan, maka ada penentu hipotetis H untuk H_{TM} . Karena H adalah penentu, H akan berhenti pada input apa pun, dan H menerima (M, w) jika dan hanya jika M berhenti pada w .

Dengan menggunakan H , kita dapat membangun penentu lain S untuk ATM sebagai berikut: $S =$ "Pada input (M, w) , M adalah mesin Turing dan w adalah input ke M .

1. Simulasikan H pada (M, w) sampai H berhenti.
2. Jika H menolak, **return** 0. // H menolak (M, w) jika M berulang pada w .
3. Simulasikan M pada w sampai M berhenti. // M akan berhenti karena H menerima (M, w) .
4. **return** $M(w)$."

Pada baris 2, ketika H menolak (M, w) , itu berarti M berulang pada w dan membenarkan penolakan w oleh S . Pada baris 3, kita tahu M akan berhenti pada w . Ketika M berhenti pada w , S mengembalikan output $M(w)$ pada baris 4. Keempat baris S akan selesai dalam jumlah langkah yang terbatas, sehingga S adalah penentu.

Hubungan antara M dan S diberikan di bawah ini:

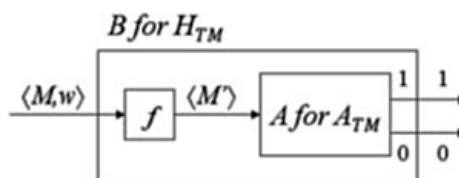
1. M menerima w : S menerima (M, w) (baris 4).
2. M menolak w : S menolak (M, w) (baris 4).
3. M berulang pada w : S menolak (M, w) (baris 2).

Dengan demikian, M menerima w jika dan hanya jika S menerima (M, w) . Jika H dapat memutuskan apakah M berhenti pada w , maka S dapat memutuskan apakah M menerima w . Dengan demikian, S adalah penentu untuk A_{TM} . Karena S tidak dapat ada, H juga tidak dapat ada. Dengan demikian, H_{TM} harus tidak dapat diputuskan.

Bukti di atas adalah reduksi dari kasus khusus kedua, di mana string input ke S dan H sama.

Reduksi adalah alat yang ampuh untuk menunjukkan bahwa suatu masalah X tidak dapat diputuskan: anggap X dapat diputuskan dan A adalah penentu untuk X . Kita menggunakan A untuk membangun penentu lain B untuk masalah Y yang tidak dapat diputuskan yang diketahui. Karena B tidak dapat eksis, maka A tidak dapat eksis, dan dengan demikian X tidak dapat diputuskan. Dalam pembuktian teorema di atas, X adalah H_{TM} , A adalah H , Y adalah A_{TM} , dan B adalah S . Kita menunjukkan bahwa jika ada penentu untuk H_{TM} , maka ada penentu untuk A_{TM} , yang merupakan kontradiksi.

Kita juga dapat menukar peran H_{TM} dan A_{TM} dengan menunjukkan hasil berikut.



Gambar 11.4 Ilustrasi pengurangan dari ATM ke HTM

Proposisi 11.3.9 Jika ada penentu untuk A_{TM} , maka ada penentu untuk H_{TM} .

Bukti Misalkan A_{TM} dapat diputuskan dan A adalah penentunya. Kita membangun penentu B untuk H_{TM} sebagai berikut dan diilustrasikan pada Gambar 11.4.

B = "Pada input (M, w) , M adalah mesin Turing dan w adalah input ke M .

1. Panggil algoritma f pada (M) : f menciptakan (M') sebagai berikut:
Untuk setiap keadaan q dan simbol pita x dari M , jika q adalah q_r (keadaan penolakan), ganti q_r dengan q_a (keadaan penerimaan); jika q bukan q_a dan $\delta(q, x)$ tidak didefinisikan, tambahkan ke M' : $\delta'(q, x) = (q_a, x, R)$.
2. Simulasikan A pada (M', w) sampai A berhenti.
3. **return** $A((M', w))$."

Hubungan antara M dan M' diberikan di bawah ini:

1. M menerima w : M' menerima w .
2. M menolak w : M' menerima w .
3. M berulang pada w : M' berulang pada w .

Dengan demikian, M berhenti pada w jika dan hanya jika M' menerima w . $B((M, w)) = A((M', w)) = 1$ jika dan hanya jika M menerima w dan M berhenti pada w . Ini membenarkan keluaran B pada baris 3 dari B . Dengan kata lain, B menerima (M, w) jika dan hanya jika M berhenti pada w ; A menerima (M', w) jika dan hanya jika M' menerima w . Karena ketiga baris B dapat dilakukan dalam sejumlah langkah yang terbatas, B adalah penentu untuk H_{TM} .

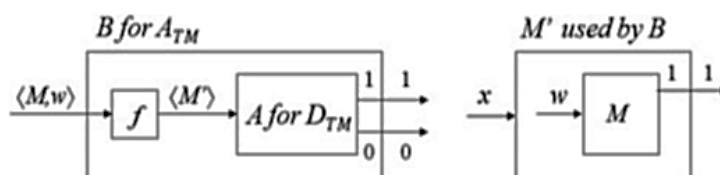
Bukti di atas adalah contoh reduksi pemetaan karena kita telah menunjukkan bahwa $(M', w) \in A_{TM}$ jika dan hanya jika $(M, w) \in H_{TM}$. Karena A_{TM} dan H_{TM} direduksi satu sama lain, keduanya disebut ekuivalen Turing, dan kelas semua bahasa (atau masalah keputusan) ekuivalen Turing disebut derajat Turing, suatu topik di luar cakupan buku ini.

Proposisi 11.3.10 $D_{TM} = \{(M) \mid TM \text{ } M \text{ adalah penentu} \}$ tidak dapat diputuskan.

Bukti Misalkan D_{TM} dapat diputuskan dan A adalah penentunya. Kita membangun penentu B untuk A_{TM} sebagai berikut dan diilustrasikan pada Gambar 11.5

$B =$ "Pada input (M, w) , M adalah mesin Turing dan w adalah input ke M .

1. Hasilkan kode mesin Turing M' (lihat di bawah), (M') , dari (M, w) .
2. **return** $A((M'))$."



Gambar 11.5 Ilustrasi reduksi dari ATM ke DTM

$M' =$ "Pada input x

1. Ganti x dengan w sebagai input baru.
2. Simulasikan M pada w hingga M berhenti.
3. Jika $M(w) = 0$, maka masuk ke keadaan perulangan.
4. Jika tidak, **return** 1."

Perhatikan bahwa B hanya membuat kode M' dan tidak mensimulasikan M pada w . B menggunakan kode M untuk membuat (M') : M' akan mengganti input x -nya dengan w pada pita M' dan kemudian mensimulasikan M pada w . Jika M menolak w , M' masuk ke dalam perulangan tak terbatas. Jika M menerima w , M' menerima x . Hubungan antara M dan M' diberikan sebagai berikut:

1. M menerima w : M' berhenti pada setiap x .
2. M tidak menerima w : M' berulang pada setiap x .

Dengan demikian, M menerima w jika dan hanya jika M' berhenti pada setiap x . $B((M, w)) = A((M')) = 1$ jika dan hanya jika $(M') \in DTM$ dan $(M, w) \in ATM$. Oleh karena itu, B adalah penentu untuk A_{TM} jika A adalah penentu untuk D_{TM} . Karena A_{TM} tidak memiliki penentu, A tidak dapat ada dan D_{TM} harus tidak dapat diputuskan.

Kita tahu bahwa A_{TM} dan H_{TM} sama-sama tidak dapat diputuskan; apakah keduanya dapat dikenali? Jawabannya adalah ya untuk keduanya.

Proposisi 11.3.11 A_{TM} dapat dikenali.

Bukti Kami menyediakan mesin Turing berikut untuk A_{TM} .

U = "Pada input (M, w) , M adalah mesin Turing dan w adalah input ke M .

1. Simulasikan M pada w sampai M berhenti.
2. **return** $M(w)$."

Jelas bahwa U menerima (M, w) jika M menerima w ; U menolak jika M menolak w ; U berulang jika M berulang pada w . Jadi, $L(U) = A_{TM} \cdot U$ dalam bukti di atas disebut mesin Turing universal, karena mensimulasikan eksekusi mesin Turing apa pun pada input apa pun. Jika kita memandang (M, w) sebagai program tersimpan dengan data, U setara dengan mesin RASP (*random-access stored-program*). Ide ini memiliki dampak besar pada pengembangan komputer saat ini dan memiliki kepentingan teoritis yang cukup besar.

Proposisi 11.3.12 H_{TM} dapat dikenali.

Bukti H_{TM} dapat dikenali oleh mesin Turing T .

T = "Pada input (M, w) , M adalah mesin Turing dan w adalah input ke M .

1. Simulasikan M pada w sampai M berhenti.
2. kembalikan 1."

T menerima (M, w) jika dan hanya jika M berhenti pada w , sehingga $L(T) = H_{TM}$.

Proposisi 11.3.13 Jika L dan $\bar{L}$ keduanya dapat dikenali, maka L dapat diputuskan.

Bukti Misalkan A dan B adalah pengenali L dan $\bar{L}$, masing-masing. Kita dapat menggunakan A dan B untuk membuat penentu untuk L :

T = "Pada input (M, w) , M adalah mesin Turing dan w adalah input ke M .

1. Simulasikan A pada w untuk satu langkah.
2. Jika A menerima, kembali 1.
3. Simulasikan B pada w untuk satu langkah.

4. Jika B menerima, kembali 0.
5. Lanjutkan ke baris 1."

Untuk setiap $w \in \Sigma^*$, $w \in L$ atau $w \in \bar{L}$. Jadi, A menerima w atau B menerima w . T akan berhenti ketika salah satu dari mereka menerima. Jika $w \in L$, T akan menerima; jika tidak, T akan menolak. Dengan demikian, T adalah penentu untuk L .

Konstruksi T adalah contoh tipikal penggunaan pengenalan sebagai komponen untuk merancang algoritma. Anda tidak dapat menggunakan pengenalan sebagai penentu (seperti panggilan prosedur dalam pemrograman komputer) karena pengenalan tidak boleh berhenti.

Hasil di atas memberikan metode mudah untuk memeriksa apakah suatu bahasa formal tidak dapat dikenali (lihat Proposisi 11.4.13).

Teorema Rice

Kita tetap berpegang pada asumsi bahwa setiap string Σ^* adalah kode dari mesin Turing (baik atau buruk).

Definisi 11.3.14 Diberikan predikat $p(M)$ untuk semua mesin Turing M , misalkan:

$$L_p = \{(M) \mid p(M) = 1, M \text{ adalah mesin Turing}\}$$

Kita katakan p adalah properti Rice jika p memenuhi dua kondisi berikut:

1. Nontrivial: Baik $L_p = \Sigma^*$ maupun $L_p = \emptyset$ tidak terpenuhi.
2. Terkait bahasa: Untuk setiap dua mesin Turing A dan B , jika $L(A) = L(B)$ dan $(A) \in L_p$, maka $(B) \in L_p$.

Kita katakan p positif jika $|L(M)| > 0$ setiap kali $p(M) = 1$; p tidak positif jika p tidak positif.

Jelas, jika p positif dan $L(M) = \emptyset$, maka $p(M) = 0$. Jika p trivial, maka L_p dapat diputuskan karena baik Σ^* maupun $\emptyset$ dapat diputuskan. Teorema Rice menyatakan bahwa L_p tidak dapat diputuskan jika p adalah sifat Rice.

Pada awal bab ini, kita telah menyebutkan teorema ketidaklengkapan kedua Gödel: jika suatu sistem formal S konsisten, maka kita tidak dapat membuktikan di S bahwa S itu sendiri konsisten. Jika kita menganggap "konsistensi" sebagai sifat Rice dari mesin Turing, maka teorema ketidaklengkapan kedua Gödel digeneralisasikan sebagai teorema Rice, yang menyatakan bahwa tidak ada penentu untuk menunjukkan bahwa mesin Turing memiliki sifat Rice atau tidak.

Teorema 11.3.15 (Teorema Rice) Jika p adalah sifat Rice, maka L_p tidak dapat diputuskan.

Bukti Kita mengasumsikan bahwa p positif; Jika asumsi ini gagal, kita mengambil negasi dari p sebagai p dalam bukti berikut. Karena komplemen dari bahasa yang dapat diputuskan juga dapat diputuskan, asumsi kita tidak membatalkan teorema tersebut.

Karena L_p tidak trivial, ada mesin Turing A sedemikian sehingga $(A) \in L_p$. Sebagai langkah pertama pembuktian dengan kontradiksi, kita mengasumsikan bahwa L_p dapat diputuskan dan S adalah penentu hipotetisnya. Artinya, $S((M)) = 1$ jika dan hanya jika $(M) \in L_p$. Dengan menggunakan S dan A , kita membangun penentu T untuk A_{TM} sebagai berikut:

T = "Pada input (M, w) , M adalah mesin Turing dan w adalah input dari M .

1. Hasilkan (M') , kode dari M' (lihat di bawah), dari (M, w) dan (A) .
2. **return** $S((M'))$."

M' = "Pada input x

1. Letakkan simbol khusus # di akhir x dan tulis w setelah #.
2. Simulasikan M pada w sampai M berhenti, lalu hapus simbol apa pun setelah #, termasuk #.
3. Jika M menerima w , simulasikan A pada x sampai A berhenti dan **return** $A(x)$.
4. Jika M menolak w , masuk ke keadaan perulangan."

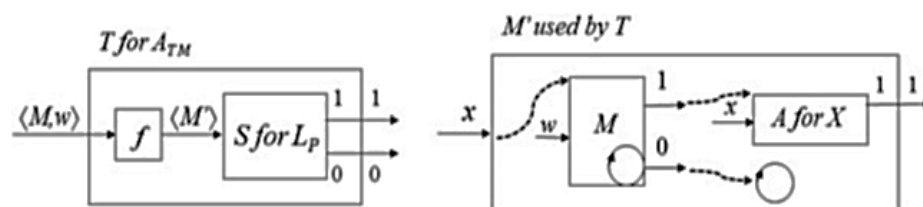
Hubungan antara M dan M' diberikan di bawah ini:

1. M menerima w : $L(M') = X = L(A)$ dan $p(M') = p(A) = 1$.
2. M menolak w atau berulang pada w : M' berulang, $L(M') = \emptyset$ dan $p(M') = 0$.

Jadi, M menerima w jika dan hanya jika $p(M') = 1$ (or $(M') \in L_p$). $T((M, w)) = S((M')) = 1$ jika dan hanya jika $(M') \in L_p$ dan $(M, w) \in A_{TM}$. Jika S dapat memutuskan $(M') \in L_p$, maka T dapat memutuskan $(M, w) \in A_{TM}$. Karena A_{TM} tidak memiliki penentu, S tidak dapat ada. Reduksi yang digunakan diilustrasikan pada Gambar 11.6, di mana algoritma f melakukan apa yang ada di baris 1 dari T .

Teorema Rice menyatakan bahwa setiap properti Rice dari mesin Turing tidak dapat diputuskan. Karena banyak properti mesin Turing tidak trivial dan terkait dengan bahasa, properti ini tidak dapat diputuskan. Misalnya, teorema Rice berlaku untuk bahasa formal berikut:

- $E_{TM} = \{(M) \mid L(M) = \emptyset \text{ untuk TM } M\}$.
- $One_{TM} = \{(M) \mid M \text{ adalah TM, } |L(M)| = 1\}$.
- $Fin_{TM} = \{(M) \mid L(M) \text{ terbatas untuk TM } M\}$.
- $A10_{TM} = \{(M) \mid M \text{ menerima setidaknya 10 string}\}$.



Gambar 11.6 Ilustrasi reduksi dari A_{TM} ke L_p dalam pembuktian teorema Rice. Panah putus-putus digunakan untuk menunjukkan pemicu untuk memulai simulasi mesin Turing.

Bahasa-bahasa di atas adalah pengkodean dari beberapa properti Rice tentang $L(M)$. Misalnya, E_{TM} adalah pengkodean dari properti " $L(M) = \emptyset$," yang bukan positif. Namun, " $L(M) \neq \emptyset$ " adalah properti Rice yang positif. Menurut teorema Rice, E_{TM} tidak dapat

diputuskan. E_{TM} tidak dapat diputuskan, karena bahasa yang dapat diputuskan tertutup terhadap komplemen.

Jika suatu properti tidak terkait dengan bahasa, maka teorema Rice tidak dapat diterapkan. Misalnya, teorema Rice tidak dapat digunakan untuk menunjukkan bahwa D_{TM} tidak dapat diputuskan, karena ada dua mesin Turing, M_1 dan M_2 , sedemikian sehingga $L(M_1) = L(M_2) = \emptyset$, $(M_1) \in D_{TM}$ dan $(M_2) \notin D_{TM}$, yang melanggar kondisi "terkait bahasa".

Melihat kembali dua teorema ketidaklengkapan Gödel yang diperkenalkan di Bagian 11.1, suatu sistem formal S yang dapat melakukan operasi aritmatika umum pada bilangan asli adalah Turing lengkap karena suatu gerakan dari mesin Turing apa pun dapat disimulasikan oleh operasi aritmatika. Jika S tidak dapat dihitung, maka kedua teorema ketidaklengkapan tersebut secara trivial benar. Jika S dapat dihitung, menurut tesis Church-Turing, S dapat disimulasikan oleh mesin Turing. Dalam kasus ini, teorema ketidaklengkapan pertama Gödel setara dengan ketidakpastian A_{TM} (Teorema 11.3.4) yang merupakan pengkodean masalah penerimaan mesin Turing. Jika ketidakkonsistenan S dilambangkan dengan " $0 = 1$ " dan w adalah pengkodean " $1 = 0$ " oleh mesin Turing M yang mensimulasikan S , maka S konsisten jika dan hanya jika $w \in L(M)$. Misalkan " $w \in L(M)$ " adalah properti Rice, maka properti ini tidak dapat diputuskan oleh teorema Rice, bukti alternatif untuk teorema ketidaklengkapan kedua Gödel.

11.4 KOMPUTABILITAS BIJEKSI PENGHITUNGAN

Konsep himpunan terhitung dikaitkan dengan Georg Cantor, yang membedakan antara himpunan terhitung dan tak terhitung pada tahun 1874. Konsep himpunan komputabel lebih muda dan muncul dalam studi model komputasi pada tahun 1930-an oleh para pendiri ilmu komputer, termasuk Gödel, Church, dan Turing. Selanjutnya, kita akan melihat apa yang akan terjadi jika kita memberlakukan aturan bahwa bijeksi dapat dihitung.

Sifat-Sifat Bijeksi Penghitungan

Dalam kombinatorika, diberikan dua himpunan, misalnya X dan Y , dari objek kombinatorial, bukti bijektif adalah teknik untuk menemukan bijeksi $f: X \rightarrow Y$ untuk menunjukkan bahwa X dan Y memiliki ukuran yang sama. Teknik ini sudah lama dan berguna dan sebagian besar diterapkan pada himpunan hingga sebagai cara untuk menemukan rumus untuk ukuran X jika ukuran Y lebih mudah dihitung. Cantor menggunakan teknik populer ini untuk mendefinisikan keterhitungan himpunan tak hingga.

Contoh 11.4.1 Jumlah total fungsi dari A ke B , di mana A dan B adalah himpunan hingga, diketahui sebagai $|B|^{|A|} = n^m$, di mana $|A| = m$ dan $|B| = n$. Misalkan $F = \{f: A \rightarrow B\}$ adalah himpunan semua fungsi dari A ke B , total atau parsial. Misalkan $F' = \{f: A \rightarrow (B \cup \{u\}) \mid f \text{ adalah total}\}$, di mana $u \notin B$. Untuk setiap $f \in F$, definisikan $f'(a) = f(a)$ jika $f(a)$ terdefinisi dan $f'(a) = u$, jika tidak. Artinya, arti dari " $f'(a) = u$ " adalah " $f(a)$ tidak terdefinisi untuk $f \in F$," maka fungsi $g: F \rightarrow F'$ sedemikian sehingga $g(f) = f'$ adalah bijektif. Jadi, $|F| = |F'| = |B \cup \{u\}|^{|A|} = (n + 1)^m$.

Tanpa bijeksi g , untuk menghitung $|F|$, kita dapat menggunakan rumus berikut:

$$|F| = \sum_{S \subseteq A} |B|^{|S|} = \sum_{k=0}^m \binom{m}{k} n^k = (n+1)^m$$

di mana $m = m!/(k!(m-k)!)$ adalah koefisien binomial.

Setiap bijeksi harus total, injektif, dan surjektif. Jika tidak total, inversnya tidak surjektif. Jika $f: X \rightarrow Y$ adalah bijeksi, maka sifat-sifat f berikut ini diinginkan:

- *Menghitung.* Y adalah N (himpunan bilangan asli), dan kita katakan $f: N \rightarrow Y$ adalah bijeksi penghitung dari Y .
- *Dapat dihitung.* Untuk setiap $x \in X$, kita menghitung $y = f(x) \in Y$, dan untuk setiap $y \in Y$, kita menghitung $x = f^{-1}(y) \in X$.
- *Meningkat.* Misalkan $f: N \rightarrow Y$ adalah bijeksi penghitung dari Y dan $>$ adalah urutan baik dari Y . Untuk setiap $i \in N$, $f(i+1) > f(i)$.

Berdasarkan Definisi 1.3.2, invers dari fungsi rank adalah bijeksi penghitung, yang muncul dalam studi keterhitungan. Jika Anda ingin menunjukkan bahwa suatu himpunan tak hingga adalah terhitung, Anda harus menemukan bijeksi penghitung untuk himpunan ini. Berikut beberapa contoh bukti bijektif:

- Contoh 11.4.1 adalah salah satu dari banyak contoh dalam kombinatorika untuk menunjukkan bahwa dua himpunan hingga memiliki ukuran yang sama. Bijeksi dalam contoh ini dapat dihitung.
- Pada Bagian 1.3, kita menggunakan bukti bijektif untuk menunjukkan bahwa beberapa himpunan tak hingga adalah terhitung (Proposisi 1.3.4). Semua bijeksi yang digunakan dalam bukti Proposisi 1.3.4 adalah bijeksi penghitung yang dapat dihitung. Bijeksi tersebut juga meningkat jika kita memilih properti urutan yang baik untuk himpunan-himpunan ini.
- Kami juga menggunakan bukti bijektif untuk menunjukkan beberapa himpunan tak terhitung memiliki ukuran yang sama (Proposisi 1.3.7). Bijeksi-bijeksi ini bukanlah bijeksi yang dapat dihitung maupun bijeksi penghitung.

Contoh-contoh di atas menunjukkan bahwa bijeksi antara himpunan terhitung dapat dihitung. Pada zaman Cantor, gagasan tentang dapat dihitung belum didefinisikan secara formal. Masuk akal untuk menduga bahwa Cantor menggunakan bijeksi seperti orang lain, dengan asumsi bahwa bijeksi penghitung dapat dihitung dengan metode yang efektif. Asumsi ini menyesatkan karena kita akan menunjukkan bahwa jika suatu bahasa formal memiliki bijeksi penghitung yang dapat dihitung, maka bahasa ini dapat dikenali. Jika setiap bahasa formal dapat dihitung, bahasa yang tidak dapat dikenali harus memiliki bijeksi penghitung yang tidak dapat dihitung. Untuk mempermudah pembahasan, kami memperkenalkan konsep berikut.

Definisi 11.4.2 Suatu himpunan S dikatakan dapat dihitung jika S terbatas atau S memiliki bijeksi penghitung yang dapat dihitung.

Dalam definisi di atas, jika kata "dapat dihitung" dihilangkan, kita memperoleh definisi terhitung (Bagian 1.3.1). Dari definisi tersebut, hubungan antara terhitung dan terhitung secara komputabel diberikan oleh proposisi berikut.

Proposisi 11.4.3

- (a) Jika S terhitung secara komputabel, maka S terhitung.
- (b) S tidak terhitung secara komputabel jika dan hanya jika S tidak terhitung atau setiap bijeksi penghitungan dari S tidak terhitung.

Selanjutnya, kita akan menunjukkan bahwa suatu bahasa formal terhitung secara komputabel jika dan hanya jika bahasa tersebut dapat dikenali. Untuk memahami bukti kesetaraan antara terhitung secara komputabel dan dapat dikenali, kita membutuhkan konsep enumerator, yang merupakan varian dari mesin Turing.

Kesetaraan antara Bahasa yang Dapat Dikenali dan Bahasa yang Dapat Dihitung Secara Rekursif

Suatu enumerator adalah mesin Turing M yang berjalan selamanya setelah dimulai dan memiliki printer terpasang seperti yang diilustrasikan pada Gambar 11.7. M dapat menggunakan printer tersebut sebagai perangkat keluaran untuk mencetak string. Himpunan string yang dicetak, yang dilambangkan dengan $E(M)$, adalah bahasa yang dihitung oleh M . Suatu bahasa formal L disebut dapat dihitung secara rekursif jika terdapat enumerator M sedemikian sehingga $L = E(M)$. Perhatikan bahwa enumerator tidak memiliki masukan, sehingga pita tersebut murni ruang kerja. Printer dapat didefinisikan secara formal sebagai pita hanya tulis dengan kepala pita bergerak satu arah. Hasil berikut diketahui untuk enumerator:

Teorema 11.4.4

- (a) Jika suatu bahasa formal L dapat dihitung secara rekursif, maka L dapat dikenali.
- (b) Jika L dapat dikenali, maka L dapat dihitung secara rekursif, dan terdapat sebuah enumerator M sedemikian sehingga $L = E(M)$ dan setiap string yang dicetak oleh M adalah unik.

Bukti

- (a) Jika L dapat dihitung oleh enumerator M , kita dapat membangun pengenali S untuk memeriksa apakah $w \in L(S)$ dengan mensimulasikan M : setiap kali M mencetak string x , S membandingkan x dengan string masukan w . Jika $x = w$, S menerima; jika tidak, S melanjutkan simulasi. Mudah untuk menunjukkan bahwa $w \in L(S)$ jika dan hanya jika $w \in E(M)$; dengan demikian, $L(S) = E(M)$ dapat dikenali.
- (b) Karena L dapat dikenali, misalkan $L = L(S)$ untuk pengenali S . Kita akan menggunakan S untuk membangun enumerator M sedemikian sehingga $L = E(M)$ dan setiap string yang dicetak oleh M adalah unik.

Misalkan $f: N \rightarrow N \times N$ dan $\gamma: \Sigma^* \rightarrow N$ keduanya merupakan bijeksi (lihat Proposisi 1.3.4). M akan bekerja sebagai berikut:

$M =$ "1. $k := 0$.

2. $(i, j) := f(k); w_j := \gamma^{-1}(j)$.

3. Simulasikan S pada w_j selama i langkah.

4. Jika S menerima w_j pada langkah ke- i , **cetak** w_j .

5. $k := k + 1$; lanjutkan ke baris 2."

Untuk setiap $x \in L(S)$ dan $j = \gamma(x)$, S akan mengambil i langkah tetap untuk menerima x . Oleh karena itu, (i, j) unik untuk $x \in L(S)$ dan sesuai dengan k yang unik, karena $(i, j) = f(k)$. Dalam M di atas, setiap $k \in N$ dipertimbangkan tepat sekali, sehingga setiap $x \in L(S)$ akan dicetak sekali oleh M .

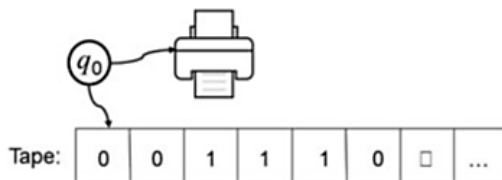
Teorema di atas menunjukkan kesetaraan antara "dapat dikenali" dan "dapat dihitung secara rekursif." Jika sebuah enumerator dapat mencetak string secara meningkat dalam urutan kanonik (Definisi 1.3.5), maka bahasa yang dienumerasi dapat diputuskan.

Teorema 11.4.5 Suatu bahasa formal L dapat diputuskan jika dan hanya jika terdapat enumerator M untuk L sedemikian sehingga string yang dicetak meningkat dalam urutan kanonik.

Teorema ini merupakan soal latihan dari Latihan 3.18.

Kesetaraan antara Dapat Dikenali dan Dapat Dihitung Secara Komputasi

Jika suatu bahasa formal L terbatas, maka bahasa tersebut dapat dihitung dan dapat diputuskan. Dalam pembahasan berikut, kita akan fokus pada kasus ketika L tak terbatas. Dua lemma berikut menunjukkan bahwa suatu bahasa formal dapat dihitung secara komputasi jika dan hanya jika bahasa tersebut dapat dikenali. Dengan demikian, dapat dikenali, dapat dihitung secara rekursif, dan dapat dihitung secara komputasi adalah konsep yang setara meskipun memiliki definisi yang berbeda.



Gambar 11.7 Ilustrasi enumerator yang merupakan mesin Turing dengan printer

Lemma 11.4.6 Setiap bahasa formal yang dapat dikenali adalah terhitung secara komputasi.

Bukti Misalkan L dapat dikenali. Kasus ketika L terbatas adalah trivial. Jika L tak terbatas, berdasarkan Teorema 11.4.4(b), kita memiliki enumerator M sedemikian sehingga $L = E(M)$ dan setiap string yang dicetak oleh M adalah unik, maka $E(M)$ terhitung karena urutan string

yang dicetak oleh M mendefinisikan bijeksi yang dapat dihitung $f: N \rightarrow E(M)$. Artinya, untuk $n \in N$, kita menghitung $f(n)$ dengan algoritma $A(n)$ sebagai berikut:

Algoritma $A(n)$:

Misalkan $c := 0$ dan simulasikan M . Ketika sebuah string x dicetak oleh M , periksa apakah $c = n$. Jika ya, kembalikan x ; jika tidak, $c := c + 1$ dan lanjutkan simulasi.

Algoritma A akan berakhir karena $E(M)$ tak terbatas dan c , yang mencatat jumlah string yang dicetak oleh M , pada akhirnya akan mencapai n . Algoritma $A(n)$ dapat dimodifikasi sebagai $A(w)$ untuk menghitung $f^{-1}(w)$ untuk $w \in E(M)$: alih-alih memeriksa $c = n$, periksa apakah $x = w$ dan, jika ya, kembalikan c . Karena f dan f^{-1} keduanya merupakan fungsi total, f adalah bijeksi (Proposisi 1.3.1). Algoritma $A(n)$ adalah bukti bahwa f dapat dihitung.

Lemma 11.4.7 Setiap bahasa formal yang dapat dihitung dapat dikenali.

Bukti Kasus ketika L terbatas adalah trivial. Jika L dapat dihitung, maka ada bijeksi yang dapat dihitung $f: N \rightarrow L$. Dengan demikian, kita dapat menggunakan f untuk merancang enumerator M yang mencetak string $f(0), f(1), f(2)$, dan seterusnya. Jelas bahwa $E(M) = L$. Berdasarkan Teorema 11.4.4(a), L dapat dikenali. Teorema berikut memperluas hasil di atas dari bahasa formal ke himpunan umum.

Teorema 11.4.8 Suatu himpunan dapat dihitung secara komputabel jika dan hanya jika himpunan tersebut dapat dihitung.

Bukti Misalkan S adalah himpunan apa pun. Kasus ketika S berhingga adalah trivial. Jika S dapat dihitung secara komputabel, maka ada bijeksi komputabel $f: N \rightarrow S$. Oleh karena itu, S dapat direpresentasikan oleh bahasa formal $\{(f(0)), (f(1)), \dots\}$, di mana $(f(i))$ adalah string yang merepresentasikan $f(i)$. Berdasarkan Lemma 11.4.7, S dapat dikenali, sehingga dapat dihitung.

Ketika S tidak dapat dihitung secara komputabel, ada dua kasus yang perlu dipertimbangkan: S dapat direpresentasikan oleh bahasa formal atau tidak. Jika ya, maka S tidak dapat dikenali berdasarkan Lemma 11.4.6, sehingga tidak dapat dihitung. Jika tidak, maka S tidak dapat dikenali oleh mesin Turing mana pun, sehingga tidak dapat dihitung.

Proposisi 11.4.9 Jika X tak hingga terhitung, maka

1. X dapat diputuskan jika dan hanya jika ada bijeksi $f: N \rightarrow X$ yang dapat dihitung dan meningkat.
2. X dapat dihitung jika dan hanya jika ada bijeksi $f: N \rightarrow X$ yang dapat dihitung.

3. X tidak dapat dihitung jika dan hanya jika setiap bijeksi $f: N \rightarrow X$ tidak dapat dihitung.

Bukti Berdasarkan Teorema 11.4.8, X dapat dihitung jika dan hanya jika X dapat dihitung dan dihitung. (3) berasal dari Proposisi 11.4.3, meskipun keberadaan bijeksi penghitung yang tidak dapat dihitung tidak diketahui. (2) secara logis setara dengan (3). Untuk (1), X dapat diputuskan jika dan hanya jika elemen-elemen X dapat dihitung dalam urutan meningkat. Enumerasi ini mendefinisikan bijeksi yang meningkat dan dapat dihitung seperti pada bukti Lemma 11.4.6, dan bijeksi yang meningkat dan dapat dihitung akan menghasilkan enumerator yang menghitung elemen dalam urutan meningkat seperti pada bukti Lemma 11.4.7.

Proposisi di atas menunjukkan bahwa sifat-sifat bijeksi penghitungan $f: N \rightarrow X$ sangat penting untuk menentukan apakah X dapat diputuskan atau dapat dihitung. Secara khusus, jika X dapat dihitung tetapi tidak dapat diputuskan, maka harus ada bijeksi penghitungan X yang dapat dihitung tetapi tidak meningkat. Untuk setiap bahasa L yang dapat dikenali, terdapat mesin Turing M sedemikian sehingga

$$L = \{w \in \Sigma^* \mid M(w) = 1\}$$

Artinya, L dapat dikenali atau dihitung karena $M(w)$ dapat dihitung, di mana M dipandang sebagai fungsi keputusan (lihat Definisi 11.2.9). Dari Teorema 11.4.8, kita memiliki hasil berikut.

Korollari 11.4.10 Misalkan $p: N \rightarrow \{0, 1\}$ adalah fungsi keputusan dan $S = \{x \in N \mid p(x) = 1\}$.

- S dapat dihitung secara total jika dan hanya jika $p(x)$ dapat dihitung secara total.
- S dapat dihitung secara terhingga jika dan hanya jika $p(x)$ dapat dihitung.
- S tidak dapat dihitung secara terhingga jika dan hanya jika $p(x)$ tidak dapat dihitung.

Hasil di atas menunjukkan bahwa, ketika $p(x)$ menjadi sulit dihitung, S mungkin tidak dapat dihitung secara terhingga. Beberapa orang menganggap bahwa $p(x)$ dapat dihitung dan sampai pada kesimpulan bahwa S dapat dihitung secara terhingga. Jika $p(x)$ adalah formula orde pertama, maka keberadaan himpunan yang tidak dapat dihitung memberi kita formula orde pertama yang tidak dapat dihitung. Pada bagian selanjutnya, kita akan menunjukkan tujuan bab ini: beberapa formula orde pertama (misalnya, $p(x)$) tidak dapat dihitung dan beberapa masalah keputusan dalam logika orde pertama tidak dapat dihitung.

Banyak bahasa formal tidak dapat dikenali, dan bahasa yang tidak dapat dikenali tidak dapat dihitung secara terhingga menurut Lemma 11.4.7. Artinya, Lemma 11.4.6 dan 11.4.7

memberikan cara alternatif untuk menunjukkan apakah suatu bahasa dapat dihitung secara terhitung atau tidak. Pertimbangkan pengkodean berikut dari beberapa masalah keputusan:

- $G_{TM} = \{(M) \mid \text{Mesin Turing } M \text{ terdefinisi dengan baik}\}$
- $E_{TM} = \{(M) \mid L(M) = \emptyset \text{ untuk mesin Turing } M\}$
- $N_{TM} = \{(M) \mid L(M) \neq \emptyset \text{ untuk mesin Turing } M\}$
- $All_{TM} = \{(M) \mid L(M) = \Sigma^* \text{ untuk mesin Turing } M\}$
- $D_{TM} = \{(M) \mid \text{mesin Turing } M \text{ adalah penentu}\}$
- $Al_{TM} = \{(M) \mid \text{mesin Turing } M \text{ adalah algoritma}\}$

Mereka adalah bahasa formal karena $(M) \in \Sigma^*$ dapat diputuskan karena kita memiliki penentu yang menerima setiap masukan. G_{TM} adalah pengkodean dari semua mesin Turing yang terdefinisi dengan baik. G_{TM} dapat diputuskan dengan asumsi bahwa kita memiliki dekoder yang efisien untuk memeriksa apakah (M) berasal dari M yang terdefinisi dengan baik. Berdasarkan Lemma 11.4.6, G_{TM} dapat dihitung secara komputabel, oleh karena itu dapat dihitung.

E_{TM} , N_{TM} , dan All_{TM} adalah pengkodean dari semua mesin Turing yang masing-masing menerima tidak ada, sesuatu, dan segalanya. N_{TM} adalah komplemen dari E_{TM} . Diketahui bahwa N_{TM} dapat dikenali dan E_{TM} tidak dapat dikenali (Proposisi 11.4.15). Berdasarkan Lemma 11.4.6, N_{TM} dapat dihitung secara komputabel. Berdasarkan Lemma 11.4.7, E_{TM} tidak dapat dihitung secara komputabel. Baik All_{TM} maupun komplemennya tidak dapat dikenali (Proposisi 11.4.17), oleh karena itu, tidak dapat dihitung secara komputabel.

Proposisi 11.4.11 Himpunan semua penentu tidak dapat dihitung secara komputabel; oleh karena itu, D_{TM} tidak dapat dikenali.

Bukti Asumsikan D_{TM} dapat dihitung secara komputabel dengan bijeksi komputabel $h: N \rightarrow D_{TM}$ sedemikian sehingga $h(i) = (M_i)$ untuk setiap $i \in N$. Dengan menggunakan h dan γ (Definisi 1.3.5), kita membangun mesin Turing X berikut:

$X = \text{"On input } w \in \Sigma^*$

1. Hitung $i = \gamma(w)$ dan $h(i) = (M_i) \in D_{TM}$.
2. Simulasikan M_i pada w sampai M_i berhenti. // M_i adalah penentu.
3. **return** $1 - M_i(w)$. // yaitu, $X(w) = \neg M_i(w)$.

Karena setiap langkah X berhenti, $(X) \in D_{TM}$. Misalkan $X = M_k$ untuk beberapa $k \in N$. Namun, $X(w_k) \neq M_k(w_k)$, sebuah kontradiksi dengan $X = M_k$. Jadi, asumsinya salah dan D_{TM} tidak dapat dihitung secara komputabel. Berdasarkan Lemma 11.4.6, D_{TM} tidak dapat dikenali.

Proposisi 11.4.12 Himpunan semua algoritma tidak dapat dihitung secara komputabel; Oleh karena itu, Al_{TM} tidak dapat dikenali.

Bukti Buktinya analog dengan Proposisi 11.4.11, di mana baris 3 dari X diubah menjadi "**return** $1 + M_i(w)$."

Perhatikan bahwa $D_{TM} \subset G_{TM}$ dan G_{TM} dapat dihitung. Jika D_{TM} dapat dihitung, maka setiap bijeksi penghitungan D_{TM} tidak dapat dihitung (Proposisi 11.4.3(b)).

Mengidentifikasi Bahasa yang Tidak Dapat Dikenali

Untuk menunjukkan bahwa suatu bahasa tidak dapat dikenali, kita dapat membuktikan bahwa bahasa tersebut tidak dapat dihitung secara komputabel, dan D_{TM} adalah contohnya (Proposisi 11.4.11). Dalam praktiknya, kita memiliki beberapa metode untuk menunjukkan ketidakmampuan mengenali suatu bahasa, dan metode-metode ini dapat digunakan untuk menunjukkan bahwa suatu bahasa tidak dapat dihitung secara komputabel. Berikut ini, kita akan meneliti metode-metode tersebut.

Proposisi 11.4.13 Jika suatu bahasa formal L dapat dikenali tetapi tidak dapat diputuskan, maka L tidak dapat dikenali.

Bukti Ini adalah konsekuensi logis dari Proposisi 11.3.13.

Dengan menerapkan proposisi di atas pada A_{TM} dan H_{TM} dengan Proposisi 11.3.11 dan 11.3.12, kita mendapatkan hasil berikut.

Proposisi 11.4.14 $\overline{A_{TM}}$ dan $\overline{H_{TM}}$ tidak dapat dikenali.

Teorema Rice sangat berguna untuk menunjukkan bahwa beberapa bahasa tidak dapat diputuskan. Jika p adalah properti Rice, maka $L_p = \{ \langle M \rangle \mid p(M) = 1 \}$ tidak dapat diputuskan. Jika kita juga mengetahui bahwa L_p dapat dikenali, maka L_p harus tidak dapat dikenali berdasarkan Proposisi 11.4.13. Misalnya, E_{TM} dapat ditunjukkan tidak dapat dikenali dengan cara ini, karena E_{TM} dapat dikenali (sebuah latihan).

Proposisi 11.4.15 E_{TM} tidak dapat dikenali.

Analog dengan Proposisi 11.3.7, kita memiliki hasil berikut dari Definisi 11.3.5:

Proposisi 11.4.16 Jika Y direduksi menjadi X dan Y tidak dapat dikenali, maka X tidak dapat dikenali.

Proposisi di atas sangat berguna untuk menunjukkan bahwa beberapa bahasa formal tidak dapat dikenali.

Proposisi 11.4.17 Baik All_{TM} maupun $\overline{All_{TM}}$ tidak dapat dikenali.

Bukti Kita akan terlebih dahulu mengerjakan All_{TM} . Asumsikan All_{TM} dapat dikenali dan A adalah pengenalnya. Kita mereduksi $\overline{A_{TM}}$ ke All_{TM} dengan membangun pengenal B untuk $\overline{A_{TM}}$ sebagai berikut, dengan asumsi keadaan awal M' bukan keadaan penerimaan:

$B =$ “Pada input (M, w) , M adalah mesin Turing dan w adalah input ke M .

1. Hasilkan kode mesin Turing M' (lihat di bawah), (M') , dari (M, w) .
2. **return** $A((M'))$.”

$M' =$ “Pada input x

1. Simulasikan M pada w paling banyak $|x|$ langkah.
2. Jika M menerima w pada baris 1, pergi ke keadaan loop.
3. **return** 1.”

Hubungan antara M dan M' diberikan sebagai berikut:

1. M menerima w dalam n langkah: M' berulang pada setiap x jika $|x| \geq n$ dan menerima x jika $|x| < n$.
2. M tidak menerima w : M' menerima setiap x .

Dengan demikian, M tidak menerima w jika dan hanya jika M' menerima setiap x . $B((M, w)) = A((M')) = 1$ jika dan hanya jika $(M') \in \overline{All_{TM}}$ dan $(M, w) \in A_{TM}$. Oleh karena itu, B adalah pengenalan untuk $\overline{A_{TM}}$ jika A adalah pengenalan untuk All_{TM} . Karena $\overline{A_{TM}}$ tidak memiliki pengenalan, A tidak dapat ada dan All_{TM} harus tidak dapat dikenali.

Sekarang pada $\overline{All_{TM}}$. Asumsikan itu dapat dikenali dan A adalah pengenalnya. Kita akan menggunakan B yang sama dalam bukti Proposisi 11.3.10 untuk mereduksi $\overline{A_{TM}}$ ke $\overline{All_{TM}}$. Satu-satunya perbedaan adalah bahwa sekarang A adalah pengenalan hipotetis untuk All_{TM} , bukan penentu untuk D_{TM} , dan $+$ adalah pengenalan untuk $\overline{A_{TM}}$. Hubungan antara M dan M' diberikan sebagai berikut:

1. M menerima w : M' menerima setiap x dan $L(M') = \Sigma^*$.
2. M tidak menerima w : M' tidak menerima x apa pun dan $L(M') = \emptyset$.

Dengan demikian, M tidak menerima w jika dan hanya jika $L(M') \neq \Sigma^*$. $B((M, w)) = A((M')) = 1$ jika dan hanya jika $L(M') \neq \Sigma^*$ dan $(M, w) \in \overline{A_{TM}}$. Oleh karena itu, B memang merupakan pengenalan untuk $\overline{A_{TM}}$ dan $\overline{A_{TM}}$ direduksi menjadi $\overline{All_{TM}}$. Berdasarkan Proposisi 11.4.14, $\overline{A_{TM}}$ tidak dapat dikenali. Berdasarkan Proposisi 11.4.16, $\overline{All_{TM}}$ harus tidak dapat dikenali.

Perhatikan bahwa dua reduksi yang digunakan dalam bukti di atas adalah reduksi pemetaan. Reduksi pertama berasal dari Prof. Tian Liu dari Universitas Beijing. Pemetaan yang sama dari reduksi kedua digunakan untuk menunjukkan bahwa A_{TM} direduksi menjadi D_{TM} (Proposisi 11.3.10). Dengan menerapkan Proposisi 11.3.6, $\overline{A_{TM}}$ direduksi menjadi $\overline{D_{TM}}$. Dengan menerapkan Proposisi 11.4.16, kita mendapatkan hasil berikut.

Proposisi 11.4.18 $\overline{D_{TM}}$ tidak dapat dikenali.

All_{TM} adalah contoh pertama kita untuk pernyataan “baik L dan $\bar{L}$ tidak dapat dikenali” dan DTM adalah contoh kedua kita.

Singkatnya, kita memiliki tiga metode untuk menunjukkan bahwa bahasa formal X tidak dapat dikenali:

1. Jika X tidak dapat dihitung secara komputabel, maka X tidak dapat dikenali (Lemma 11.4.6).

Contoh: D_{TM}, A_{TM}

2. Jika X dapat dikenali tetapi tidak dapat diputuskan, maka X tidak dapat dikenali (Proposisi 11.4.13).

Contoh: $E_{TM}, \overline{A_{TM}}, \overline{H_{TM}}$

3. Jika Y direduksi menjadi X dan Y tidak dapat dikenali, maka X tidak dapat dikenali (Proposisi 11.4.16).

Contoh: $All_{TM}, \overline{All_{TM}}, \overline{D_{TM}}$

Menggabungkan hasil di atas dengan Lemma 11.4.7, kita memiliki hasil berikut.

Proposisi 11.4.19 Bahasa formal berikut tidak dapat dihitung secara komputabel:

$$E_{TM}, All_{TM}, \overline{All_{TM}}, \overline{D_{TM}}, \overline{A_{TM}}, \overline{H_{TM}}$$

11.5 KELENGKAPAN TURING

Menurut tesis Church-Turing, setiap model komputasi dapat disimulasikan oleh mesin Turing. Di sisi lain, mudah untuk merancang model komputasi baru yang memiliki kekuatan yang sama dengan mesin Turing, selama model komputasi baru tersebut terbukti mampu melakukan apa yang dapat dilakukan oleh mesin Turing. Konsep kelengkapan Turing hanya mengekspresikan sifat ini: jika suatu model komputasi memiliki kemampuan untuk mensimulasikan mesin Turing, maka model tersebut lengkap Turing. Baik fungsi rekursif Gödel maupun kalkulus λ Church adalah lengkap Turing.

Contoh 11.5.1 Mesin dua tumpukan adalah variasi dari mesin Turing di mana pita diganti dengan dua tumpukan: tumpukan pertama menyimpan semua simbol dari kepala pita (termasuk) ke kanan (tidak termasuk spasi kosong di belakang); tumpukan kedua menyimpan semua simbol dari kepala pita (tidak termasuk) ke kiri. Informasi keadaan tetap sama. Mudah untuk menunjukkan bahwa mesin dua tumpukan bersifat Turing lengkap dengan mensimulasikan semua gerakan Turing dengan operasi pada tumpukan. Awalnya, string input disimpan dalam tumpukan pertama dengan simbol pertama di atas tumpukan. Simulasi gerakan Turing berjalan sebagai berikut: simbol a dikeluarkan dari tumpukan pertama; jika tumpukan kosong, gunakan simbol kosong. Jika $\delta(q, a) = (p, b, R)$, masukkan b ke dalam tumpukan kedua; jika $\delta(q, a) = (p, b, L)$, masukkan b ke dalam tumpukan pertama dan kemudian pindahkan simbol teratas dari tumpukan kedua (jika tumpukan kedua kosong, gunakan simbol kosong) ke tumpukan pertama dengan operasi tumpukan.

Signifikansi dari sifat Turing lengkap adalah bahwa semua properti yang tidak dapat diputuskan dari mesin Turing akan diwarisi oleh model komputasi baru. Dengan demikian, masalah penghentian mesin dua tumpukan tidak dapat diputuskan, karena jika kita memiliki algoritma

untuk menentukan penghentian mesin dua tumpukan, algoritma ini dapat digunakan untuk menunjukkan penghentian mesin Turing. Demikian pula, jika kita dapat menunjukkan kelengkapan Turing dari suatu logika, kita dapat memperoleh hasil yang sama.

Kelengkapan Turing Prolog

Proposisi 11.5.2 Prolog adalah Turing lengkap.

Bukti Kami menunjukkan bagaimana mensimulasikan mesin Turing dengan program Prolog, menggunakan ide mesin dua tumpukan. Isi pita mesin Turing dibagi menjadi dua daftar, L dan R , yang sesuai dengan dua tumpukan dalam mesin dua tumpukan. Awalnya, $L = []$ dan R berisi string input. Kita mengubah L menjadi L' dan R menjadi R' sesuai dengan pergerakan mesin Turing, hingga mesin Turing berhenti. Transformasi tersebut diilustrasikan oleh program Prolog untuk mesin Turing dalam Contoh 11.2.4.

```
% M1 checks  $a^j b^k$  for  $j \leq k$ . Its delta definition is defined below.
delta(q0, b1, qa, b1, right).
delta(q0, a, q1, x, right).
delta(q0, b, q3, b, right).
delta(q0, y, q3, y, right).
delta(q1, a, q1, a, right).
delta(q1, y, q1, y, right).
delta(q1, b, q2, y, left).
delta(q2, y, q2, y, left).
delta(q2, a, q2, a, left).
delta(q2, x, q0, x, right).
delta(q3, b, q3, b, right).
delta(q3, y, q3, y, right).
delta(q3, b1, qa, b1, left).

% General code for all Turing machines
turing(W, Tape) :-
    write(' '), write(q0), write(W), nl, % initial configuration
    move([], L, q0, W, R),
    reverse(L, Lr),
    append(Lr, [qa|R], Tape).

% move(L1, L2, Q, R1, R2): move from (L1, Q, R1) to (L2, P, R2)
move(L, L, qa, R, R) :- !. % halt on qa, the accept state
move(L0, L, Q, R0, R) :- % simulate one TM move
    current(R0, A, Rest), % load the current symbol in A
    delta(Q, A, P, B, Direct), % delta(Q, A) = (P, B, Direct)
    result(Direct, L0, L1, B, Rest, R1), % update L and R
    write('|- '), write(rev(L1)), write(P), write(R1), nl,
    move(L1, L, P, R1, R). % continue to move
```

```

% current(R1, Sym, R2): pick current symbol Sym from R1
current([], b1, []).           % current symbol is the blank b1
current([Sym|Rest], Sym, Rest). % current symbol is Sym

% result(dir, L1, L2, B, R1, R2): B is the symbol pointed by tape head
% According to "dir", stack1 changes from L1 to L2.
% stack2 changes from R1 to R2,
result(left, [C|L], L, B, R, [C,B|R]). % move top symbol of L to R
result(right, L, [B|L], B, R, R).      % push B at the top of L

?- turing([a,a,b,b], Tape).
   q0[a,a,b,b]
|- rev([x])q1[a,b,b]
|- rev([a,x])q1[b,b]
|- rev([x])q2[a,y,b]
|- rev([])q2[x,a,y,b]
|- rev([x])q0[a,y,b]
|- rev([x,x])q1[y,b]
|- rev([y,x,x])q1[b]
|- rev([x,x])q2[y,y]
|- rev([x])q2[x,y,y]
|- rev([x,x])q0[y,y]
|- rev([y,x,x])q3[y]
|- rev([y,y,x,x])q3[]
|- rev([y,x,x])qa[y,b1]
Tape = [x,x,y,qa,y,b1].

```

Ini mengilustrasikan bagaimana urutan konfigurasi mesin Turing diimplementasikan dalam Prolog sebagai relasi antar konfigurasi.

Untuk setiap mesin Turing, kita dapat menentukan fungsi delta-nya seperti yang diilustrasikan dalam contoh di atas, dan kode setelah definisi delta dapat digunakan kembali.

Bukti di atas menunjukkan bahwa setiap mesin Turing dapat disimulasikan oleh kode di atas selama memori yang digunakan oleh Prolog tidak terbatas. Kode Prolog dapat digunakan sebagai alat debugging untuk mendesain mesin Turing pada input yang diberikan.

Teorema 11.5.3 Penghentian program Prolog tidak dapat diputuskan.

Bukti Sketsa Jika kita memiliki algoritma untuk memutuskan penghentian program Prolog pada input apa pun, maka kita dapat menggunakan algoritma tersebut untuk memutuskan penghentian mesin Turing pada input, karena pergerakan mesin Turing dapat disimulasikan oleh program Prolog. Penghentian program Prolog dalam simulasi memastikan penghentian mesin Turing.

Teorema 11.5.4 Keterpuasan suatu himpunan klausa Horn tidak dapat diputuskan.

Bukti Sketsa Misalkan H adalah klausa yang digunakan dalam program Prolog dari bukti Proposisi 11.5.2 untuk mensimulasikan mesin Turing M , termasuk klausa negatif yang dihasilkan dari kueri yang diturunkan dari q_0w . Mudah untuk memeriksa bahwa H adalah himpunan klausa Horn. Jika kita memiliki algoritma A untuk memberi tahu bahwa H tidak dapat dipenuhi, maka ada bukti resolusi H . Bukti resolusi ini memastikan bahwa w diterima oleh M . Dengan demikian, algoritma A dapat memberi tahu apakah mesin Turing M menerima w atau tidak. Ini merupakan kontradiksi dengan fakta bahwa masalah penerimaan mesin Turing tidak dapat diputuskan.

Teorema 11.5.5 Keterpuasan suatu rumus orde pertama tidak dapat diputuskan.

Bukti diserahkan sebagai latihan.

Kelengkapan Turing dari Sistem Penulisan Ulang

Di Bagian. 7.2.5, kita secara singkat memperkenalkan konsep sistem penulisan ulang string (SRS). Axel Thue memperkenalkan gagasan ini dengan harapan dapat menyelesaikan masalah kata untuk sistem Thue dari alfabet terbatas: diberikan SRS (Σ, R) , misalkan $=_R$ menunjukkan penutupan simetris dari $\Rightarrow$, relasi transitif dan monoton yang diinduksi oleh R atas Σ^* , bagaimana memutuskan $s =_R t$ untuk setiap $s, t \in \Sigma^*$? Baru pada tahun 1947 masalah kata tersebut terbukti tidak dapat diputuskan—hasil ini diperoleh secara independen oleh Emil Post dan Andrey Markov.

Proposisi 11.5.6 SRS adalah Turing lengkap.

Bukti Misalkan M adalah mesin Turing apa pun. Dengan asumsi keadaan dan pita tidak berbagi simbol, setiap konfigurasi M adalah string simbol pita dan keadaan berbentuk $\alpha q \beta$, di mana q adalah simbol keadaan, dan α dan β adalah string simbol pita. Kita dapat menggunakan SRS R untuk menggambarkan semua pergerakan M sebagai berikut: konfigurasi awal q_0w direpresentasikan oleh string $q_0w\$$, di mana $\$$ adalah penanda akhir dari semua simbol non-kosong pada pita M . Jika $\delta(q, a) = (p, b, R)$, kita memiliki aturan penulisan ulang $qa \rightarrow bp$; lebih lanjut, jika a adalah $\sqcup$ (simbol kosong), yaitu, $\delta(q, \sqcup) = (p, b, R)$, kita menambahkan aturan penulisan ulang $q\$ \rightarrow bp\$$. Jika $\delta(q, a) = (p, b, L)$, kita memiliki serangkaian aturan penulisan ulang: untuk setiap simbol pita c , $cqa \rightarrow pcb$; lebih lanjut, jika a adalah $\sqcup$, kita menambahkan aturan penulisan ulang berikut: $cq\$ \rightarrow pcb\$$, di mana c adalah simbol pita apa pun. Kemudian, mudah untuk menunjukkan bahwa $q_0w \vdash^* \alpha q_a \beta$ di M jika dan hanya jika $q_0w\$ \Rightarrow^* \alpha q_a \beta\$$ di R .

Contoh 11.5.7 Untuk mesin Turing M_0 yang diberikan dalam Contoh 11.2.3, aturan penulisan ulang untuk setiap langkah delta yang dihasilkan oleh bukti di atas diberikan di bawah ini:

$$1. \delta(q_0, 0) = (q_1, 0, R) \qquad q_0 0 \rightarrow 0 q_1$$

$$\begin{array}{ll}
2. \delta(q_1, 0) = (q_1, 0, R) & q_1 0 \rightarrow 0q_1 \\
3. \delta(q_1, 1) = (q_2, 1, R) & q_1 1 \rightarrow 1q_2 \\
4. \delta(q_2, 1) = (q_2, 1, R) & q_2 1 \rightarrow 1q_2 \\
5. \delta(q_2, \sqcup) = (q_a, \sqcup, R) & q_2 \sqcup \rightarrow \sqcup q_2, q_a \$ \rightarrow \sqcup q_a \$
\end{array}$$

Mudah untuk melihat bahwa $q_0 0011\$ \Rightarrow^* 0011 \sqcup \$$ oleh SRS yang terdiri dari aturan penulisan ulang di atas. Karena SRS adalah kasus khusus dari TRS (sistem penulisan ulang istilah), kelengkapan Turing dari SRS menyiratkan kelengkapan Turing dari TRS.

Teorema 11.5.8 Penghentian sistem penulisan ulang tidak dapat diputuskan.

Bukti Sketsa Jika kita memiliki algoritma untuk memutuskan penghentian sistem penulisan ulang apa pun (baik SRS atau TRS), maka kita dapat menggunakan algoritma tersebut untuk memutuskan penghentian mesin Turing karena langkah-langkah mesin Turing dapat dinyatakan sebagai himpunan aturan penulisan ulang. Penghentian sistem penulisan ulang memastikan penghentian mesin Turing.

Teorema 11.5.9 Masalah kata dari sistem Thue tidak dapat diputuskan.

Bukti Sketsa Jika kita memiliki algoritma untuk memutuskan masalah kata sistem Thue, maka kita dapat menggunakan algoritma tersebut untuk memutuskan masalah penerimaan mesin Turing (dikodekan sebagai A_{TM}): diberikan mesin Turing M dan string input w , kita membuat M' dari M dengan memperkenalkan keadaan penerimaan baru q_f sedemikian sehingga M' mensimulasikan M pada w , dan setelah M memasuki keadaan penerimaan q_a , M' akan menghapus setiap simbol non-kosong (diganti dengan simbol kosong) sebelum memasuki q_f , sehingga konfigurasi penerimaan M' adalah unik. Sekarang, misalkan $s = q_0 w \$$ (konfigurasi awal) dan $t = q_f \$$ (konfigurasi penerimaan), maka M menerima w jika dan hanya jika $s =_R t$, di mana R adalah SRS yang dibuat dari definisi M' seperti yang kita lakukan pada Contoh 11.5.7. Ingat kembali bahwa masalah kekongruenan logika orde pertama dengan kesamaan (Definisi 7.1.7) adalah masalah memutuskan apakah $s =_E t$ atau tidak untuk dua suku s dan t , di mana E adalah himpunan persamaan yang diberikan. Berikut ini adalah konsekuensi dari masalah kata.

Korolari 11.5.10 Masalah kekongruenan logika orde pertama dengan kesamaan tidak dapat diputuskan.

Dalam bab ini, kita telah memperkenalkan konsep mesin Turing dan membuktikan secara formal bahwa masalah penghentian mesin Turing yang terkenal tidak dapat diputuskan. Karena program Prolog atau sistem penulisan ulang dapat mensimulasikan gerakan mesin Turing apa pun, penghentian program Prolog atau sistem penulisan ulang juga tidak dapat diputuskan. Masalah penerimaan mesin Turing juga tidak dapat diputuskan. Karena masalah

ini dapat ditentukan oleh himpunan klausa Horn, ini menyiratkan bahwa kepuasan klausa Horn tidak dapat diputuskan. Akibatnya, keterpuasan logika orde pertama tidak dapat diputuskan. Keterpuasan logika apa pun yang mengandung logika orde pertama, seperti logika orde tinggi, juga tidak dapat diputuskan. Hasil teoritis ini membantu kita memahami logika secara menyeluruh dan menghindari upaya yang tidak perlu untuk menemukan algoritma untuk masalah yang tidak dapat diputuskan ini.

11.6 LATIHAN SOAL

1. Untuk mesin Turing M_1 pada Contoh 11.2.1, berikan urutan langkah M_1 untuk setiap string input: (a) a^2b^3 ; (b) a^2b ; (c) ab^2a . Setiap urutan harus diakhiri dengan M_1 menerima atau menolak input.
2. Berikan desain lengkap mesin Turing M_2 pada Contoh 11.2.10 untuk mengenali $A_2 = \{b^i a^j b^k \mid i * j = k, i, j \geq 0\}$. Artinya, berikan definisi δ terperinci dari M_2 dan kelompokkan langkah-langkah dari definisi Anda sesuai dengan deskripsi tingkat tinggi dari M_2 . Berikan urutan langkah pada string input $w = b_2 a_1 b_2$.
3. Berikan deskripsi tingkat tinggi dari mesin Turing M_3 untuk mengenali $A_3 = \{a^i b^j c^k \mid k = [i/j], i, j \geq 0\}$. Setiap langkah deskripsi menggunakan mesin Turing yang sudah dikenal atau diungkapkan (mungkin secara informal) oleh urutan gerakan mesin Turing.
4. Berikan deskripsi tingkat tinggi dari mesin Turing M_4 untuk mengenali $A_4 = \{a^i b^j \mid i, j \geq 0\}$, dan pita berisi tepat $a^i b^j$ ketika M_4 berhenti dengan q . Setiap langkah deskripsi menggunakan mesin Turing yang sudah dikenal atau diungkapkan oleh urutan gerakan mesin Turing.
5. Misalkan A adalah bahasa formal. Tunjukkan bahwa A dapat diputuskan jika (a) $|A| = 1$; (b) A terbatas.
6. Misalkan $C10_{TM} = \{(M) \mid TM \text{ } M \text{ bergerak paling banyak sepuluh langkah untuk setiap string masukan}\}$. Tunjukkan bahwa $C10_{TM}$ dapat diputuskan dengan merancang sebuah penentu.
7. Tunjukkan bahwa bahasa-bahasa berikut ini dapat diputuskan (decidable):
 - (a) A_1 adalah himpunan semua bilangan biner yang habis dibagi 4.
 - (b) A_2 adalah himpunan semua bilangan biner yang habis dibagi 3.
 - (c) $A_3 = \{0^i 1^j 0^k \mid i + j = k\}$.
8. Tunjukkan bahwa bahasa-bahasa yang dapat diputuskan (decidable) bersifat tertutup terhadap (a) gabungan (union), (b) irisan (intersection), dan (c) komplemen (complement). Artinya, misalkan A dan B dapat diputuskan (decidable), maka bahasa-bahasa berikut ini dapat diputuskan (decidable): (a) $A \cup B$; (b) $A \cap B$; (c) $\bar{A} = \Sigma^* - A$.
9. Tunjukkan bahwa bahasa-bahasa yang dapat dikenali (recognitive) bersifat tertutup terhadap (a) gabungan (union) dan (b) irisan (intersection). Artinya, misalkan A dan B dapat dikenali (recognitive), maka bahasa-bahasa berikut ini dapat dikenali (recognitive): (a) $A \cup B$; (b) $A \cap B$.
10. Berikan reduksi pemetaan dari A_{TM} ke H_{TM} .

11. Buktikan secara formal bahwa kebenaran parsial dari tripel Hoare tidak dapat diputuskan.
12. Buktikan dengan reduksi dari ATM bahwa sifat-sifat berikut dari mesin Turing tidak dapat diputuskan (teorema Rice tidak dapat digunakan):
 - (a) $One_{TM} = \{(M) \mid L(M) = \{0\}\}$.
 - (b) $Two_{TM} = \{(M) \mid L(M) = \{0, 1\}\}$.
 - (c) $Fin_{TM} = \{(M) \mid L(M) \text{ terbatas}\}$.
 - (d) $A10_{TM} = \{(M) \mid M \text{ menerima setidaknya sepuluh string}\}$.
13. (Teorema 11.4.5) Buktikan bahwa suatu bahasa formal L dapat diputuskan jika dan hanya jika terdapat enumerator M untuk L sedemikian sehingga string yang dicetak meningkat dalam urutan kanonik.
14. Buktikan bahwa $\overline{E_{TM}}$ dapat dikenali, di mana $E_{TM} = \{(M) \mid L(M) = \emptyset \text{ untuk TM } M\}$.
15. Buktikan bahwa jika A dan B keduanya dapat dihitung secara komputabel, maka $A \cup B$ dan $A \cap B$ dapat dihitung secara komputabel.
16. Buktikan bahwa himpunan fungsi keputusan (total atau parsial), $D_F = \{f: N \rightarrow \{0, 1\}\}$, tidak dapat dihitung secara komputabel. (Petunjuk: perluas Contoh 11.4.1 ke $A = N$ dan $B = \{0, 1\}$).
17. Misalkan $One_{TM} = \{(M) \mid |L(M)| = 1\}$. Buktikan bahwa baik One_{TM} maupun $\overline{One_{TM}}$ tidak dapat dikenali.
18. Misalkan $Fin_{TM} = \{(M) \mid L(M) \text{ terbatas}\}$. Buktikan bahwa baik Fin_{TM} maupun $\overline{Fin_{TM}}$ tidak dapat dikenali.
19. Misalkan $De_{TM} = \{(M) \mid L(M) \text{ dapat diputuskan untuk TM } M\}$. Buktikan bahwa baik De_{TM} maupun $\overline{De_{TM}}$ tidak dapat dikenali.
20. Buktikan bahwa D_{TM} tidak dapat dikenali tanpa Proposisi 11.4.11.
21. Beberapa mesin Turing memungkinkan kepala pita tetap diam setelah pergerakan (hanya mengubah keadaan atau simbol saat ini). Bagaimana mensimulasikan tindakan ini dalam program Prolog pada Bagian 11.5?
22. Beberapa mesin Turing memungkinkan pita memiliki simbol tak terbatas di sisi kiri kepala pita (sehingga tidak ada "pergerakan keluar dari pita"). Bagaimana mensimulasikan tindakan ini dalam program Prolog pada Bagian 11.5.1? Mohon berikan solusi yang detail.
23. Buktikan bahwa kepuasan suatu rumus orde pertama tidak dapat diputuskan.

BAB 12

PROSEDUR PENGAMBILAN KEPUTUSAN

Desain sistem komputer, baik perangkat lunak maupun perangkat keras, rentan terhadap kesalahan, dan verifikasi formal dari desain tersebut membutuhkan alat penalaran otomatis yang ampuh. Prosedur pengambilan keputusan merupakan komponen penting dari alat-alat ini. Ingat bahwa dalam Bab 1 (Bagian 1.3), prosedur pengambilan keputusan didefinisikan sebagai algoritma yang selalu berhenti dengan jawaban yang benar untuk pertanyaan ya/tidak.

Untuk logika proposisional, pertanyaan keputusan mengenai kepuasan, validitas, atau implikasi dapat diputuskan. Untuk logika temporal linier yang diperkenalkan dalam Bab 10, pertanyaan-pertanyaan ini juga dapat diputuskan. Untuk logika orde pertama, pertanyaan-pertanyaan ini umumnya tidak dapat diputuskan (lihat Bab 11). Namun, jika kita membatasi diri pada beberapa segmen logika orde pertama, pertanyaan-pertanyaan ini dapat menjadi dapat diputuskan. Misalnya, segmen-segmen logika orde pertama berikut ini dapat diputuskan.

- **Rumus orde pertama dengan model berukuran tertentu**

Dapat diputuskan untuk memeriksa apakah suatu himpunan rumus orde pertama memiliki model berukuran tertentu, karena rumus-rumus ini dapat dikonversi menjadi himpunan rumus dasar yang kepuasannya dapat diputuskan. Pada Bagian 7.5, kami memperkenalkan Mace4 sebagai alat untuk model berukuran terbatas. Pada Bagian 7.5, kami menunjukkan bagaimana rumus dasar dikonversi menjadi rumus proposisional yang setara.

- **Kalkulus predikat monadik**

Kalkulus predikat monadik adalah kasus khusus dari logika orde pertama (juga disebut kalkulus predikat) di mana aritas semua simbol predikat adalah satu (misalnya, mereka hanya mengambil satu argumen dan disebut monadik atau unari) dan aritas semua simbol fungsi adalah 0. Pada Bagian 5.2, kami menunjukkan bahwa predikat monadik mendefinisikan secara unik suatu himpunan dan merupakan interpretasi tipe dalam logika banyak jenis. Kalkulus predikat monadik berguna dalam studi relasi antar himpunan. Dapat diputuskan apakah suatu formula orde pertama monadik dengan kesamaan valid atau dapat dipenuhi. Namun, kami tidak mengetahui adanya alat otomatis yang memanfaatkan sifat ini. Jika kita hanya menambahkan satu predikat biner (selain kesamaan) atau satu fungsi biner, maka logika yang dihasilkan tidak dapat diputuskan.

- **Masalah keputusan klasik**

Terdapat klasifikasi lengkap kelas formula orde pertama yang dapat diputuskan berdasarkan bentuk kuantifikasi dari bentuk normal prenex (lihat Bagian 5.4) tanpa simbol fungsi dengan aritas > 0 : jika prefiks kuantifikasi berbentuk: (a) $\exists^* \forall^*$; (b) $\exists^* \forall \exists^*$; atau (c) $\exists^* \forall^2 \exists^*$, di mana $\exists^*$ berarti Anda dapat memiliki 0 atau lebih $\exists$, maka kepuasan

formula prenex ini dapat diputuskan. Di sisi lain, jika prefiks kuantifikasi berbentuk (d) $\forall^3\exists$ atau (e) $\forall\exists\forall$, maka terdapat formula prenex yang kepuasannya tidak dapat diputuskan. Perhatikan bahwa setiap bentuk normal prenex termasuk dalam salah satu dari (a)–(e). Rumus monadik dapat dikonversi menjadi rumus prenex ekuivalen berbentuk $\exists^*\forall^* F(class(a))$. Karena tidak ada alat penalaran yang didasarkan pada keterputusan masalah klasik ini, kita lewati topik ini di sini.

Dalam tren terkini pada prosedur pengambilan keputusan, pemilihan segmen logika orde pertama yang dapat diputuskan dimotivasi oleh verifikasi formal program seperti yang dibahas dalam Bab 9. Dalam praktik verifikasi formal, kita perlu menangani tiga kesulitan berikut dari rumus kompleks dalam penalaran otomatis:

1. Rumus dengan kuantifikasi

Dalam Bab 5, kami menunjukkan bahwa kuantifikasi dapat dihilangkan dengan proses yang disebut Skolemisasi. Di akhir Bab 9, kami juga menunjukkan cara menghindari kuantifikasi dalam invarian loop, dengan mengambil pendekatan teori domain dan menggunakan fungsi yang didefinisikan secara rekursif untuk mengganti rumus dengan kuantifikasi. Pendekatan ini mendefinisikan serangkaian fungsi dengan sifat khusus dan membutuhkan prosedur pengambilan keputusan untuk menangani fungsi-fungsi ini.

2. Rumus dengan satu atau lebih domain khusus

Bahasa pemrograman pada umumnya menyediakan tipe data yang telah ditentukan sebelumnya, seperti nilai Boolean, bilangan bulat 32-bit atau 64-bit, bilangan floating point, array, dan lain-lain. Semantik operasi pada tipe data tersebut menimbulkan masalah baru, seperti overflow atau tak terbatas, dan tidak lagi sesuai dengan teori matematika tentang tipe tak terbatas, misalnya, bilangan asli. Secara umum, kita menggunakan domain atau teori (singkatan dari teori domain) untuk merujuk pada tipe data khusus dan fungsi terkait. Untuk memverifikasi secara formal program yang ditulis dalam bahasa pemrograman tersebut, kita membutuhkan prosedur pengambilan keputusan yang menangani teori-teori ini secara efisien. Pendekatan ini disebut sebagai satisfiability modulo theories (SMT) dalam komunitas penalaran otomatis.

3. Rumus dengan kombinasi Boolean sembarang

Untuk menghindari penanganan kombinasi Boolean sembarang dalam teori yang berbeda, Nieuwenhuis, Oliveras, dan Tinelli menganjurkan kerangka kerja yang disebut DPLL(T), yang merupakan perluasan dari algoritma DPLL yang terkenal untuk logika proposisional dengan teori T . DPLL(T) menangani kombinasi Boolean dengan cara yang sama seperti algoritma DPLL dan memungkinkan prosedur pengambilan keputusan untuk berbagai teori T hanya menangani konjungsi literal. Fitur ini sangat mengurangi kompleksitas prosedur pengambilan keputusan untuk T .

Prosedur pengambilan keputusan untuk kepuasan modulo teori (SMT) terus menarik minat banyak aplikasi verifikasi. Pemecah masalah berdasarkan pendekatan SMT memeriksa kepuasan rumus orde pertama yang berisi fungsi dari berbagai domain seperti nilai Boolean, vektor bit, aritmatika bilangan bulat, dan array. Masalah verifikasi sering diterjemahkan ke

dalam rumus SMT, yang merupakan rumus orde pertama terbatas yang dapat diterima oleh pemecah masalah SMT. Pemecah SMT mengambil rumus-rumus ini dan mencoba untuk melaporkan kepuasan atau ketidakpuasan rumus tersebut.

Keuntungan utama pemecah SMT dibandingkan pemecah SAT murni adalah tingkat abstraksi yang lebih tinggi yang menghindari ledakan eksponensial pengkodean proposisional dari masalah yang sama. Dengan menggunakan rumus secara langsung dalam teori-teori seperti aritmatika, array, dan fungsi yang tidak diinterpretasikan, pemecah SMT menjanjikan kinerja yang lebih tinggi daripada pemecah SAT murni. Bahkan, pemecah SMT memiliki berbagai aplikasi dalam verifikasi perangkat keras dan perangkat lunak, pemeriksaan statis yang diperluas, penyelesaian kendala, perencanaan, penjadwalan, pembuatan kasus uji, dan keamanan komputer.

12.1 MEMPERLUAS DPLL DENGAN TEORI T

Sebagian besar pemecah SMT menggunakan kerangka kerja $DPLL(T)$, di mana T merupakan singkatan dari teori, termasuk kesamaan, aritmatika linier, array, dll. $DPLL(T)$ dinyatakan sebagai perluasan dari algoritma DPLL. Secara garis besar, $DPLL(T)$ bekerja dengan mengubah rumus SMT menjadi rumus SAT di mana rumus atomik digantikan oleh variabel proposisional. $DPLL(T)$ berulang kali menemukan model untuk rumus SAT dan berkonsultasi dengan pemecah teori untuk memeriksa konsistensi di bawah teori spesifik domain, hingga ditemukan model yang konsisten dengan teori, atau rumus SAT kehabisan model.

Abstraksi Proposisional

Dalam Contoh 9.2.8, kita perlu membuktikan validitas (2.1.1.2), yang di sini diberi label ulang sebagai X :

$$X : (I \wedge i < n \wedge \neg(m < A[i])) \rightarrow I[i \infty i + 1],$$

di mana I adalah invarian loop ($m \doteq arrMax(i, A) \wedge (i \leq n)$). Selain operator logika, rumus ini melibatkan aritmatika (misalnya, $+$, $<$, dan $\leq$), array, dan fungsi $arrMax$.

Langkah pertama SMT disebut abstraksi proposisional, yang menggantikan rumus atomik dari rumus orde pertama dengan variabel proposisional. Untuk rumus X , kita dapat memperkenalkan variabel proposisional berikut $p_1 - p_6$ untuk rumus atomik dalam X :

$$\begin{array}{ll} p_1 : m \doteq arrMax(i, A) & p_2 : i \leq n \\ p_3 : i < n & p_4 : m < A[i] \\ p_5 : m \doteq arrMax(i + 1, A) & p_6 : i + 1 \leq n \end{array}$$

sehingga abstraksi dari X adalah rumus SAT Y :

$$Y : (p_1 \wedge p_2 \wedge p_3 \wedge \neg p_4) \rightarrow (p_5 \wedge p_6).$$

Dalam logika proposisional, kita mengatakan bahwa formula Y adalah abstraksi dari X , yang dilambangkan dengan $Y = \gamma(X)$, jika setiap formula atomik yang berbeda dari X digantikan oleh variabel proposisional baru dan hasilnya adalah Y , yang unik hingga penamaan ulang variabel proposisional.

Secara umum, jika B adalah abstraksi dari formula A , yaitu, $B = \gamma(A)$, apa hubungan antara A dan B ? Pertama, A dan B tidak berbagi simbol predikat atau fungsi; oleh karena itu, kedua formula tersebut tidak berbagi model. Dengan demikian, tidak ada hubungan kesetaraan atau implikasi di antara keduanya, karena konsep kesetaraan dan implikasi didasarkan pada himpunan modelnya.

Selanjutnya, kami menyatakan bahwa A dan B tidak dapat dipenuhi secara setara, yaitu, $A \approx B$ adalah salah. Misalnya, misalkan A adalah $a \doteq b \wedge b \doteq c \wedge a \neq c$ yang tidak dapat dipenuhi oleh aksioma kesamaan.

Namun, $\gamma(A)$ adalah $p_1 \wedge p_2 \wedge \neg p_3$, yang dapat dipenuhi.

Dalam logika orde pertama, teori bebas kuantifikasi (QF) menunjukkan himpunan rumus orde pertama tanpa kuantifikasi. Rumus-rumus ini merupakan gabungan dari rumus dasar dan rumus yang mengandung variabel bebas. Arti dari variabel bebas ini bergantung pada aplikasi: semuanya dikuantifikasi secara universal atau semuanya dikuantifikasi secara eksistensial. Oleh karena itu, tidak perlu menulis kuantifikasi secara eksplisit dalam rumus-rumus ini. Misalnya, dalam rumus Skolemisasi, arti variabelnya dikuantifikasi secara universal. Untuk kesederhanaan, kita katakan variabel-variabel ini dikuantifikasi secara universal. Dalam kasus ini, rumus QF $F(x_1, \dots, x_n)$ setara dengan

$$\forall x_1, \dots, x_n F(x_1, \dots, x_n),$$

di mana F tidak mengandung kuantifikasi apa pun dan $free(F) = \{x_1, \dots, x_n\}$. Di sisi lain, dalam masalah penyatuan atau dalam aljabar sekolah menengah, variabel dalam himpunan persamaan dikuantifikasi secara eksistensial.

Dalam proses abstraksi, setiap variabel bebas dianggap sebagai "dasar": variabel-variabel ini memiliki kuantifikasi yang sama di p dan $\neg p$. Misalnya, jika p mewakili $q(x)$, maka $\neg p$ mewakili $\neg q(x)$. Seandainya kita memberikan kuantifikasi x di p sebagai $p \equiv \forall x q(x)$, kita akan memiliki $\neg p \equiv \exists x \neg q(x)$; oleh karena itu, kuantifikasi x akan berbeda di p dan $\neg p$.

Proposisi 12.1.1 Untuk setiap rumus QF A , jika A dapat dipenuhi, maka $\gamma(A)$ juga dapat dipenuhi.

Bukti Misalkan $B = \gamma(A)$, abstraksi dari A . Untuk setiap formula atomik A_i dari A , misalkan $p_i = \gamma(A_i)$ adalah variabel proposisional yang menggantikan A_i dalam abstraksi. Jika A dapat dipenuhi, maka A memiliki model orde pertama I sedemikian sehingga $I(A) = 1$. Ada tiga kasus yang perlu dipertimbangkan:

- A adalah dasar: Kita mendefinisikan interpretasi proposisional σ untuk B sebagai berikut: $\sigma(p_i) = I(A_i)$, di mana $p_i = \gamma(A_i)$. Mudah untuk memeriksa bahwa σ adalah

model dari B dengan membandingkan pohon formula A dan B di bawah σ dan σ' , masing-masing.

- Variabel bebas A dikuantifikasi secara eksistensial: Misalkan $free(A) = \{x_1, \dots, x_n\}$, maka $I(A) = 1$ berarti $I(A, \theta) = 1$, untuk beberapa penugasan $\theta = \{x_i \sim d_i : 1 \leq i \leq n\}$, di mana d_i adalah elemen domain dari I . Dalam hal ini, kita mendefinisikan interpretasi proposisional σ untuk B sebagai berikut: $\sigma(p_i) = I(A_i, \theta)$, di mana $p_i = \gamma(A_i)$. Mudah untuk memeriksa bahwa σ adalah model dari B seperti pada kasus ketika A adalah himpunan dasar.
- Variabel bebas A dikuantifikasi secara universal: Misalkan $free(A) = \{x_1, \dots, x_n\}$, maka $I(A) = 1$ berarti $I(A, \theta) = 1$ untuk setiap penugasan $\theta = \{x_i \sim d_i : 1 \leq i \leq n\}$, di mana d_i adalah elemen domain apa pun dari I . Kita hanya membutuhkan salah satu penugasan untuk mendefinisikan model proposisional σ untuk B seperti pada kasus sebelumnya.

Karena kita telah menghabiskan semua kasus A , proposisi tersebut harus benar.

Ada alasan sederhana untuk kondisi yang diperlukan bahwa A bebas kuantifikasi dalam proposisi di atas: jika A mengandung kuantifikasi, maka kuantifikasi ini dalam $\gamma(A)$ menjadi tidak bermakna. Proposisi ini dapat diperluas ke rumus prenex, di mana semua kuantifikasi berada di atas operator logika lainnya, dan kita dapat menyimpan kuantifikasi untuk variabel sebelum abstraksi.

Proposisi di atas masih berlaku ketika hanya sebagian dari rumus atomik A digantikan oleh variabel proposisional. Proposisi ini juga menyiratkan bahwa, jika $\neg A$ dapat dipenuhi, maka $\neg B$ juga dapat dipenuhi. Ini adalah bukti lain bahwa A tidak menyiratkan B : jika $A \models B$, kita akan memiliki $\neg B \models \neg A$. Dari $\neg A \models \neg B$, kita akan menyimpulkan bahwa $A \equiv B$, sebuah kontradiksi.

Akibat 12.1.2 Misalkan B adalah abstraksi dari A , yaitu, $B = \gamma(A)$, maka

- Jika B tidak dapat dipenuhi, maka A juga tidak dapat dipenuhi.
- Jika B valid dan A adalah dasar, maka A valid.

Bukti

- Dari Proposisi 12.1.1, jika A dapat dipenuhi, maka B dapat dipenuhi. Jadi, jika B tidak dapat dipenuhi, maka A tidak dapat dipenuhi (kontraposisif dari implikasi).
- Misalkan $X = \neg A$ dan $Y = \neg B$, maka mudah untuk melihat bahwa $Y = \gamma(X)$. Berdasarkan (a), jika Y tidak dapat dipenuhi, maka X juga tidak dapat dipenuhi. Namun, Y tidak dapat dipenuhi jika dan hanya jika B valid, dan X tidak dapat dipenuhi jika dan hanya jika dasar A valid. Jadi, jika B valid, maka A juga valid.

Biasanya, dalam verifikasi formal, kita ingin menunjukkan bahwa A valid dengan menunjukkan bahwa $\neg A$ tidak dapat dipenuhi. Jika $\neg A$ dapat dipenuhi, maka model dari $\neg A$ adalah contoh tandingan dari validitas A . Untuk tujuan ini, korolari di atas jarang berguna, karena jarang abstrak dari $\neg A$ tidak dapat dipenuhi. Penggunaan umum abstraksi adalah sebagai berikut: misalkan X adalah rumus QF yang diturunkan dari $\neg A$. Untuk menunjukkan bahwa X tidak

dapat dipenuhi, misalkan Y adalah abstrak dari X . Jika Y tidak dapat dipenuhi, kita selesai; Jika tidak, maka untuk setiap model Y , kita harus memeriksa apakah konjungsi literal dari X yang sesuai dengan model Y memang merupakan model X . Jika ya, maka X dapat dipenuhi dan ditemukan contoh tandingan dari A ; jika X tidak memiliki model seperti itu, maka X tidak dapat dipenuhi dan A valid.

Contoh DPLL(T)

Untuk kerangka kerja DPLL(T), abstraksi $\gamma(A)$ memainkan dua peran: (1) Sebagai filter model A sehingga kita membuang lebih sedikit upaya untuk memeriksa apakah A memiliki model. (2) Kita hanya perlu memeriksa apakah konjungsi literal dari A memiliki model atau tidak.

Contoh 12.1.3 Misalkan X adalah $(a \doteq b \vee b \doteq a) \wedge b \doteq c \wedge a \neq c$ dan $Y = \gamma(X) = (p_1 \vee p_2) \wedge p_3 \wedge \neg p_4$. Y memiliki tiga model: $\{p_1, p_2, p_3, \neg p_4\}$, $\{p_1, \neg p_2, p_3, \neg p_4\}$, dan $\{\neg p_1, p_2, p_3, \neg p_4\}$. Konjungsi rumus atomik yang sesuai dengan $\{p_1, p_2, p_3, \neg p_4\}$ adalah $a \doteq b \wedge b \doteq a \wedge b \doteq c \wedge a \neq c$, yang tidak dapat dipenuhi.

Demikian pula, rumus SMT yang sesuai dengan dua model Y lainnya juga tidak dapat dipenuhi. Dengan demikian, X tidak dapat dipenuhi.

Sekarang, kembali ke contoh verifikasi di awal bagian ini, misalkan A adalah $\neg X$ dan B adalah $\neg Y$ dalam contoh tersebut. Artinya, B adalah $\neg((p_1 \wedge p_2 \wedge p_3 \wedge \neg p_4) \rightarrow (p_5 \wedge p_6))$, di mana

$$\begin{array}{ll} p_1 : m \doteq \text{arrMax}(i, A) & p_2 : i \leq n \\ p_3 : i < n & p_4 : m < A[i] \\ p_5 : m \doteq \text{arrMax}(i + 1, A) & p_6 : i + 1 \leq n \end{array}$$

Jika B tidak dapat dipenuhi, maka A juga tidak dapat dipenuhi (yaitu, X valid) berdasarkan Proposisi 12.1.1. Jika B dapat dipenuhi, kita harus menunjukkan bahwa setiap model B mengarah pada kontradiksi A dalam teori aritmatika, array, dan arrMax . Dengan kata lain, $\neg X$ tidak memiliki model; oleh karena itu, X harus valid. Sekarang, kita mengubah B menjadi himpunan S yang terdiri dari lima klausa:

$$S = \{p_1, p_2, p_3, \overline{p_4}, (\overline{p_5} \mid \overline{p_6})\}$$

S memiliki tiga model di mana p_1, p_2 , dan p_3 bernilai benar, p_4 bernilai salah, dan salah satu dari p_5 dan p_6 harus bernilai salah. Dalam aritmatika bilangan bulat, p_3 dan p_6 , yaitu $i < n$ dan $i + 1 \leq n$, adalah ekuivalen. Dengan demikian, p_6 harus bernilai benar ketika p_3 bernilai benar. Selain itu, $p_1 \wedge \neg p_4 \rightarrow p_5$ bernilai benar berdasarkan definisi arrMax dan max :

$$\text{arrMax}(i + 1, A) = \max(\text{arrMax}(i, A), A[i]) \quad \text{definition of arrMax}$$

$$\begin{aligned}
&= \max(m, A[i]) & p_1 : m &\doteq \text{arrMax}(i, A) \\
&= m & \neg p_4 : \neg(m < A[i]), \\
& & \text{definition of max}
\end{aligned}$$

Dengan demikian, p_5 harus benar ketika p_1 benar dan p_4 salah. Dengan kata lain, tidak ada model S yang dapat diterima untuk A , jadi, $A = \neg X$ tidak dapat dipenuhi.

Pada Bab 4, kami telah menyajikan algoritma DPLL secara detail dan menjelaskan beberapa teknik untuk implementasi DPLL yang efisien, yang merupakan metode pencarian berbasis enumerasi untuk kepuasan proposisional. DPLL mencoba menemukan model untuk sekumpulan klausa proposisional. Di sini, model dari DPLL dapat dilihat sebagai sekumpulan literal yang benar dalam model tersebut.

Contoh 12.1.4 Pertimbangkan masalah menentukan kepuasan dari rumus orde pertama X berikut dalam bentuk normal konjungtif:

$$X: g(a) \doteq c \wedge (f(g(a)) \neq f(c) \vee g(a) \doteq d) \wedge c \neq d$$

Langkah pertama $DPLL(T)$ adalah abstraksi, di mana T mewakili kesamaan: gunakan variabel proposisional untuk setiap rumus atomik dan dapatkan himpunan klausa proposisional untuk pemecah SAT. Artinya, dengan p_1 untuk $g(a) \doteq c$, p_2 untuk $f(g(a)) = f(c)$, p_3 untuk $g(a) = c$, dan p_4 untuk $c = d$, kita memperoleh klausa proposisional berikut:

$$\{p_1, \neg p_2 \vee p_3, \neg p_4\}$$

$DPLL(T)$ akan memasukkan klausa-klausa ini ke dalam pemecah SAT yang mengembalikan $\{p_1, \neg p_2, \neg p_4\}$ sebagai model pertamanya. Rumus orde pertama yang sesuai dengan model ini adalah

$$g(a) \doteq c \wedge f(g(a)) \neq f(c) \wedge c \neq d$$

yang dapat dibuktikan tidak dapat dipenuhi dengan prosedur pengambilan keputusan untuk kesamaan. $DPLL(T)$ dapat menambahkan negasi model sebagai klausa baru ($\neg p_1 \vee p_2 \vee p_4$) ke dalam pemecah SAT untuk meminta model berikutnya. Pemecah SAT akan mengembalikan $\{p_1, p_3, \neg p_4\}$ sebagai model keduanya. Rumus orde pertama yang sesuai dengan model ini adalah

$$g(a) \doteq c \wedge g(a) \doteq d \wedge c \neq d$$

yang juga dapat dibuktikan tidak dapat dipenuhi. $DPLL(T)$ kemudian menambahkan ($\neg p_1 \vee \neg p_3 \vee p_4$) ke pemecah SAT, yang akan mengembalikan "tidak dapat dipenuhi." Akhirnya,

DPLL(T) mengklaim bahwa rumus asli X tidak dapat dipenuhi karena pemecah SAT kehabisan model.

Contoh di atas mengilustrasikan keuntungan besar dari kerangka kerja DPLL(T): algoritma DPLL dasar menangani operator logika, dan prosedur keputusan untuk teori T menangani konjungsi literal, sehingga mengurangi kompleksitas prosedur keputusan yang dibutuhkan oleh teori-teori tersebut. Bahkan, banyak teori yang menarik memiliki prosedur keputusan yang efisien untuk kepuasan himpunan (atau konjungsi) literal.

Mari kita sebut prosedur keputusan yang digunakan dalam DPLL(T) sebagai pemecah teori untuk T . Pemecah teori ini menggunakan berbagai algoritma untuk setiap domain yang muncul dalam input. Contoh di atas juga mengilustrasikan beberapa aspek menarik dari DPLL(T).

1. Rumus atomik dalam rumus masukan diabstraksikan ke variabel proposisional. Abstraksi ini dilakukan sekali untuk selamanya sebagai langkah pra-pemrosesan.
2. Pemecah SAT dipanggil beberapa kali, setiap kali dengan penambahan klausa masukan untuk menghindari model yang berulang. Dengan demikian, pemecah SAT inkremental diinginkan untuk mendukung DPLL(T).
3. Pemecah teori dipanggil beberapa kali, masing-masing dengan konjungsi literal sebagai masukan. Setiap masukan sesuai dengan model proposisional yang ditemukan oleh pemecah SAT. Pemecah teori tidak perlu inkremental karena masukan berubah secara tidak terduga. Artinya, interaksi antara pemecah SAT dan pemecah teori berlangsung bolak-balik dalam DPLL(T).
4. Pendekatan pemecahan kepuasan teori melalui kepuasan proposisional ini dikatakan malas: informasi domain digunakan secara malas ketika memeriksa konsistensi model proposisional dalam teori T . Sebaliknya, pendekatan *eager* mengubah masalah kepuasan teori asli menjadi contoh proposisional yang ekuisatisfiable.
5. Dalam pendekatan lazy ini, setiap alat melakukan apa yang dikuasainya:
 - Pemecah SAT menangani kombinasi Boolean.
 - Pemecah teori menangani konjungsi literal dalam teori.

DPLL(X): Algoritma DPLL(T)

Kerangka kerja DPLL(T) dapat dengan mudah diimplementasikan jika kita memiliki implementasi algoritma DPLL dan prosedur keputusan untuk memeriksa konsistensi himpunan literal dalam suatu teori. Algoritma DPLL T berikut adalah modifikasi kecil dari Algoritma 4.2.5, yaitu DPLL(C) dengan CDCL (pembelajaran klausa berbasis konflik) untuk SAT.

Algoritma 12.1.5 $DPLL_T(X)$ menerima input berupa himpunan X dari klausa orde pertama dan mengembalikan nilai true jika X dapat dipenuhi dan mengembalikan nilai false jika tidak.

proc $DPLL_T(X)$: **Boolean**

```

0   $C := abstraction(X)$  //  $C$  is an abstract of  $X$ .
1   $U := initialize(C)$  // initialize data structure for  $C$ 
    $U$ : unit clauses.
2   $level := 0$ ;  $\sigma := \{\}$  //  $\sigma$  is a partial model of  $C$ .
```

```

3  while (true) do
4      res := BCP(U) // BCP(U) extends  $\sigma$  .
5      if (res = "SAT") // res contains a conflicting clause.
6          if (level = 0) return false // C as well as X is unsatisfiable
7          U := insertNewClause(conflictAnalysis(res)) // new level is set
8          undo() // undo to the new level
9      else
10         A := pickLiteral() // pick an unassigned literal as decision literal.
11         if (A = nil) // a model of C is found.
12             if isTheoryConsistent( $\sigma$ ) return  $\top$  // X is satisfiable.
13             else U := insertNewClause(negateDecisions( $\sigma$ ))
14         else U := {A}; level := level + 1

```

Satu-satunya perbedaan antara DPLL_T dan Algo. 4.2.5 adalah sebagai berikut:

- Pada baris 0: *abstraction*(*X*) menghitung abstraksi dari *X*.
- Pada baris 11–13: Alih-alih mengembalikan *true* setelah (*A* = *nil*) (setiap literal memiliki nilai kebenaran), kita memanggil *isTheoryConsistent*(σ) untuk memeriksa apakah formula SMT yang sesuai dengan model proposisional σ dari *C* konsisten dalam teori-teori *X*. Jika ya, kembalikan *true*; jika tidak, buat klausa baru dari negasi literal keputusan dalam σ : *negateDecisions*(σ) mengembalikan negasi dari konjungsi literal keputusan dalam σ . Kemudian, *insertNewClause* dipanggil pada klausa yang baru dibuat (lihat baris 7).

Implementasi *abstraction*(*X*) cukup sederhana. Tujuan baris 13 adalah untuk mendukung CDCL (pembelajaran klausa yang diarahkan konflik).

Algoritma *DPLL_T*(*X*) hanya mengilustrasikan betapa mudahnya kita dapat mengimplementasikan kerangka kerja *DPLL*(*T*) jika kita memiliki pemecah SAT dan implementasi *isTheoryConsistent*(σ). Yang terakhir bergantung pada teori-teori yang muncul di *X* dan merupakan fokus dari bagian-bagian selanjutnya.

Contoh 12.1.6 Misalkan *T* adalah kesamaan dan *X* berisi klausa-klausa berikut:

$$(b \doteq c), (h(a) \doteq h(c)) \vee p, (a \doteq b) \vee \neg p \vee (a \doteq d), (a \doteq b) \vee (a \neq d)$$

Prosedur abstraksi akan mengubah *X* menjadi *C* yang berisi klausa-klausa berikut:

$$(s), \quad (t \vee p), \quad (q \vee \neg p \vee r), \quad (q \vee \neg r)$$

di mana *q* mewakili $a \doteq b$ mewakili $a \doteq d$ mewakili $b \doteq c$ dan *t* mewakili $h(a) = h(c)$.

Setelah baris 1, $U = \{s\}$. Pada baris 4, *BCP*(*U*) akan membuat *s* menjadi benar, dan $\sigma = \{s\}$.

Misalkan *pickLiteral* memilih $\neg t$ menjadi benar dan $U = \{s, \neg t\}$. *BCP*(*U*) akan membuat *p* menjadi benar karena $(t \vee p)$. Sekarang, $\sigma = \{s, \neg t, p\}$.

Selanjutnya, misalkan *pickLiteral* memilih $\neg r$ menjadi benar dan $U = \{\neg r\}$. $BCP(U)$ akan membuat q menjadi benar karena $(q \vee \neg p \vee r)$. Sekarang $\sigma = \{p, q, \neg r, s, \neg t\}$, yang merupakan model C dan $isTheoryConsistent(\sigma)$ disebut. Rumus SMT yang sesuai untuk σ adalah

$$p \wedge a \doteq b \wedge a = d \wedge b \doteq c \wedge h(a) \neq h(c)$$

Dari $a \doteq b$ dan $b \doteq c$, $a \doteq c$ adalah benar dan $h(a) \neq h(c)$ adalah salah berdasarkan aksioma kesamaan. Dengan demikian, $isTheoryConsistent(\sigma)$ mengembalikan $\perp$, dan klausa baru dari negasi konjungsi dari dua literal keputusan adalah $(t \vee r)$.

Prosedur *insertNewClause*($t \vee r$) akan menambahkan klausa ini dan melakukan backtracking ke level 1 dengan $U = \{r\}$. $BCP(U)$ akan membuat r dan q menjadi benar dengan model kedua $\sigma = \{s, \neg t, p, r, q\}$, yang rumus SMT-nya adalah

$$p \wedge a \doteq b \wedge a \doteq d \wedge b \doteq c \wedge h(a) \neq h(c)$$

Sekali lagi, $isTheoryConsistent(\sigma)$ menemukan formula ini tidak konsisten dan mengembalikan false. Klausa baru dari negasi literal keputusan adalah (t) . Sekarang, DPLLT akan melakukan backtracking ke level 0 dan menemukan model $\sigma = \{p, q, r, s, t\}$ (atau $\sigma = \{p, q, \neg r, s, t\}, \dots$) dan mengembalikan true pada baris 12 karena $isTheoryConsistent(\sigma)$ akan mengembalikan true.

Tentu saja, modifikasi minimal DPLL menjadi DPLLT tidak selalu memberi kita implementasi DPLL(T) yang ideal. Di akhir bab ini, kita akan membahas beberapa masalah untuk membuat DPLLT lebih praktis dan efisien.

Selanjutnya, kita akan menyajikan prosedur pengambilan keputusan untuk teori-teori berikut, yang sering muncul dalam verifikasi program dan dapat diintegrasikan dalam DPLLT:

- Kesamaan dengan fungsi yang tidak diinterpretasikan
- Aritmatika linear

12.2 KESAMAAN DENGAN FUNGSI YANG TIDAK DIINTERPRETASIKAN

Dalam beberapa aplikasi, akan lebih mudah dan efisien untuk mengingat definisi beberapa fungsi. Misalnya, dalam verifikasi program, $1 + 3$ dapat diganti dengan 4 karena $+$ adalah penjumlahan bilangan bulat. Dalam logika persamaan, predikat kesamaan “ $\doteq$ ” diasumsikan memenuhi aksioma kesamaan (Bagian 7.1). Fungsi seperti $+$ dan dengan makna yang dimaksudkan disebut “fungsi yang diinterpretasikan.” Di sisi lain, fungsi yang dapat mengambil interpretasi apa pun disebut “fungsi yang tidak diinterpretasikan.”

Fungsi yang Tidak Diinterpretasikan

Dalam logika orde pertama, simbol fungsi diasumsikan tidak diinterpretasikan dan dapat diinterpretasikan sebagai fungsi apa pun dengan aritas yang sama. Dengan kata lain, secara default, setiap simbol fungsi adalah fungsi yang tidak diinterpretasikan.

Mengganti fungsi yang diinterpretasikan dengan fungsi yang tidak diinterpretasikan dalam suatu rumus adalah cara generalisasi (atau abstraksi) dan dapat berguna dalam memperoleh hasil yang lebih kuat atau menyederhanakan pembuktian.

Contoh 12.2.1 Kita ingin membuktikan kesetaraan program P_1 dan P_2 :

P_1	P_2
<pre> int a[1..3] int i := 2, x := a[3] while (i > 0) x := a[i] + x; i := i - 1 return x </pre>	<pre> int a[1..3] int y y := a[1] + (a[2] + a[3]) return y </pre>

Karena badan loop while di P_1 dieksekusi dua kali, kita dapat menguraikan loop dengan memperkenalkan variabel bantu x_0, x_1, x_2 untuk mencatat nilai x sebelum dan sesudah $x := a[i] + x$. Hubungan antara keduanya dapat dinyatakan dengan

$$x_0 \doteq a[3] \wedge x_1 \doteq a[2] + x_0 \wedge x_2 \doteq a[1] + x_1$$

Bersama dengan $y \doteq a[1] + (a[2] + a[3])$, kami ingin membuktikan $x_2 \doteq y$.

Jika $+$ adalah fungsi yang diinterpretasikan, akan sulit untuk menentukan nilai $a[2] + a[3], x_2$, dan y . Jika kita mengganti $+$ dengan fungsi f yang tidak diinterpretasikan, verifikasi menjadi pembuktian teorema berikut:

$$(x_0 \doteq a[3] \wedge x_1 \doteq f(a[2], x_0) \wedge x_2 \doteq f(a[1], x_1) \wedge y \doteq f(a[1], f(a[2], a[3]))) \rightarrow x_2 \doteq y.$$

Hal di atas dapat dibuktikan kebenarannya dengan menulis ulang, dengan melihat setiap kesamaan sebagai aturan penulisan ulang dari kiri ke kanan. Setelah teorema terbukti, kesetaraan P_1 dan P_2 berlaku untuk setiap operasi biner f pada setiap tipe $a[1..3]$ (misalnya, double atau matriks).

Di sisi lain, jika rumus umum tidak dapat dibuktikan, bukan berarti kesetaraan program tidak berlaku. Misalnya, dalam P_2 , jika kita memiliki $y := (a[1] + (a[2] + a[3]))$, maka kedua program tersebut masih setara karena $+$ bersifat asosiatif. Namun, $f(a[1], f(a[2] + a[3])) \doteq f(f(a[1], a[2]), a[3])$ tidak lagi benar, kecuali jika f diketahui bersifat asosiatif.

Kami ingin menunjukkan bahwa x dan y yang digunakan dalam P_1 dan P_2 adalah variabel kode; bukan variabel logika orde pertama. Rumus untuk kesetaraan P_1 dan P_2 memang mendasar (yaitu, bebas variabel) karena x dan y adalah fungsi 0-arity dalam logika orde pertama. Untuk menghindari kebingungan, kami menyebut fungsi 0-arity ini sebagai pengidentifikasi.

Masalah Penutupan Kongruensi

Ingatlah bahwa aksioma kesamaan yang diperkenalkan dalam Bab 7 adalah lima rumus berikut, di mana variabel diasumsikan terkuantifikasi secara universal:

reflexivity :	$x \doteq x;$
symmetry :	$(x \doteq y) \rightarrow (y \doteq x);$
transitivity :	$(x \doteq y) \wedge (y \doteq z) \rightarrow (x \doteq z);$
functionmonotonicity :	For $1 \leq i \leq k, (x_i \doteq y_i) \rightarrow f(x_1, \dots, x_k) \doteq f(y_1, \dots, y_k);$
predicatemonotonicity :	For $1 \leq i \leq k, x_i \doteq y_i \wedge p(x_1, \dots, x_k) \rightarrow p(y_1, \dots, y_k).$

Himpunan literal A disebut kesamaan dasar jika satu-satunya predikat dalam A adalah " $\doteq$ " dan tidak ada literal yang mengandung variabel. Kita sering menulis $\neg(s \doteq t)$ sebagai $s \neq t$. Literal positif $s \doteq t$ disebut persamaan dan literal negatif $s \neq t$ disebut ketidaksetaraan. Jika konstanta c_1 dan c_2 muncul dalam A , kita mengasumsikan bahwa $(c_1 \neq c_2) \in A$, sehingga kita tidak membedakan pengidentifikasi (yaitu, fungsi dengan aritas nol) dan konstanta. Jika predikat selain $\doteq$ diperlukan, kita memperlakukannya sebagai fungsi: ganti literal positif $p(x)$ dengan $p(x) \doteq \top$ dan literal negatif $\neg p(x)$ dengan $p(x) \doteq \perp$, dan tambahkan $\top \neq \perp$ ke dalam A .

Diberikan himpunan kesamaan dasar A , apakah A dapat dipenuhi dengan aksioma kesamaan? Misalnya, dalam Contoh 12.2.1, misalkan A adalah

$$\{x_0 \doteq a[3], x_1 \doteq f(a[2], x_0), x_2 \doteq f(a[1], x_1), \\ y \doteq f(a[1], f(a[2], a[3])), x_2 \neq y\},$$

Maka A tidak dapat dipenuhi jika dan hanya jika $x_2 \doteq y$. merupakan konsekuensi logis dari empat persamaan pertama.

Masalah keputusan di atas secara tradisional disebut penutupan kongruensi kesamaan dengan fungsi yang tidak diinterpretasikan. Karena suatu fungsi tidak diinterpretasikan secara default dalam logika orde pertama, kami lebih suka menyebutnya penutupan kongruensi kesamaan dasar.

$\doteq$ Beberapa orang juga menyebutnya penutupan kongruensi kesamaan bebas kuantifikasi. Himpunan rumus bebas kuantifikasi adalah gabungan dari rumus dasar dan rumus dengan variabel bebas. Untuk masalah penutupan kongruensi yang dibahas, tidak ada variabel bebas yang disajikan. Dengan demikian, "dasar" adalah label yang lebih baik daripada "bebas kuantifikasi" untuk masalah penutupan kongruensi.

Pendekatan konvensional, yang diusulkan oleh Shostak, Nelson, dan Oppen, terdiri dari langkah-langkah berikut:

1. Bagi himpunan A dari kesamaan dasar menjadi persamaan E (literal positif) dan ketidaksetaraan D (literal negatif).
2. Hitung penutupan kekongruan $=_E$, yang merupakan penutupan kekongruan yang dihasilkan oleh E (Definisi 7.1.7). Ini adalah relasi kesetaraan minimum yang memenuhi aksioma monotonisitas.

3. Jika ada $(s \neq t) \in D$ sedemikian sehingga $s =_E t$, maka kembalikan “tidak dapat dipenuhi”; jika tidak, kembalikan “dapat dipenuhi.”

Keabsahan pendekatan di atas dinyatakan sebagai teorema berikut:

Teorema 12.2.2 Diberikan himpunan A dari kesamaan dasar, termasuk persamaan dan ketidaksamaan, misalkan $E \subseteq A$ adalah himpunan semua persamaan dalam A . Jika ada $(s \neq t) \in A$ sedemikian sehingga $s =_E t$, maka A tidak dapat dipenuhi; jika tidak, A dapat dipenuhi.

Pembuktiannya dapat dilakukan dengan model Herbrand (Proposisi 7.1.3) dan diserahkan sebagai latihan. Sekarang, kita siap untuk pembuktian teorema berikut:

Teorema 12.2.3 Penutupan kekongruenan dari kesamaan dasar dapat diputuskan.

Bukti Diberikan himpunan A dari kesamaan dasar, termasuk persamaan dan ketidaksamaan, misalkan $E \subseteq A$ adalah himpunan semua persamaan dalam A . Menurut Teorema 7.2.22, ada prosedur keputusan untuk $=_E$. Untuk setiap $s \neq t \in A$, kita periksa apakah $s =_E t$: jika ya, kembalikan “tidak dapat dipenuhi.” Jika setiap pemeriksaan mengatakan “tidak,” kembalikan “dapat dipenuhi.” Teorema 12.2.2 memastikan kebenaran jawabannya.

Contoh 7.2.21 menunjukkan bagaimana $A = \{f(a) \neq a, f^3a \doteq a, f^5a \doteq a\}$, di mana f^3a adalah $f(f(f(a)))$ dan f^5a adalah $f(f(f(f(f(a)))))$, tidak dapat dipenuhi.

Algoritma NOS

Kongruensi adalah relasi ekuivalensi yang memenuhi aksioma monotonisitas. Untuk menyimpan relasi ekuivalensi dalam komputer, struktur data terbaik adalah himpunan: elemen-elemen yang setara disimpan dalam himpunan yang sama sehingga sifat refleksif, simetris, dan transitif mudah dipertahankan. Setiap himpunan mewakili kelas yang setara, dan kumpulan kelas yang setara merupakan partisi dari semua elemen.

Algoritma penutupan kongruensi didasarkan pada ide ini: untuk menghitung $=_t$ untuk himpunan E persamaan dasar

1. Tempatkan setiap subistilah dari E ke dalam kelas kongruensinya sendiri.
2. Untuk setiap $s \doteq t \in E$, jika s dan t tidak berada dalam kelas yang sama, operasi penggabungan akan (i) menggabungkan kedua kelas yang berisi s dan t dan (ii) menyebarkan $s \doteq t$ berdasarkan aksioma monotonisitas. Artinya, kita mencoba menggabungkan istilah-istilah seperti $f(s, x)$ dan $f(t, x)$ ke dalam satu kelas jika istilah-istilah tersebut muncul dalam input.

Ide ini dikreditkan kepada Nelson dan Oppen serta Shostak; kedua tim tersebut secara independen menerbitkan algoritma serupa sekitar waktu yang sama. Kita akan menyebut algoritma yang didasarkan pada ide ini sebagai algoritma NOS, di mana NOS berasal dari huruf pertama Nelson, Oppen, dan Shostak.

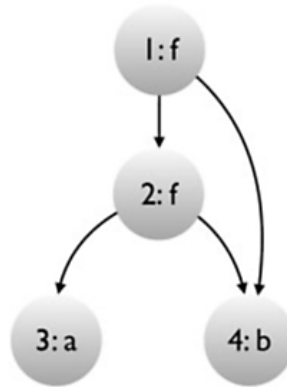
Penyebaran $s \doteq t$ melalui aksioma monotonitas dilakukan sebagai berikut: jika s muncul dalam $f(\dots, s, \dots)$ dan t muncul dalam $f(\dots, t, \dots)$, maka algoritma NOS akan memastikan bahwa $f(\dots, s, \dots)$ dan $f(\dots, t, \dots)$ harus berada dalam kelas yang sama dengan memeriksa apakah argumennya setara secara berpasangan.

Algoritma NOS menggunakan grafik istilah dan struktur data Union-Find untuk himpunan, dengan cara yang sangat mirip dengan algoritma unifikasi (Algoritma 6.1.19).

Algoritma $NOS(E)$ mengambil himpunan E persamaan dasar sebagai masukan dan membuat grafik istilah $G = (V, E)$ dari semua istilah dalam E , berbagi subistilah (setidaknya berbagi simpul pengidentifikasi untuk semua kemunculan pengidentifikasi). Jika himpunan D ketidaksetaraan diberikan, istilah-istilah dalam D juga ditambahkan ke dalam G . $G = (V, E)$ memiliki sifat-sifat berikut:

1. G adalah grafik asiklik terarah (DAG) dengan kemungkinan beberapa sisi antara dua simpul.
2. Setiap simpul $v \in V$ memiliki label, $label(v)$, yang merupakan simbol fungsi dari E .
3. Jika $label(v) = f$ dan $aritas(f) = k$, maka v memiliki k penerus terurut dalam G . Penerus ke- i dapat diakses oleh $child(v, i)$. Oleh karena itu, jika f adalah pengidentifikasi ($aritas(f) = 0$), v tidak memiliki penerus.
4. Setiap subistilah t dari E diwakili oleh sebuah simpul unik $v_t \in V$ sedemikian sehingga (a) t adalah sebuah pengenalan dan $label(v_t) = t$; atau (b) jika $t = f(t_1, \dots, t_k)$, maka $label(v_t) = f$ dan v_t memiliki k penerus $v_{t_1}, \dots, v_{t_k}$, sedemikian sehingga setiap v_{t_i} mewakili t_i untuk $1 \leq i \leq k$.
5. Untuk mengimplementasikan algoritma Union-Find, kita menambahkan struktur data berikut ke setiap simpul $v \in V$. Akibatnya, v memiliki makna ganda: v adalah simpul dari grafik istilah G dan simpul dari pohon yang digunakan oleh algoritma Union-Find untuk konstruksi kelas yang setara:
 - $up(v)$ mendefinisikan induk dari v dalam sebuah pohon, yang juga mewakili relasi kesetaraan di antara simpul-simpulnya. Awalnya, $up(v) = v$. Artinya, v adalah akar, dan setiap simpul awalnya berada dalam kelas kesetaraannya sendiri.
 - Jika v adalah akar, $w(v)$, yang disebut bobot pohon, mencatat jumlah simpul dalam pohon. Awalnya, $w(v) = 1$, yaitu, pohon tersebut adalah simpul tunggal.
 - Untuk mengimplementasikan propagasi kongruensi, kita menambahkan daftar ke setiap simpul $v \in V$: $use(v)$ mencatat daftar semua pendahulu (juga disebut induk) dari v dalam G ; ini tidak boleh dikacaukan dengan $up(v)$, yang merupakan induk dari v dalam pohon.

$$f(a, b) = a \wedge f(f(a, b), b) \neq a$$



Gambar 12.1 DAG $G = (V, E)$ untuk $A = \{f(a, b) \doteq a, f(f(a, b), b) \neq a\}$,
di mana $V = \{1, 2, 3, 4\}$,
 $use(1) = \emptyset$, $use(2) = \{1\}$, $use(3) = \{2\}$, $use(4) = \{1, 2\}$

Contoh 12.2.4 Misalkan $A = \{f(a, b) \doteq a, f(f(a, b), b) \neq a\}$. DAG untuk suku-suku A ditunjukkan pada Gambar 12.1.

Setelah DAG G dibuat, untuk setiap persamaan $s \doteq t$, kita akan memanggil $merge(v_s, v_t)$, di mana v_s dan v_t adalah simpul-simpul G yang mewakili s dan t .

```

proc merge( $v_1, v_2$ )
1    $s_1 := Find(v_1); s_2 := Find(v_2)$ 
2   if  $s_1 = s_2$  return // already in the same class, exit
3    $P_1 := use(s_1); P_2 := use(s_2)$ 
4    $Union(s_1, s_2)$  // Union two classes into one
5   for  $(t_1, t_2) \in P_1 \times P_2$  do // Congruence propagation
6     if  $Find(t_1) \neq Find(t_2) \wedge congruent(t_1, t_2)$ 
7        $merge(t_1, t_2)$ 
  
```

Kode semu untuk operasi kongruen, gabungan, dan pencarian diberikan di bawah ini.

```

proc congruent( $t_1, t_2$ ) : Boolean
1   if  $label(t_1) \neq label(t_2)$  return false
2   for  $i := 1$  to  $arity(label(t_1))$  do
3     if  $Find(child(t_1, i)) \neq Find(child(t_2, i))$  return false
4   return true
  
```

```

proc Union( $s_1, s_2$ )
  // Initially, for any node  $s$ ,  $w(s) = 1$ , the weight of a singleton tree
1   if  $w(s_1) < w(s_2)$   $x := s_1; s_1 := s_2; s_2 := x$  // switch  $s_1$  and  $s_2$ 
2    $up(s_2) := s_1$  //  $s_2$ 's representative is  $s_1$ 
3    $use(s_1) := use(s_1) \cup use(s_2); use(s_2) = \emptyset$ 
4    $w(s_1) := w(s_1) + w(s_2)$ 
  
```

proc *Find*(v)

```

1   $r := up(v)$  // look for the root of the tree containing  $v$ 
2  while ( $r \neq up(r)$ )  $r := up(r)$  //  $r$  is a root iff  $r = up(r)$ 
3  if ( $r \neq v$ ) // compress the path from  $v$  to  $r$ 
4      while ( $r \neq v$ )  $up(v) := r; v := up(v)$ 
5  return  $r$ 

```

Algoritma 12.2.5 Algoritma $NOS(A)$ menerima himpunan A dari persamaan dasar, yang merupakan campuran persamaan atau pertidaksamaan, sebagai masukan, membuat grafik istilah $G = (V, E)$ dari semua istilah dalam A , membangun kelas kekongruenan untuk persamaan dalam A , dan memeriksa kepuasan A .

proc $NOS(A) : Boolean$

```

1   $G := createGraph(A)$  //  $up(v) = v$  for each  $v$  in  $G$ 
2  for ( $s \doteq t$ )  $\in A$  do  $merge(v_s, v_t)$  //  $v_s, v_t$  represent  $s, t$  in  $G$ 
3  for ( $s \neq t$ )  $\in A$  do
4      if  $Find(v_s) = Find(v_t)$  return  $\perp$  // unsatisfiable
5  return  $\top$  // satisfiable

```

Contoh 12.2.6 Lanjutan dari Contoh 12.1: untuk menangani $f(a, b) \doteq a, merge(2, 3)$ dipanggil. Baris pertama $merge$ memanggil $Find$ dua kali dengan hasil $s_1 = Find(2) = 2$ dan $s_2 = Find(3) = 3$. Karena $s_1 \neq s_2$, kita mendapatkan $P_1 = \{1\}$ dan $P_2 = \{2\}$, mempersiapkan propagasi kongruensi.

Pada baris 4 $merge, Union(2, 3)$ dipanggil. Karena $w(2) = w(3) = 1$, bagian kedua dari baris 1 dilewati. Pada baris 2, $up(3)$ diubah menjadi 2, artinya node 3 menggunakan node 2 sebagai nama kelasnya. Pada baris 3, $w(2)$ diubah dari 1 menjadi 2, karena pohon kelas meningkatkan bobotnya. Pada baris 4, induk dari node 3 berpindah ke induk dari node 2.

Sekarang, kembali ke baris 5 dari $merge$: karena $P_1 \times P_2 = \{(1, 2)\}$, $t_1 = 1$ dan $t_2 = 2$ adalah satu-satunya kasus untuk t_1 dan t_2 . Karena $Find(1) = 1$ dan $Find(2) = 2$, $congruent(1, 2)$ dipanggil dan mengembalikan true, karena $Find(child(1, 1)) = Find(2) = 2 = Find(3) = Find(child(2, 1))$ dan $Find(child(1, 2)) = Find(child(2, 2)) = 4$. Sekarang, $merge(1, 2)$ dipanggil secara rekursif pada baris 7 dan akan menggabungkan node 1 ke dalam $\{2, 3\}$ (yaitu, $up(1) = 2$). Kelas kongruensi akhir adalah $\{1, 2, 3\}$ dan $\{4\}$.

Untuk ketidaksetaraan $f(f(a, b), b) \neq a$ dari A , karena $f(f(a, b), b)$ (diwakili oleh node 1) dan a (node 3) berada dalam kelas yang sama, maka $f(f(a, b), b) \neq a$ adalah salah dan A tidak dapat dipenuhi.

Nelson dan Oppen membuktikan bahwa kompleksitas waktu $NOS(A)$ adalah $O(n^2)$, di mana n adalah jumlah simbol A . Berbagai peningkatan dan perluasan $NOS(A)$ telah diusulkan, dan kompleksitas terbaik yang diketahui adalah $O(n \log(n))$. Alih-alih menyajikan bukti kompleksitas algoritma NOS , kami akan memperkenalkan pendekatan Kapur menggunakan prosedur penyelesaian Knuth-Bendix dengan analisis kompleksitas yang terperinci.

Kekongruenan Dasar dengan Penyelesaian

Menurut Teorema 12.2.3, prosedur penyelesaian Knuth-Bendix untuk persamaan dasar dapat digunakan sebagai prosedur keputusan untuk penutupan kekongruenan kesamaan dasar. Untuk membuat prosedur penyelesaian efisien, Deepak Kapur menyarankan dua langkah pra-pemrosesan yang membuat persamaan atau aturan penulisan ulang menjadi “Flat” dan “tunggal”.

Definisi 12.2.7 Suatu istilah t dikatakan *flat* jika t merupakan pengidentifikasi atau $t = f(a_1, \dots, a_k)$ dan $a_1, \dots, a_k$ merupakan pengidentifikasi. Suatu persamaan $s \doteq t$ (atau aturan penulisan ulang $s \rightarrow t$) dikatakan datar jika s dan t keduanya *flat*. $s \doteq t$ (atau $s \rightarrow t$) dikatakan tunggal jika t merupakan pengidentifikasi. $s \doteq t$ (atau $s \rightarrow t$) adalah unit jika s dan t keduanya merupakan pengidentifikasi.

Kapur mengusulkan dua langkah pra-pemrosesan: perataan dan pemotongan pada E dari persamaan dasar. Perataan akan meratakan suku-suku jika suku-suku dalam suatu persamaan tidak rata dengan memperkenalkan pengidentifikasi baru untuk sub-suku yang bukan pengidentifikasi. Pemotongan akan mengubah persamaan yang bukan tunggal menjadi dua persamaan tunggal dengan memperkenalkan pengidentifikasi baru. Setelah persamaan masukan rata dan tunggal, maka aturan penulisan ulang yang dihasilkan juga demikian. Berikut adalah definisi formal dari operasi pra-pemrosesan:

- **Perataan:** Jika t dalam E tidak rata, maka t adalah $f(\dots, g(s), \dots)$ dan kita mengganti t dengan $f(\dots, c, \dots)$ dan menambahkan $g(s) \doteq c$ ke dalam E , di mana c adalah pengidentifikasi baru.
- **Pemotongan (Slicing):** Jika $(s \doteq t) \in E$, dan baik s maupun t bukan pengidentifikasi, maka kita mengganti $s \doteq t$ dengan $s \doteq c$ dan $t \doteq c$ di E , di mana c adalah pengidentifikasi baru.
- **Penggantian (Switching):** Jika $(s \doteq t) \in E$, dan s adalah pengidentifikasi dan t bukan, maka kita mengganti $s \doteq t$ dengan $t \doteq s$ di E .

Misalkan $E_1 = \text{flattenSlicing}(E_0)$ menunjukkan prosedur yang menerapkan operasi di atas ke E_0 hingga E_1 menjadi datar dan tunggal.

Proposisi 12.2.8 Jika $E_1 = \text{flattenSlicing}(E_0)$, maka $|E_1| = O(|E_0|)$, di mana $|E_i|$ menunjukkan jumlah total simbol fungsi (termasuk pengidentifikasi) yang muncul di $E_i, i = 0, 1$.

Bukti Misalkan n_i adalah jumlah total simbol fungsi (termasuk pengidentifikasi) yang muncul di E_i untuk $i = 0, 1$. Maka, $n_0 \leq n_1 \leq 3n_0$, karena untuk operasi perataan, dua salinan konstanta baru ditambahkan ke E untuk setiap posisi subterm non-pengidentifikasi selain posisi akar. Demikian pula, untuk operasi pemotongan, pengidentifikasi baru diperkenalkan

untuk posisi akar dari term non-pengidentifikasi. Dengan kata lain, jumlah total kemunculan pengidentifikasi baru yang diperkenalkan oleh $flattenSlicing(E_0)$ tidak boleh lebih dari dua kali jumlah total posisi subterm di E_0 .

Contoh 12.2.9 Misalkan $E_0 = \{f(a, g(a)) = g(g(a))\}$ adalah input ke prosedur penyelesaian Knuth-Bendix yang dimodifikasi. Maka, $flattenSlicing(E_0)$ akan kembali E_1 :

$$E_1 = \{g(a) \doteq c_1, f(a, c_1) \doteq c_3, g(a) \doteq c_2, g(c_2) \doteq c_3\}$$

di mana pengidentifikasi baru c_1 dan c_2 diperkenalkan dengan perataan dan c_3 dengan pemotongan. Jika kita memasukkan E_1 ke dalam prosedur penyelesaian Knuth-Bendix, prosedur tersebut berakhir dengan aturan penulisan ulang berikut:

$$(1) g(a) \rightarrow c_1, \quad (2) f(a, c_1) \rightarrow c_3, \quad (3) g(c_1) \rightarrow c_3, \quad (4) c_2 \rightarrow c_1$$

Sistem penulisan ulang ini bersifat kanonik (konfluen dan terminasi), datar, dan tunggal. Sistem ini dapat berfungsi sebagai prosedur pengambilan keputusan untuk $=_{E_0}$.

Misalkan U adalah himpunan semua aturan unit yang dikembalikan oleh prosedur penyelesaian Knuth-Bendix. Setiap aturan dalam U yang sisi kirinya adalah pengidentifikasi yang diperkenalkan oleh perataan dan pemotongan dapat dihapus, karena pengidentifikasi ini tidak muncul di tempat lain selain dalam aturan ini; dengan demikian, tidak ada kemungkinan untuk menggunakan aturan ini. Untuk contoh di atas, aturan terakhir, yaitu $c_2 \rightarrow c_1$, dapat dihapus.

Contoh 12.2.10 Misalkan $E_0 = \{f^3a \doteq a, f^5a \doteq a\}$. Maka, $E_1 = flattenSlicing(E_0)$, di mana E_1 berisi persamaan-persamaan berikut:

$$\{f(a) \doteq c_1, f(c_1) \doteq c_2, f(c_2) \doteq a, f(c_2) \doteq c_3, f(c_3) \doteq c_4, f(c_4) \doteq a\}$$

Tiga persamaan pertama berasal dari $f^3a \doteq a$; semua persamaan kecuali $f(c_2) \doteq a$ berasal dari $f^5a \doteq a$, karena kita menggunakan kembali dua persamaan pertama. Dengan memasukkan E_1 ke dalam prosedur Knuth-Bendix, tiga persamaan pertama akan menghasilkan tiga aturan penulisan ulang:

$$\{(1) f(a) \rightarrow c_1, \quad (2) f(c_1) \rightarrow c_2, \quad (3) f(c_2) \rightarrow a\}$$

Persamaan keempat, $f(c_2) \doteq c_3$, ditulis ulang menjadi $a \doteq c_3$, dari mana kita peroleh

$$(4) c_3 \rightarrow a.$$

Persamaan kelima, $f(c_3) \doteq c_4$, ditulis ulang oleh (4) dan (1) menjadi $c_1 \doteq c_4$, dari mana kita peroleh

$$(5) \quad c_4 \rightarrow c_1$$

Persamaan terakhir, $f(c_4) \doteq a$, ditulis ulang oleh (5) dan (2) menjadi $c_2 \doteq a$, dari mana kita peroleh

$$(6) \quad c_2 \rightarrow a,$$

yang menulis ulang (2) menjadi $f(c_1) \rightarrow a$. Selanjutnya, (6) mereduksi (3) menjadi $f(a) \doteq a$, kemudian dengan (1) menjadi $c_1 \doteq a$, dari mana kita peroleh

$$(7) \quad c_1 \rightarrow a.$$

(1) menjadi $f(a) \rightarrow a$ berdasarkan (7). (7) juga mereduksi (2) menjadi $f(a) \doteq a$, kemudian berdasarkan (1) menjadi $a \doteq a$.

Sistem penulisan ulang terakhir adalah

$$\{(1) f(a) \rightarrow a, \quad (4) c_3 \rightarrow a, \quad (5) c_4 \rightarrow a, \quad (6) c_2 \rightarrow a, \quad (7) c_1 \rightarrow a\}$$

Jika kita menghapus aturan-aturan di U yang sisi kirinya merupakan pengidentifikasi baru, maka hanya aturan pertama, yaitu $f(a) \rightarrow a$, yang dibutuhkan untuk prosedur pengambilan keputusan $=_{E_0}$.

Selama eksekusi prosedur penyelesaian Knuth-Bendix (Proc. 7.2.15), misalkan E adalah himpunan persamaan yang tersisa dan R adalah himpunan aturan penulisan ulang saat ini, baik E maupun R adalah ground, flat, dan singleton. R dapat dipartisi menjadi U dan N , yaitu $R = U \cup N$, di mana U adalah aturan unit dan N adalah aturan non-unit. Kedua contoh di atas mengilustrasikan beberapa sifat (E, R) yang dirangkum oleh lemma berikut.

Lemma 12.2.11 Misalkan input E ke prosedur penyelesaian Knuth-Bendix adalah himpunan persamaan ground dan R adalah himpunan aturan penulisan ulang saat ini. Ketika aturan penulisan ulang baru $l \rightarrow r$ dibuat, untuk setiap $x \rightarrow y$ dari R , x tidak dapat ditulis ulang oleh $l \rightarrow r$ pada akar x . Terlebih lagi, jika x adalah konstanta, maka $x \rightarrow y$ akan tetap berada di R selamanya.

Bukti: Jika $l \rightarrow r$ dapat menulis ulang x pada akarnya, maka $l = x$, karena keduanya merupakan bilangan dasar. Jadi, l dapat ditulis ulang menjadi y oleh $x \rightarrow y$, sebuah kontradiksi dengan asumsi bahwa l berada dalam bentuk normal oleh R . Jika x adalah konstanta, maka satu-satunya cara untuk menulis ulang x adalah pada akarnya dan itu tidak mungkin. Kita dapat

mengamati sifat-sifat berikut ketika sistem penulisan ulang dijaga tetap datar dan tunggal dalam prosedur penyelesaian Knuth-Bendix:

1. Berdasarkan Lemma 12.2.11, untuk setiap $u \rightarrow v \in U$, yaitu, itu adalah aturan unit, $u \rightarrow v$ tidak akan pernah dihapus oleh aturan penulisan ulang lain selama eksekusi Prosedur 7.2.15.
2. Ketika aturan unit baru $l \rightarrow r$ dibuat, aturan unit ini dapat menulis ulang beberapa aturan N di mana l muncul dalam aturan tersebut.
3. Ketika aturan non-unit baru $l \rightarrow r$ dibuat, di mana r adalah pengidentifikasi dan l bukan, aturan non-unit ini tidak dapat menulis ulang aturan apa pun dari R , karena $l \rightarrow r$ tidak dapat menulis ulang sisi kiri dari aturan apa pun dari R pada akhirnya (Lemma 12.2.11); substitilah pada posisi lain dari aturan semuanya adalah pengidentifikasi, karena R datar dan tunggal.

Berdasarkan sifat-sifat di atas dan menggunakan beberapa ide dari algoritma Kapur serta algoritma prosedur Knuth-Bendix dimodifikasi dengan menggunakan tiga struktur data berikut untuk setiap pengidentifikasi c :

- $nf(c)$: bentuk normal dari c oleh U ; awalnya, $nf(c) = c$.
- $class(c)$: himpunan semua pengidentifikasi yang bentuk normalnya adalah c , yaitu, $class(c) = \{b : nf(b) = c\}$; awalnya, $class(c) = \{c\}$.
- $use(c)$: himpunan aturan dalam N yang menggunakan c di sisi kirinya, yaitu, $use(c) = \{l \rightarrow r \in N : l \text{ berisi } c\}$; awalnya $use(c) = \emptyset$.

Prosedur 12.2.12 Prosedur $KBG(E, >)$ mengambil himpunan E persamaan dasar, datar, dan tunggal serta urutan total $>$ di mana $f > c$ untuk simbol fungsi f dan pengenalan c apa pun dan menghasilkan sistem penulisan ulang kanonik dari E .

proc $KBG(E)$

```

1   $U := N := \emptyset$ ; initialize  $nf()$ ,  $class()$ , and  $use()$ 
2  while ( $E \neq \emptyset$ ) do
3       $(s \doteq t) := pickEquation(E)$ ;  $E := E - \{s \doteq t\}$ 
4       $s := NF(s)$ ;  $t := NF(t)$  //normalize  $s$  and  $t$  by  $R = U \cup N$ 
5      if ( $s = t$ ) continue // identity is discarded
6      else if  $identifier?(s)$  // if  $s$  is also an identifier
7          if ( $|class(t)| < |class(s)|$ )  $x := s$ ;  $s := t$ ;  $t := x$  // switch  $s$  and  $t$ 
8           $U := U \cup \{s \rightarrow t\}$  //  $s \rightarrow t$  is a unit rule
9          for  $c \in class(s)$  do  $nf(c) := t$ 
10          $class(t) := class(t) \cup class(s)$ ;  $class(s) := \emptyset$ 
11         for  $(l \rightarrow r) \in use(s)$  do // inter-reduction
12              $N := N - \{l \rightarrow r\}$ ;  $E := E \cup \{l \doteq r\}$ 
13         else //  $s \rightarrow t$  is non-unit
14              $N := N \cup \{s \rightarrow t\}$  // suppose  $s = f(a_1, \dots, a_k)$ 
15             for  $i := 1$  to  $k$  do  $use(a_i) := use(a_i) \cup \{s \rightarrow t\}$ 
16 return  $U \cup N$ 

```

```

proc  $NF(t)$ 
17  if  $identifier?(t)$  return  $nf(t)$ 
18  else assume  $t = f(t_1, \dots, t_k)$ 
19       $s := f(NF(a_1), \dots, NF(a_k))$ 
20      if  $s$  is flat and  $(s \rightarrow b) \in N$  return  $nf(b)$ 
21      else return  $s$ 

```

Seperti prosedur penyelesaian Knuth-Bendix (Proc. 7.2.15), prosedur di atas mengambil satu persamaan dari E pada setiap iterasi loop utama, menormalisasi persamaan dengan $R = U \cup N$, mencoba membuat aturan penulisan ulang dari persamaan yang dinormalisasi, dan menambahkan aturan tersebut ke R sambil menjaga R tetap tereduksi. Beberapa aturan R kembali ke E sebagai hasil reduksi. Loop utama berakhir ketika E menjadi kosong. Namun, *KBG* memiliki fitur khusus yang membuatnya lebih efisien daripada prosedur penyelesaian umum:

- Komputasi pasangan kritis dihilangkan karena reduksi sudah cukup untuk persamaan dasar.
- Prosedur normalisasi $NF(t)$ menggunakan $nf(c)$ alih-alih U untuk setiap pengidentifikasi c ; aturan penulisan ulang N dapat digunakan paling banyak sekali karena t bersifat datar. Bahkan, $NF(t)$ dapat dilakukan dalam waktu $O(|t|)$ karena kita dapat menggunakan tabel hash atau struktur trie untuk menyimpan N . Perhatikan bahwa, karena t datar, $|t|$ adalah konstanta jika aritas dari fungsi apa pun adalah konstanta.
- Ketika aturan unit $s \rightarrow t$ dibuat, kita memperbarui $nf(c)$ dengan t untuk setiap $c \in class$ (baris 9) dan menggabungkan kelas ke dalam kelas (t) (baris 10). Ingat bahwa $nf(c)$ adalah bentuk normal dari c oleh U . Karena keberadaan $nf(c)$, aturan penulisan ulang di U bersifat dangkal: aturan tersebut tidak dihapus atau disederhanakan.
- Struktur data $use(s)$ mencatat himpunan aturan non-unit yang menggunakan pengidentifikasi s di sisi kirinya (baris 14). Struktur data ini berguna pada baris 9 untuk menghapus aturan $l \rightarrow r$ dari N ketika l dapat ditulis ulang oleh aturan baru $s \rightarrow t$.
- Ketika aturan non-unit ditambahkan ke N (baris 14), N serta $R = U \cup N$ tetap saling tereduksi, sehingga tidak perlu menulis ulang N dengan aturan baru.

Kebenaran prosedur di atas berasal langsung dari Teorema 7.2.22. Kompleksitas waktu prosedur diberikan secara eksplisit oleh teorema berikut.

Teorema 12.2.13 Jika E adalah himpunan persamaan dasar, datar, dan tunggal, dan n adalah jumlah total simbol dalam E , maka prosedur *KBG* berjalan dalam $O(n \log(n))$.

Bukti Loop utama (baris 3–15) menangani $s \doteq t$ dari E dan setiap baris loop utama membutuhkan waktu yang berbeda untuk dieksekusi. Baris 4 membutuhkan waktu $O(|s| +$

$|t|$) atau $O(1)$ jika kita menganggap aritas dari setiap simbol adalah konstan. Baris 9 membutuhkan waktu $O(|class(s)|)$; Baris 11–12 membutuhkan waktu $O(|use(s)|)$. Semua baris lainnya membutuhkan waktu konstan. Jika setiap persamaan E diproses hanya sekali, maka total waktunya adalah $O(n)$, karena jumlah $|s| + |t|, |class(s)|$, atau $|use(s)|$ untuk semua persamaan $s \doteq t$ dan mengidentifikasi c dibatasi oleh $O(n)$. Jika tidak ada persamaan yang diproses lebih dari m kali, maka total waktu KBG adalah $O(nm)$.

Berapa kali sebuah persamaan dapat kembali dari N ke E (baris 12)? Perhatikan bahwa akar penyebab pemindahan persamaan dari N ke E pada baris 12 adalah karena kita memiliki aturan unit baru $s \rightarrow t$ (baris 8) dan s muncul dalam persamaan yang dipindahkan. Saat membuat aturan penulisan ulang $s \rightarrow t$ dari $s \doteq t$, kita memastikan bahwa $|class(s)| \leq |class(t)|$ (baris 7) dan kemudian menggabungkan kelas ke dalam kelas (t) (baris 10). Dengan kata lain, sebuah persamaan berpindah dari N ke E hanya ketika (sisi kiri) persamaan ini berisi pengidentifikasi yang kelasnya akan digabungkan ke dalam kelas lain. Karena kelas yang lebih kecil tidak dapat digabungkan ke dalam kelas yang lebih besar lebih dari $\log(n)$ kali, tidak ada persamaan yang dapat dipindahkan dari N ke E lebih dari $\log(n)$ kali. Jadi, kompleksitas total KBG dibatasi oleh $O(n \log(n))$.

Teorema 12.2.14 Penutupan kekongruenan kesamaan dasar dapat diputuskan dalam $O(n \log(n))$, di mana n adalah ukuran input, yaitu, jumlah total simbol yang muncul dalam input.

Bukti Diberikan himpunan A dari kesamaan dasar, misalkan $E \subseteq A$ adalah semua persamaan A . Pertama, kita memanggil $KBG(flattenSlicing(E))$ untuk mendapatkan prosedur keputusan R untuk $=_E$. Selanjutnya, untuk setiap ketidaksetaraan $(s \neq t) \in A$, kita periksa apakah $NF(s) = NF(t)$ menggunakan R ; jika ya, kembalikan “tidak dapat dipenuhi.” Seluruh proses membutuhkan waktu $O(n \log(n))$ (Teorema 12.2.13) karena $NF(s)$ dan $NF(t)$ membutuhkan waktu $O(|s| + |t|)$. Perhatikan bahwa s dan t tidak selalu datar.

Meskipun ide Kapur tentang perataan dan pemotongan digunakan dalam prosedur di atas, perataan bersifat opsional karena hasil yang sama dapat dicapai hanya dengan pemotongan (dengan asumsi setiap fungsi hash membutuhkan waktu konstan); pembuktiannya diserahkan sebagai latihan. Perataan dan pemotongan telah berhasil digunakan oleh Kapur untuk menangani simbol fungsi yang komutatif dan asosiatif. Algoritma Nieuwenhuis dan Oliveras juga dapat menghasilkan bukti otomatis dari $s =_E t$.

12.3 ARITMATIKA LINIER

Aritmatika linier adalah teori logika orde pertama di mana rumusnya bebas kuantifikasi dan mengandung variabel yang dikuantifikasi secara eksistensial di atas domain D , di mana D dapat berupa tipe rasional atau bilangan bulat dan mendukung operator aritmatika biasa seperti $+$, $-$, $\cdot$, $=$, $\leq$, $\geq$, dll., selain operator Boolean konvensional. Rumusnya linier karena

perkalian, yaitu $\cdot$, tidak diperbolehkan atas dua variabel. Rumus aritmatika linear atomik terlihat seperti

$$a_1 \cdot x_1 + \cdots + a_n \cdot x_n \leq b, \text{ or } a_1 x_1 + \cdots + a_n x_n \leq b.$$

di mana $\leq$ dapat digantikan oleh operator relasi lain, variabel $x_1, \dots, x_n$ bertipe D , dan konstanta $a_1, \dots, a_n, b \in D$. Rumus aritmatika linear kompleks dapat diperoleh dari rumus atomik dengan operator Boolean seperti pada logika orde pertama.

input	output	input	output
$A \geq B$	$B \leq A$	$\neg(A > B)$	$A \leq B$
$A = B$	$A \leq B \wedge B \leq A$	$\neg(A < B)$	$B \leq A$
$A > B$	$B + 1 \leq A$	$\neg(A \geq B)$	$A + 1 \leq B$
$A < B$	$A + 1 \leq B$	$\neg(A \leq B)$	$B + 1 \leq A$

Contoh 12.3.1 Perhatikan kendala $\neg(x_1 - 2x_2 < 3)$. Terapkan $\neg(A < B) \equiv B \leq A, 3 \leq x_1 - 2x_2$. Dengan memindahkan 3 ke kanan dan $x_1 - 2x_2$ ke kiri dari $\leq$, kita memiliki $-x_1 + 2x_2 \leq -3$.

Dua baris teratas dari tabel di atas juga dapat digunakan untuk pra-pemrosesan masalah pemrograman linier. Kendala seperti $x \geq a$ untuk $a \neq 0$ dapat diganti dengan $y \geq 0$ dengan $x = y + a$.

Teori aritmatika linear juga mendefinisikan interpretasi rumus aritmatika linear: semua fungsi aritmatika, predikat, dan konstanta diinterpretasikan oleh definisi matematika yang sudah dikenal, dan semua operator Boolean diinterpretasikan seperti yang biasa kita lihat dari logika orde pertama.

Dalam aritmatika linear bilangan bulat, batasan yang menggunakan predikat selain $\leq$ dapat dengan mudah dikonversi ke bentuk standar, seperti yang ditunjukkan oleh tabel berikut.

Aritmatika linier sangat ekspresif karena banyak masalah dalam komputasi ilmiah dan industri dapat diekspresikan dalam aritmatika linier. Aritmatika linier relevan dengan penalaran otomatis karena sebagian besar program menggunakan variabel aritmatika (misalnya, bilangan bulat) dan melakukan operasi aritmatika pada variabel tersebut. Oleh karena itu, verifikasi perangkat lunak harus berurusan dengan aritmatika. Aritmatika linier telah diselidiki secara menyeluruh setidaknya dalam dua arah: (i) optimasi melalui pemrograman linier (LP), pemrograman bilangan bulat (ILP), dan pemrograman bilangan bulat riil campuran (MILP) dan (ii) eliminasi kuantifikasi orde pertama. Dalam bab ini, kita hanya akan fokus pada (i). Secara khusus, kita tertarik pada kepuasan masalah kendala aritmatika linier dalam konteks kombinasi teori, seperti yang terjadi, misalnya, dalam penyelesaian SMT (*satisfiability modulo theories*).

Metode Simpleks dengan Contoh

Metode simpleks Dantzig (atau algoritma simpleks), yang ditemukan pada tahun 1947, adalah metode populer untuk sistem persamaan linier. Setiap sistem persamaan mendefinisikan masalah optimasi pada domain bilangan riil dengan fungsi tujuan dan daftar persamaan aritmatika linier. Algoritma simpleks dianggap oleh banyak orang sebagai salah satu dari sepuluh algoritma yang memiliki pengaruh terbesar pada pengembangan dan praktik ilmu pengetahuan dan teknik di abad kedua puluh. Mari kita pertimbangkan contoh sistem persamaan linear berikut:

$$\begin{aligned}
 &\text{Maximize } 4x_1 + 3x_2 && (12.1) \\
 &\text{subject to} \\
 &\quad x_1 + 2x_2 + x_3 = 16 \\
 &\quad x_1 + x_2 + x_4 = 9 \\
 &\quad 3x_1 + 2x_2 + x_5 = 24 \\
 &\quad x_1, x_2, x_3, x_4, x_5 \geq 0
 \end{aligned}$$

Untuk fungsi tujuan $4x_1 + 3x_2$, kita dapat memperkenalkan variabel baru x_0 dan menuliskannya dalam bentuk standar sebagai $x_0 - 4x_1 - 3x_2 = 0$, maka (12.1) menjadi (12.2):

$$\begin{aligned}
 &\text{Maximize } x_0 \text{ dimana} && (12.2) \\
 (a) \quad &x_0 - 4x_1 - 3x_2 \\
 (b) \quad &x_1 + 2x_2 + x_3 = 16 \\
 (c) \quad &x_1 + x_2 + x_4 = 9 \\
 (d) \quad &3x_1 + 2x_2 + x_5 = 24 \\
 &\quad x_1, x_2, x_3, x_4, x_5 \geq 0
 \end{aligned}$$

Perhatikan bahwa (12.2) memiliki representasi matriks sederhana, yang digunakan untuk implementasi metode simpleks.

x_0	x_1	x_2	x_3	x_4	x_5	rhs
1	-4	-3	0	0	0	0
0	1	2	1	0	0	16
0	1	1	0	1	0	9
0	3	2	0	0	1	24

Dalam suatu sistem persamaan linear, variabel dasar adalah variabel yang hanya muncul sekali dalam sistem tersebut. Dalam bentuk standarnya, koefisien variabel dasar adalah 1. Untuk contoh ini, x_0, x_3, x_4 , dan x_5 adalah variabel dasar; x_1 dan x_2 adalah variabel non-dasar. Variabel dasar mudah ditemukan dalam matriks di mana kolom yang sesuai dengan variabel tersebut hanya memiliki satu entri bukan nol. Fungsi tujuan x_0 selalu merupakan variabel dasar. Dalam sistem persamaan linear, jika jumlah variabel lebih banyak daripada jumlah

persamaan, mudah untuk menemukan variabel dasar untuk setiap persamaan dengan operasi aritmatika.

Dalam metode simpleks, variabel dasar dan variabel non-dasar dapat bertukar status jika pertukaran ini meningkatkan nilai fungsi tujuan. Metode simpleks mengulangi pertukaran semacam ini sampai diperoleh nilai optimal.

Dalam contoh ini, x_0 adalah fungsi dari x_1, x_2, x_3, x_4, x_5 . Nilai $(x_1, x_2, x_3, x_4, x_5)$ mendefinisikan sebuah titik dalam ruang pencarian. Titik-titik yang dipertimbangkan oleh metode simpleks adalah titik-titik di mana variabel non-basis bernilai nol dan nilai variabel basis ditentukan oleh persamaan. Dalam contoh ini, kita menetapkan variabel non-basis $x_1 = x_2 = 0$. Dari empat persamaan (12.2), kita memiliki $x_0 = 0, x_3 = 16, x_4 = 9$, dan $x_5 = 24$. Jadi, titik awal yang dipertimbangkan oleh metode simpleks adalah $(0, 0, 16, 9, 24)$.

Metode simpleks mencari solusi optimal dengan berulang kali menukar variabel basis dan variabel non-basis. Selanjutnya, mari kita lihat bagaimana memilih variabel basis dan non-basis untuk pertukaran.

Dari (12.2)(a), $x_0 = 4x_1 + 3x_2$. Untuk meningkatkan nilai x_0 , kita perlu meningkatkan x_1 atau x_2 , satu variabel pada satu waktu. Jika x_1 ditambah 1, x_0 bertambah 4. Jika x_2 ditambah 1, x_0 bertambah 3. Jelas, x_1 lebih disukai karena peningkatan x_0 yang lebih cepat. Oleh karena itu, x_1 dipilih untuk pertukaran.

Setelah variabel non-basis x_1 dipilih, variabel basis mana, x_3, x_4 , atau x_5 , yang harus kita ganti? Untuk menjawab pertanyaan ini, perhatikan (12.2)(b) – (d) dengan kondisi $x_2 = 0$:

$$\begin{array}{ll} x_1 + x_3 = 16 & x_3 = 16 - x_1 \geq 0 \\ x_1 + x_4 = 9 & \text{yang memberi kita } x_4 = 9 - x_1 \geq 0 \\ 3x_1 + x_5 = 24 & x_5 = 24 - 3x_1 \geq 0 \end{array}$$

Karena $x_3, x_4, x_5 \geq 0$. Dari $16 - x_1 \geq 0, x_1 \leq 16$; dari $9 - x_1 \geq 0, x_1 \leq 9$; dari $24 - 3x_1 \geq 0, x_1 \leq 8$. Karena x_1 harus memenuhi ketiga pertidaksamaan tersebut, kita memilih $x_1 = \min\{16, 9, 8\} = 8$. Karena nilai ini berasal dari $x_5 = 24 - 3x_1 \geq 0, x_5$ dipilih untuk menjadi non-basis. Persamaan $3x_1 + 2x_2 + x_5 = 24$ disebut persamaan pivot dan x_5 disebut variabel pivot; proses memilih persamaan pivot disebut pivoting.

Dari $x_1 = 8$ dan $x_1 + x_3 = 16, x_3$ memiliki nilai baru yaitu 8. Dari $x_1 + x_4 = 9, x_4 = 1$. Jadi, titik pencarian berikutnya yang dipertimbangkan oleh metode simpleks adalah $(x_1, x_2, x_3, x_4, x_5) = (8, 0, 8, 1, 0)$, di mana x_2 dan x_5 adalah variabel non-basis.

Sekarang, kita perlu menormalisasi (12.2) sehingga variabel basis hanya muncul sekali dan x_0 dinyatakan sebagai fungsi dari variabel non-basis. Hal ini dapat dilakukan sebagai berikut: dari persamaan pivot (12.2)(c), kita memiliki

$$x_1 + (2/3)x_2 + (1/3)x_5 = 8 \quad (12.3)$$

Menambahkan (12.3) sebanyak 4 kali ke (12.2)(a), kita peroleh

$$x_0 - (1/3)x_2 + (4/3)x_5 = 32$$

Mengurangi (12.3) dari (12.2)(b), kita peroleh

$$(4/3) x_2 + x_3 - (1/3) x_5 = 8$$

Mengurangi (12.3) dari (12.2)(c), kita peroleh

$$(1/3) x_2 + x_4 - (1/3) x_5 = 1$$

Menggabungkan persamaan-persamaan ini, kita peroleh sistem persamaan baru dalam bentuk standar:

$$\begin{array}{ll} \text{Maximize } x_0 \text{ dimana} & (12.4) \\ (a) & x_0 - (1/3)x_2 - (4/3)x_5 = 32 \\ (b) & (4/5)x_2 + x_3 - (1/3)x_5 = 8 \\ (c) & (1/3)x_2 - x_4 - (1/3)x_5 = 1 \\ (d) & x_1 + (2/3)x_2 + (1/3)x_5 = 8 \\ & x_1, x_2, x_3, x_4, x_5 \geq 0 \end{array}$$

Matriks yang bersangkutan diberikan di bawah ini:

x_0	x_1	x_2	x_3	x_4	x_5	rhs
1	0	$-1/3$	0	0	$4/3$	32
0	0	$4/3$	1	0	$-1/3$	8
0	0	$1/3$	0	1	$-1/3$	1
0	1	$2/3$	0	0	$1/3$	8

Perhatikan bahwa matriks di atas dapat diperoleh dari representasi matriks (12.2) dengan operasi aritmatika yang sesuai pada (12.2). Misalnya, (12.3) sama dengan “membagi baris keempat matriks untuk (12.2) dengan 3.” “Menambahkan (12.3) sebanyak 4 kali ke (12.2)(a)” dapat dilakukan dengan “menambahkan perkalian baris keempat dengan 4 ke baris pertama matriks.” “Mengurangi (12.3) dari (12.2)(b)” dapat dilakukan dengan “mengurangi baris keempat matriks dari baris kedua matriks.” “Mengurangi (12.3) dari (12.2)(c)” dapat dilakukan dengan “mengurangi baris keempat matriks dari baris ketiga matriks.” Hasilnya persis sama dengan matriks untuk (12.4). Inilah cara metode simpleks diimplementasikan.

Pada titik (8, 0, 8, 1, 0), nilai x_0 adalah 32, meningkat dari nilai sebelumnya yaitu 0. Apakah ini nilai optimal untuk x_0 ? Dari (12.4)(a), kita memiliki

$$x_0 = (1/3) x_2 - (4/3) x_5 + 32$$

Sekarang, $x_0 = 32$ ketika $x_2 = x_5 = 0$. Jika x_2 ditambah 1, x_0 bertambah $1/3$. Artinya, ada ruang untuk penambahan x_0 dengan memilih x_2 sebagai variabel basis baru.

Proses pemilihan variabel basis yang ada untuk pertukaran sama seperti sebelumnya: menemukan variabel pivot dalam persamaan pivot. Dengan asumsi $x_5 = 0$, dari (12.4)(b) – (d), kita memiliki

$$\begin{array}{ll} (4/3)x_2 + x_3 = 8 & x_3 = 8 - (4/3)x_2 \geq 0 \\ (1/3)x_2 + x_4 = 1 & \text{yang memberi kita } x_4 = 1 - (1/3)x_2 \geq 0 \\ x_1 + (2/3)x_2 = 8 & x_1 = 8 - (2/3)x_2 \geq 0 \end{array}$$

Dari $8 - (4/3)x_2 \geq 0$, $x_2 \leq 6$; dari $1 - (1/3)x_2 \geq 0$, $x_2 \leq 3$; dari $8 - (2/3)x_2 \geq 0$, $x_2 \leq 12$. Oleh karena itu, kita memilih $x_2 = 3$ untuk memenuhi ketiga pertidaksamaan tersebut, dan persamaan pivotnya adalah $(1/3)x_2 + x_4 - (1/3)x_5 = 1$ ((12.4)(c)). Sekarang, kalikan (12.4)(c) dengan 3, kita peroleh

$$x_2 + 3x_4 - x_5 = 3 \quad (12.5)$$

Kalikan (12.5) dengan $1/3$ dan tambahkan hasilnya ke (12.4)(a), kita peroleh

$$x_0 + x_4 + x_5 = 33 \quad (12.6)$$

Kalikan (12.5) dengan $-4/3$ dan tambahkan hasilnya ke (12.4)(b), kita peroleh

$$x_3 - 4x_4 + x_5 = 4 \quad (12.7)$$

Kalikan (12.5) dengan $-2/3$ dan tambahkan hasilnya ke (12.4)(d), kita peroleh

$$x_1 - 2x_4 + x_5 = 6 \quad (12.8)$$

Representasi matriks dari persamaan (12.5)–(12.8) diberikan sebagai berikut:

x_0	x_1	x_2	x_3	x_4	x_5	rhs
1	0	0	0	1	1	34
0	0	0	1	-4	1	4
0	0	1	0	3	-1	3
0	1	0	0	-2	1	6

Dari (12.6), kita memiliki $x_0 = 33 - x_4 - x_5$. Karena $x_4, x_5 \geq 0$, x_0 memiliki nilai maksimal 33 ketika $x_4 = x_5 = 0$. Nilai variabel dasar dapat dihitung dari persamaan dengan $x_4 = x_5 = 0$: $x_1 = 6$, $x_2 = 3$, dan $x_3 = 4$. Artinya, x_0 memiliki nilai optimal 33 pada titik $(x_1, x_2, x_3, x_4, x_5) = (6, 3, 4, 0, 0)$. Metode simpleks berhenti pada hasil ini.

Dari contoh ini, kita dapat melihat bahwa metode simpleks adalah prosedur pencarian lokal yang dimulai dari $(0, 0, 16, 9, 24)$, menemukan titik tetangga terbaiknya $(8, 0, 8, 1, 0)$

melalui persamaan pivot, dan kemudian menemukan titik terbaik berikutnya. Pencarian berhenti ketika mencapai optimum lokal. Untungnya, karena semua kendala bersifat linier, optimum lokal adalah optimum global. Pembahasan rinci tentang pernyataan ini berada di luar cakupan buku ini.

Algoritma Simpleks

Misalkan sistem persamaan linier memiliki m kendala dan n variabel, dengan $m + 1$ variabel pertama sebagai variabel dasar. Oleh karena itu, kita mengasumsikan $m \leq n$ karena, ketika $m > n$, sistem linier tersebut tidak memiliki solusi atau salah satu kendalanya berlebihan. Bentuk standarnya diberikan di bawah ini:

$$\begin{array}{rcl} \text{Maximize } x_0 \text{ dimana} & & (12.9) \\ x_0 + \cdots + a_{0,m+1}x_{m+1} + \cdots + a_{0,n}x_n & = & b_0 \\ x_1 + \cdots + a_{1,m+1}x_{m+1} + \cdots + a_{1,n}x_n & = & b_1 \\ x_2 + \cdots + a_{2,m+1}x_{m+1} + \cdots + a_{2,n}x_n & = & b_2 \\ \vdots & & \vdots \\ x_m + a_{m,m+1}x_{m+1} + \cdots + a_{m,n}x_n & = & b_m \\ x_1, x_2, \dots, x_n & \geq & 0 \end{array}$$

Teorema 12.3.2 (Kriteria Optimalitas) Untuk sistem persamaan linear (12.9), jika $a_{0,j} \geq 0$ untuk $j = m + 1, \dots, n$, maka nilai maksimum x_0 adalah b_0 dan dicapai pada titik $(b_1, b_2, \dots, b_m, 0, \dots, 0)$.

Bukti Dari baris pertama (12.9), kita memiliki $x_0 = b_0 - (a_{0,m+1}x_{m+1} + \cdots + a_{0,n}x_n)$. Karena $a_{0,j} \geq 0$ dan $x_j \geq 0$ untuk $j = m + 1, \dots, n$, maka $a_{0,m+1}x_{m+1} + \cdots + a_{0,n}x_n \geq 0$. Oleh karena itu, nilai maksimum x_0 adalah b_0 ketika $x_j = 0$ untuk $j = m + 1, \dots, n$. Dalam situasi ini, $x_i = b_i$ untuk $i = 1, \dots, m$.

Teorema 12.3.3 (Kriteria Tak Terbatas) Untuk sistem persamaan linear (12.9), jika terdapat indeks $j, m + 1 \leq j \leq n$, sedemikian sehingga $a_{0,j} < 0$ dan $a_{i,j} \leq 0$ untuk semua $i = 1, \dots, m$ maka nilai optimal x_0 adalah ∞ .

Bukti Jika terdapat indeks $j, m + 1 \leq j \leq n, a_{0,j} < 0$, dan $a_{i,j} \leq 0$ untuk semua $i = 1, \dots, m$, maka kita memiliki banyak titik tak terbatas, di mana x_j dapat memiliki nilai apa pun, $x_i = b_i - a_{i,j}x_j$ untuk $i = 1, \dots, m$, dan $x_i = 0$ untuk $m + 1 \leq i \leq n$ dan $i \neq j$. Masing-masing titik ini mendefinisikan titik solusi yang layak untuk x_0 , yaitu, $x_0 = b_0 - a_{0,j}x_j$. Dengan demikian, x_0 menuju ∞ ketika x_j menuju ∞ karena $a_{0,j} < 0$.

Contoh 12.3.4 Misalkan kita ingin memaksimalkan x_0 dengan

$$\begin{array}{rcl} x_0 & + & 2x_3 - 3x_4 = 10 \\ x_1 & + & 2x_3 - x_4 = 3 \end{array}$$

$$x_2 - x_3 - 2x_4 = 2$$

Misalkan $x_3 = 0$, maka $(x_0, x_1, x_2, x_3, x_4) = (3x_4 + 10, x_4 + 3, 2x_4 + 2, 0, x_4)$ adalah titik solusi yang layak untuk semua kendala. Nilai x_0 menuju ∞ ketika x_4 juga menuju ∞ .

Kedua teorema di atas memberi kita dua kriteria penghentian untuk algoritma simpleks: satu memberi kita solusi optimal dan yang lainnya memberi kita solusi tak terbatas. Misalkan *haltCondition*(*M*) adalah prosedur yang mengembalikan nilai true jika dan hanya jika kondisi dalam kedua teorema di atas benar. Kemudian, garis besar algoritma simpleks dapat dijelaskan di bawah ini, mengikuti ilustrasi subbagian sebelumnya.

proc simplex(*S*)

- 1 Convert *S* in standard form and store it in matrix *M*.
- 2 Let *r* be the row of *M* with basic variable x_0 , which is to be maximized.
- 3 **while** $\neg \text{haltCondition}(M)$ **do**
- 4 select a nonbasic variable v_1 in *r* with the minimal coefficient.
- 5 select a basic variable v_2 by pivoting with respect to v_1 .
- 6 switch the roles of v_1 and v_2 and convert *M* in standard form.

Kompleksitas algoritma simpleks bersifat eksponensial dalam kasus terburuk. Karena ada prosedur pengambilan keputusan polinomial untuk pemrograman linier, ini berarti setidaknya secara teoritis ada prosedur pengambilan keputusan yang lebih cepat. Tetapi berbeda dengan prosedur pengambilan keputusan polinomial tersebut, algoritma simpleks memiliki semua properti yang diperlukan untuk pemecah pemrograman linier yang efisien: menghasilkan penjelasan konflik minimal, menangani backtracking secara efisien, dan sangat inkremental. Dalam praktiknya, properti ini lebih penting daripada perbedaan antara kompleksitas kasus terburuk polinomial dan eksponensial. Hal ini juga membantu karena algoritma simpleks jarang mencapai kasus terburuknya dalam praktik.

Pemrograman Linier

Nama "pemrograman linier" sudah ada sebelum era komputer. Di sini "pemrograman" berarti "tabel," karena data dari masalah pemrograman linier, atau program linier singkatnya, biasanya disimpan dalam sebuah tabel. Bentuk standar dari program linier diberikan sebagai berikut:

$$\begin{array}{ll}
 \text{Maximize } a_{0,1}x_1 + \cdots + a_{0,n}x_n & (12.8) \\
 \text{subject to} & \\
 a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n & \leq b_1 \\
 a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n & \leq b_2 \\
 \cdots & \vdots \\
 a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n & \leq b_m \\
 x_1, x_2, \dots, x_n & \geq 0
 \end{array}$$

di mana setiap $a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n \leq b_i$ adalah kendala linear atomik dan kita mencari nilai $x_{i,1} \leq i \leq n$, sedemikian sehingga fungsi tujuan dimaksimalkan.

Contoh 12.3.5 Di bawah ini adalah program linear dengan $n = 2$ dan $m = 3$:

$$\begin{array}{ll} \text{Maximize } 4x_1 + 3x_2 & (12.9) \\ \text{subject to} & \end{array}$$

$$\begin{array}{ll} x_1 + 2x_2 & \leq 16 \\ x_1 + x_2 & \leq 9 \\ 3x_1 + 2x_2 & \leq 24 \\ x_1, x_2 & \geq 0 \end{array}$$

Solusinya akan dibahas sebentar lagi.

Banyak masalah aplikasi dapat ditentukan sebagai program linear. Untuk menyelesaikan masalah ini menggunakan metode simpleks, kita memperkenalkan m variabel baru, yang disebut variabel slack, satu untuk setiap m kendala, untuk mengubahnya menjadi persamaan.

Contoh 12.3.6 Melanjutkan dari contoh sebelumnya, kita memperkenalkan tiga variabel, x_3 , x_4 , dan x_5 , dan mengubah (12.8) menjadi sistem persamaan linear:

$$\begin{array}{ll} \text{Maximize } 4x_1 + 3x_2 & (12.10) \\ \text{subject to} & \end{array}$$

$$\begin{array}{ll} x_1 + 2x_2 + x_3 & = 16 \\ x_1 + x_2 + x_4 & = 9 \\ 3x_1 + 2x_2 + x_5 & = 24 \\ x_1, x_2, x_3, x_4, x_5 & \geq 0 \end{array}$$

(12.10) identik dengan (12.1) di Bagian 12.3.1 dan kita telah melihat bagaimana (12.1) diselesaikan dengan metode simpleks. Oleh karena itu, solusi optimal dari $4x_1 + 3x_2$ adalah 33 ketika $x_1 = 6$ dan $x_2 = 3$.

Menurut definisi, pemrograman linier adalah masalah optimasi. Kita dapat mengubahnya menjadi masalah keputusan dengan memperkenalkan konstanta c , sehingga “Maksimumkan $a_{0,1}x_1 + \dots + a_{0,n}x_n$ ” menjadi “Putuskan $a_{0,1}x_1 + \dots + a_{0,n}x_n \geq c$.” Masalah keputusan lain yang terkait dengan pemrograman linier adalah memutuskan apakah himpunan semua kendala konsisten. Artinya, dapatkah kita menemukan nilai $x_1, x_2, \dots, x_n$ sedemikian sehingga semua kendala benar? Masalah ini dapat dengan mudah diselesaikan jika kita mengubah pemrograman linier menjadi sistem persamaan linier melalui variabel slack.

Pemrograman Bilangan Bulat

Bentuk standar program bilangan bulat sama dengan pemrograman linier, dengan batasan tambahan bahwa variabel hanya menerima bilangan bulat:

$$\begin{array}{ll} \text{Maximize } a_{0,1}x_1 + \dots + a_{0,n}x_n & (12.11) \\ \text{subject to} & \end{array}$$

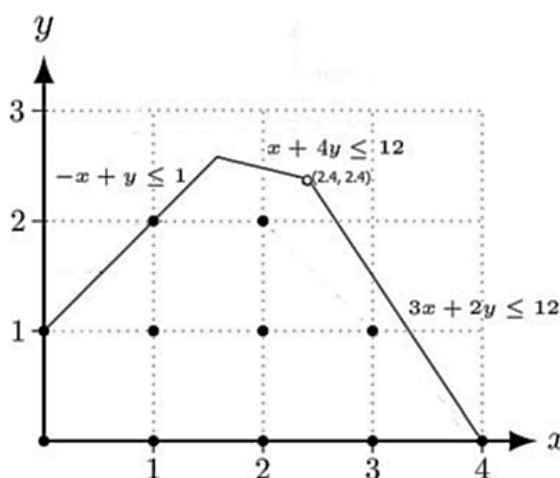
$$\begin{array}{ll} a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n}x_n & \leq b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \dots + a_{2,n}x_n & \leq b_2 \\ & \dots \end{array}$$

$$a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n \leq b_m$$

$$x_1, x_2, \dots, x_n \text{ are integers,}$$

di mana $a_{i,j}, b_i$ diasumsikan sebagai bilangan bulat.

Jika variabel hanya mengambil nilai 0 atau 1, pemrograman bilangan bulat menjadi pemrograman linier 0 – 1. Sementara pemrograman linier dapat diselesaikan dalam waktu polinomial, pemrograman linier 0 – 1 adalah NP-lengkap. Dengan kata lain, peluang menemukan algoritma waktu polinomial untuk pemrograman linier 0 – 1, atau pemrograman bilangan bulat secara umum, sangat kecil.



Gambar 12.2 Contoh pemrograman bilangan bulat

Contoh 12.3.7 Perhatikan contoh pemrograman bilangan bulat berikut ini:

$$\begin{aligned} &\text{Maximize } 4x + 3y \\ &\text{subject to} \end{aligned} \tag{12.12}$$

$$\begin{aligned} -x + y &\leq 1 \\ x + 5y &\leq 12 \\ 3x + 2y &\leq 12 \end{aligned}$$

x, y are non – negative integers.

Jika kita menghilangkan batasan bahwa “ x, y adalah bilangan bulat,” masalah ini menjadi contoh pemrograman linier. Solusi optimal dapat ditemukan dengan metode simpleks dengan solusi optimal $(x, y) = (2, 4, 2, 4)$, yaitu $4x + 3y = 16,8$. Dengan batasan bahwa x, y adalah bilangan bulat, terdapat 11 solusi yang layak (titik hitam pada Gambar 12.2) dan satu solusi optimal, yaitu $(x, y) = (4, 0)$, atau $4x + 3y = 16,8$.

Meskipun metode simpleks efektif untuk menyelesaikan program linier, tidak ada satu teknik efektif pun untuk menyelesaikan program bilangan bulat. Sebaliknya, sejumlah teknik telah dikembangkan, dan kinerja teknik tertentu tampaknya sangat bergantung pada masalahnya. Metode yang ada hingga saat ini dapat diklasifikasikan secara luas menjadi tiga pendekatan:

- **Teknik pencarian lokal:** Mulailah dengan beberapa solusi kandidat yang menjanjikan, lalu pilih beberapa variabel (misalnya, variabel yang muncul dalam kendala yang dilanggar) untuk memodifikasi nilainya (misalnya, menambah atau mengurangi 1) secara berurutan untuk menemukan tetangga dari solusi kandidat ini. Pilih yang terbaik di antara tetangga-tetangga ini dan ulangi. Solusi kandidat dapat ditemukan dengan (1) melonggarkan pemrograman bilangan bulat menjadi pemrograman linier; (2) menggunakan metode simpleks untuk menemukan solusi optimal untuk pemrograman linier yang dilonggarkan; dan (3) membulatkan solusi optimal ke atas/bawah menjadi solusi bilangan bulat.
- **Teknik enumerasi:** Teknik enumerasi yang terkenal adalah prosedur branch-and-bound. Prosedur ini sangat mirip dengan prosedur branch-and-bound untuk kepuasan maksimum (Bagian 4.4) dan sangat efektif untuk pemrograman linier 0-1.
- **Teknik bidang potong:** Ide dasar metode bidang potong adalah memotong bagian dari wilayah layak dari ruang pencarian pemrograman linier, sehingga solusi bilangan bulat optimal menjadi titik ekstrem dan karenanya dapat ditemukan dengan metode simpleks. Dalam praktiknya, metode branch-and-bound sebagian besar mengungguli metode bidang potong. Bidang potong adalah algoritma pertama yang dikembangkan untuk pemrograman bilangan bulat yang dapat dibuktikan konvergen dalam jumlah langkah terbatas. Meskipun teknik ini dianggap tidak efisien, teknik ini telah memberikan wawasan tentang pemrograman bilangan bulat yang telah mengarah pada algoritma lain yang lebih efisien.

Contoh 12.3.8 Dalam Contoh 12.3.7, $(x, y) = (2.4, 2.4)$ adalah solusi optimal dari masalah pemrograman linier yang direlaxasi. Solusi pembulatan ke bawah adalah $(x, y) = (2, 2)$, yang merupakan salah satu titik awal untuk pencarian lokal. Tetangga dari $(x, y) = (2, 2)$ adalah $\{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)\}$.

Dari $(2, 2)$, pencarian lokal akan menemukan $(3, 1)$, kemudian $(4, 0)$, solusi optimal.

Menerapkan teknik enumerasi pada contoh ini, kita mulai dengan y terlebih dahulu dengan $y = 0$. Batasan memberi kita $0 \leq x \leq 4$ dan nilai optimalnya adalah $(x, y) = (4, 0)$. Ketika $y = 1, 2$, nilai optimalnya adalah $(x, y) = (3, 1), (2, 2)$, masing-masing. Nilai optimal keseluruhan adalah $(x, y) = (4, 0)$.

Dengan menerapkan teknik bidang potong pada contoh ini, kita dapat menambahkan $x < 2$ dan $x \geq 2$, masing-masing, ke kendala asli dan mencari solusi optimal untuk submasalah yang lebih terkendala. Solusi optimal keseluruhan berasal dari $x \geq 2$.

Selain itu, beberapa prosedur komposit telah diusulkan, yang menggabungkan teknik menggunakan beberapa pendekatan ini. Bahkan, ada tren dalam sistem komputer untuk pemrograman bilangan bulat untuk memasukkan sejumlah pendekatan dan mungkin memanfaatkan semuanya ketika menganalisis masalah tertentu. Pembaca yang tertarik dapat

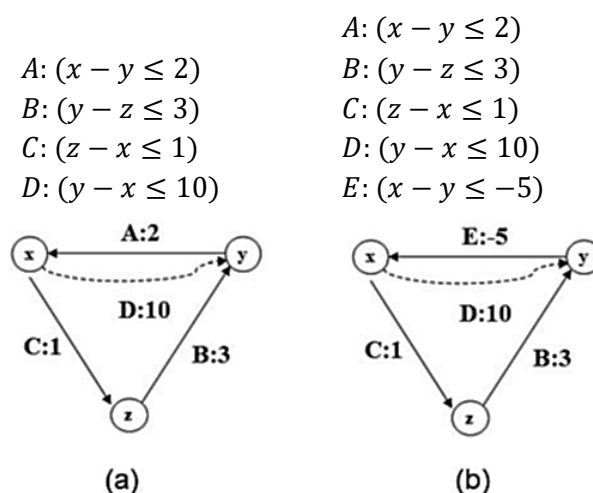
berkonsultasi dengan buku teks tentang pemrograman bilangan bulat untuk detail teknik-teknik ini.

Logika Perbedaan

Logika perbedaan adalah subkelas aritmatika linier di mana setiap kendala atomik berisi dua variabel, dan koefisien dari kedua variabel tersebut adalah 1 dan -1 . Misalnya, $x_1 - x_2 \leq 3$, atau setara dengan $x_1 \leq x_2 + 3$, adalah kendala perbedaan. Jenis variabel dapat berupa bilangan riil atau bilangan bulat, memberi kita variasi dari masalah yang sama. Sebuah solusi adalah penugasan semua variabel dengan semua kendala terpenuhi.

Kendala perbedaan ini secara alami muncul dalam banyak aplikasi. Misalnya, pemeriksaan batas array dalam sebuah program dan kendala waktu dalam penjadwalan pekerjaan dapat ditentukan sebagai kendala perbedaan. Aplikasi lain adalah penjadwalan waktu nyata: waktu yang diminta oleh suatu tugas dapat dimodelkan sebagai perbedaan minimal antara tugas ini dan tugas berikutnya: setiap variabel sesuai dengan mesin dan tugas dan mewakili waktu ketika tugas ini dieksekusi pada mesin ini. Jika durasi setiap eksekusi tetap, seseorang dapat menggunakan aritmatika perbedaan untuk mengekspresikan kendala seperti waktu mulai dan berakhir paling awal, urutan pergantian mesin, dan urutan eksekusi.

Terdapat prosedur pengambilan keputusan yang efisien dan terkenal untuk kendala perbedaan. Dengan diberikan serangkaian kendala perbedaan, seseorang dapat mengurangi masalah pemeriksaan kepuasannya menjadi masalah mendeteksi siklus negatif dalam grafik yang dihasilkan secara tepat. Kemudian, algoritma deteksi siklus negatif apa pun dapat digunakan untuk memutuskan kendala yang diberikan. Misalnya, algoritma Bellman-Ford klasik dapat memutuskan m kendala perbedaan pada n variabel dalam waktu $O(nm)$ dan kompleksitas ruang $O(n + m)$.



Gambar 12.3 Dua contoh aritmatika perbedaan bilangan bulat

Contoh 12.3.9 Pertimbangkan himpunan kendala perbedaan bilangan bulat berikut:

$$(A): (x - y \leq 2), \quad (B): (y - z \leq 3), \quad (C): (z - x \leq 1), \quad (D): (y - x \leq 10)$$

Graf berarah berbobot yang sesuai dengan himpunan ini ditunjukkan pada Gambar 12.3a. Jelas, tidak ada siklus negatif dalam graf tersebut. Terdapat penugasan tak terbatas untuk x, y , dan z , sedemikian sehingga semua kendala terpenuhi. Misalnya, $(x, y, z) = (0, 0, 0)$, atau $(2, 6, 3)$.

Jika kita menambahkan $(E): (x - y \leq -5)$ ke himpunan kendala di atas, maka (E) membuat (A) menjadi benar secara trivial, sehingga menggantikan (A) . Graf baru adalah (b) pada Gambar 12.3. Sekarang, terdapat siklus negatif dalam (b): $x \Rightarrow z \Rightarrow y \Rightarrow x$. Tidak peduli bagaimana kita menetapkan nilai untuk x, y , dan z , kita akan memiliki $0 \leq -1$, sebuah kontradiksi.

Kita sekarang telah melihat pemecah teori dasar untuk aritmatika perbedaan. Hal ini menentukan kepuasan suatu konjungsi literal dengan memeriksa tidak adanya siklus negatif dalam graf berbobot yang dihasilkan.

12.4 MEMBUAT DPLL(T) PRAKTIS

Pengembangan kerangka kerja $DPLL(T)$ sangat termotivasi oleh masalah SMT yang muncul dalam verifikasi formal desain komputer, termasuk desain perangkat lunak dan perangkat keras. Masalah SMT ini melibatkan berbagai teori yang jauh lebih luas daripada yang telah kami sajikan di bagian sebelumnya. Apa saja masalah ini dan prosedur pengambilan keputusannya? Untuk membuat $DPLL(T)$ praktis, kita perlu mengetahui masalah-masalah ini dan, yang lebih penting, mengembangkan prosedur pengambilan keputusan untuk masalah-masalah ini. Beberapa masalah SMT melibatkan lebih dari satu teori. Kita dapat memanggil prosedur pengambilan keputusan ini satu per satu sampai ditemukan model yang konsisten dengan semua teori. Apakah ada cara yang efisien untuk menggabungkan prosedur pengambilan keputusan yang terlibat sehingga model yang konsisten dengan semua teori dapat ditemukan dengan cepat?

Pada Bagian 12.1 bab ini, kami menyajikan algoritma $DPLL(T)$, yang mengilustrasikan bagaimana $DPLL(T)$ dapat diimplementasikan dengan modifikasi minimal pada algoritma $DPLL$. Bagaimana cara mendapatkan implementasi $DPLL(T)$ yang efisien?

Pertanyaan-pertanyaan di atas adalah topik penelitian yang sedang berlangsung, dan kita akan membahasnya secara singkat di bagian ini.

Membuat DPLL(T) Efisien

Untuk membuat $DPLL(T)$ praktis, kita harus mengimplementasikan $DPLL(T)$ secara efisien. Di bagian pertama, kita menunjukkan bagaimana mengimplementasikan $DPLL(T)$ dengan mudah jika kita memiliki prosedur $isTheoryConsistent(\sigma)$ yang mengimplementasikan prosedur pengambilan keputusan untuk sekumpulan literal SMT yang sesuai dengan model σ . Kekurangan utama dari pendekatan ini adalah bahwa pemecah teori dapat mendeteksi ketidakpuasan untuk subset literal SMT yang sesuai dengan model proposisional parsial. Kita dapat menggunakan informasi ini untuk mempercepat $DPLL(T)$, karena waktu pencarian yang dihabiskan dari model parsial ini ke model lengkap dapat dihemat. Untuk mengimplementasikan ide ini, kita perlu memanggil $isTheoryConsistent(\sigma)$ setiap kali σ diperluas.

Berikutnya, kita akan menyajikan versi baru dari $DPLL(X)$. Asumsi dan prosedur berikut akan digunakan:

- Kita akan menggunakan variabel global dalam $DPLL(X)$ untuk X (sekumpulan klausa SMT), $\gamma(X)$ (abstrak dari X), σ (model parsial dari $\gamma(X)$), dan $level$ (nomor level $DPLL$, yaitu jumlah literal keputusan aktif), dan $conflictClause$ (klausa yang bertentangan terbaru), sehingga variabel-variabel tersebut tidak muncul dalam parameter prosedur.
- Prosedur $abstraction(X)$ menghitung $\gamma(X)$ dan menyimpan pasangan $(A, \gamma(A))$ untuk setiap formula atomik A dari X .
- Prosedur $initialize(C)$ menyimpan klausa non-unit dari C dalam struktur data watch-literal dan mengembalikan klausa unit dari C .
- Prosedur $BCP(U)$ melakukan propagasi kendala Boolean pada $\gamma(X)$ dengan klausa unit U dan σ dan memperluas σ . Prosedur ini akan mengembalikan *false* jika ditemukan klausa yang bertentangan, dan klausa ini akan disimpan dalam $conflictClause$.
- Prosedur $isTheoryConsistent(\sigma)$ memeriksa konsistensi literal SMT yang sesuai dengan σ . Jika ditemukan inkonsistensi, prosedur ini akan mengidentifikasi subset σ yang menyebabkan inkonsistensi. Negasi dari subset ini menciptakan klausa yang bertentangan, yang akan disimpan dalam $conflictClause$, untuk mendukung pembelajaran klausa berbasis konflik ($CDCL$). Ini adalah pendekatan alternatif dan lebih baik daripada membuat klausa dari negasi semua literal keputusan.
- Prosedur $theoryPropagate(\sigma)$ adalah prosedur baru dan opsional. Ini merupakan kelanjutan dari prosedur $isTheoryConsistent(\sigma)$. Pemecah teori akan menghitung beberapa literal baru dalam teori sebagai konsekuensi logis dari literal yang diturunkan dari σ . Literal yang digunakan akan dikumpulkan sebagai alasan dari literal yang diturunkan. Literal proposisional yang sesuai akan dimasukkan ke $BCP(U)$ sebagai klausa unit baru dengan alasan yang sesuai. Prosedur ini tidak diperlukan untuk penyelesaian. Jika propagasi mahal, dapat dihilangkan atau dipanggil ketika model proposisional lengkap ditemukan.

Mengikuti algoritma yang diusulkan oleh Cesare Tinelli, versi baru $DPLL(X)$ diberikan di bawah ini, bersama dengan prosedur $propagateGivesConflict(U)$ dan $analyzeConflict()$.

Algoritma 12.4.1 $DPLL(X)$ mengambil sebagai input himpunan X dari klausa SMT dan mengembalikan *true* jika X dapat dipenuhi dan mengembalikan *false*, jika tidak.

```

proc  $DPLL(X)$ : Boolean
    // return true iff SMT clauses  $X$  is satisfiable
1    $U := initialize(abstraction(X))$  //  $U$  is the set of unit clauses in  $\gamma(X)$ 
2    $level := 0$ ;  $\sigma := \{\}$  //  $\sigma$  is a partial model of  $\gamma(X)$ .
3   while (true) do
4       while ( $propagateGivesConflict(U)$ )
5           if ( $level = 0$ ) return false //  $\gamma(X)$  as well as  $X$  is unsatisfiable

```

```

6      else  $U := \text{analyzeConflict}()$ 
7       $\text{restartIfApplicable}()$  // implement restart strategy
8       $A := \text{pickLiteral}()$  // pick an unassigned literal as decision literal.
9      if ( $A = \text{nil}$ ) // a model of  $\gamma(X)$  is found.
10     if  $\text{isTheoryConsistent}(\sigma)$  return true //  $X$  is satisfiable.
11     else  $U := \text{analyzeConflict}()$ 
12     else  $U := \{A\}; \text{level} := \text{level} + 1$ 

```

```

proc  $\text{propagateGivesConflict}(U)$ : Boolean
    // return true iff an inconsistency is found
1    $\text{good} := \text{true}$ 
2   while ( $\text{good}$ ) do
3       if ( $\text{not } \text{BCP}(U)$ ) return true
        // check theory consistency of  $\sigma$ 
4       if ( $\text{not } \text{isTheoryConsistent}(\sigma)$ ) return true
5        $\text{good} := \text{theoryPropagate}(\sigma)$ 
6   return false

```

```

proc  $\text{analyzeConflict}()$ 
    // return a unit clause learned from a conflicting clause
1    $c := \text{conflictClause}$  // this is the latest conflicting clause
2    $U := \text{insertNewClause}(\text{conflictAnalysis}(c))$  // new level is set
3    $\text{undo}()$ ; // undo to the new level
4   return  $U$  // new clause is unit under  $\sigma$  after  $\text{undo}()$ .

```

Prosedur *conflictAnalysis*, *undo*, dan *insertNewClause* sama dengan yang ada di Bagian 4.2 untuk pembelajaran klausa yang diarahkan konflik.

Contoh 12.4.2 Misalkan X berisi klausa dari Contoh 12.1.6:

$$(b \doteq c), (h(a) \doteq h(c)) \vee p, (a \doteq b) \vee \neg p \vee (a \doteq d), (a \doteq b) \vee (a \neq d)$$

Prosedur abstraksi menghasilkan himpunan klausa proposisional yang sama:

$$(s), (t \vee p), (q \vee \neg p \vee r), (q \vee \neg r)$$

di mana q mewakili $a \doteq b$, r mewakili $a \doteq d$, s mewakili $b \doteq c$ dan t mewakili $h(a) \doteq h(c)$. Setelah eksekusi baris 1 dari $DPLL(X)$, $U = \{s\}$. Pada baris 4, $\text{propagateGivesConflict}(U)$ akan mengembalikan false dengan $\sigma = \{s\}$ (yaitu, $b \doteq c$ adalah benar).

Pada baris 8, misalkan *pickLiteral* memilih $\neg t$ (yaitu, $h(a) \neq h(c)$) menjadi benar. Sekarang, baris 12 akan dieksekusi, $U = \{\neg t\}$ dan $\text{level} = 1$.

Pada baris 4, $\text{propagateGivesConflict}(U)$ melakukan langkah-langkah berikut:

1. $\text{BCP}(U)$ akan membuat $\neg t$ dan p menjadi benar (karena $(t \vee p)$). Oleh karena itu, $\sigma = \{s, \neg t, p\}$.

2. *theoryPropagate* akan membuat $\neg q$ (yaitu, $a \neq b$) menjadi benar, karena $\neg t$ (yaitu, $h(a) \neq h(c)$) dan s (yaitu, $b \doteq c$) benar, maka $a \doteq b$ tidak mungkin benar. Jadi, *Reason*($\neg q$) adalah $(s \wedge \neg t) \rightarrow \neg q$, atau setara dengan $(\neg q \vee \neg s \vee t)$. *theoryPropagate* mengembalikan nilai benar.
3. Dengan $\neg q$ ditambahkan ke U , *B_{CP}*(U) akan menambahkan $\neg q$ ke σ dan menemukan bahwa $(q \vee \neg p \vee r)$ menjadi satuan. Jadi, r (yaitu, $a \doteq d$) ditambahkan ke σ , dan *B_{CP}*(U) akan menemukan bahwa $(q \vee \neg r)$ menjadi bertentangan. *B_{CP}*(U) mengembalikan *false* setelah $(q \vee \neg r)$ disimpan dalam *conflictClause*, dan *propagateGivesConflict*(U) mengembalikan *true*.

Dengan $(q \vee \neg r)$ disimpan dalam *conflictClause*, pada baris 6 dari *DPLL_T*(X), *analyzeConflict*() dieksekusi dan *conflictAnalysis* melakukan resolusi berikut:

- Mulai dengan klausa $c = (q \vee \neg r)$.
- Selesaikan pada r dengan *Reason*(r) = $(q \vee \neg p \vee r)$ dan $(q \vee \neg r)$, resolvennya adalah $(q \vee \neg p)$.
- Resolve pada q dengan *Reason*($\neg q$) = $(\neg q \vee \neg s \vee t)$ dan $(q \vee \neg p)$, resolvennya adalah $(\neg p \vee \neg s \vee t)$.
- Selesaikan pada p dengan *Reason*(p) = $(t \vee p)$ dan $(\neg p \vee \neg s \vee t)$, resolvennya adalah $(\neg s \vee t)$.

Klausul baru yang dihasilkan oleh *conflictAnalysis*() adalah $(\neg s \vee t)$. Klausul ini menjadi unit setelah *DPLL_T*(X) melakukan backtrack ke level = 0. *analyzeConflict*() mengembalikan klausa ini pada baris 6 dari *DPLL_T*(X) dan t akan menjadi benar oleh *B_{CP}*(U). Dibandingkan dengan Contoh 12.1.6, versi lama *DPLL_T*(X) membutuhkan dua literal keputusan untuk menemukan bahwa t harus benar; versi baru hanya membutuhkan satu.

Kombinasi Teori

Dalam verifikasi perangkat lunak, rumus seperti berikut ini muncul:

$$a \doteq b + 1 \wedge (A \doteq \text{write}(B, a + 1, 4) \vee f(a - 1) \neq f(b)) \wedge \text{read}(A, b + 2) \doteq 5$$

Untuk menangani rumus ini secara efisien, kita memerlukan prosedur pengambilan keputusan untuk teori-teori berikut:

- Teori fungsi tak terinterpretasi (T_{EUF})
- Teori aritmatika bilangan bulat linier (T_{LIA})
- Teori array (T_{arr})

T_{EUF} dan T_{LIA} telah diperkenalkan di bagian sebelumnya. Untuk T_{arr} , kita memerlukan dua operator pada array, baca dan tulis:

- *read*(A, i): membaca dari array A pada indeks i . Mengembalikan $A[i]$.
- *write*(A, i, d): menulis d ke array A pada indeks i . Mengembalikan A setelah eksekusi " $A[i] := d$."

Tiga aksioma berikut diasumsikan dalam T_{arr} :

$$\text{read}(\text{write}(A, i, d), i) \doteq d.$$

$$\begin{aligned} \text{read}(\text{write}(A, i, d), j) &\doteq \text{read}(A, j) \text{ if } i \neq j. \\ (\forall i \text{ read}(A, i) &\doteq \text{read}(B, i)) \leftrightarrow (A \doteq B) \end{aligned}$$

Aksioma terakhir disebut aksioma ekstensi, yang dibutuhkan dalam contoh ini. Dalam contoh tersebut, " $A \doteq \text{write}(B, a + 1, 4)$ " meminta untuk menunjukkan bahwa array A sama dengan array B setelah eksekusi $B[a + 1] := 4$.

Contoh 12.4.3 Kita dapat membuktikan ketidakpuasan dari

$$a \doteq b + 1 \wedge (A \doteq \text{write}(B, a + 1, 4) \vee f(a - 1) \neq f(b)) \wedge \text{read}(A, b + 2) \doteq 5$$

sebagai berikut: dengan T_{LIA} , $a - 1 \doteq b$ dan $a + 1 \doteq b + 2$ adalah benar ketika $a \doteq b + 1$ adalah benar. Dengan T_{EUF} , $f(a - 1) \neq f(b)$ adalah salah ketika $a - 1 \doteq b$ adalah benar. Dengan T_{arr} , $\text{read}(\text{write}(B, a + 1, 4), a + 1) \doteq 4$ adalah benar dan $A \doteq \text{write}(B, a + 1, 4)$ adalah salah ketika $\text{read}(\text{write}(B, a + 1, 4), a + 1) \doteq 4, \text{read}(A, b + 2) \doteq 5, a + 1 \doteq b + 2$, dan $4 = 5$ adalah benar. Karena $f(a - 1) \neq f(b)$ dan $A \doteq \text{write}(B, a + 1, 4)$ keduanya salah, maka rumus input adalah salah.

Sekarang, pertanyaannya adalah: dengan diberikan pemecah teori untuk ketiga teori individual tersebut, dapatkah kita menggabungkannya untuk mendapatkan satu pemecah untuk $(T_{LIA} \cup T_{arr} \cup T_{EUF})$? Tampaknya dari bukti contoh di atas, kita dapat memanggil pemecah untuk T_{EUF} terlebih dahulu untuk mendapatkan semua kesamaan. Namun, kesamaan baru dapat dibuat dari T_{LIA} (misalnya, $a \doteq b$ dari $a \geq b$ dan $b \geq a$) atau T_{arr} . Menggabungkan pemecah teori menjadi satu memang merupakan tugas yang sulit.

Algoritma Kombinasi Nelson dan Oppen

Nelson dan Oppen mengusulkan algoritma kombinasi yang akan menggabungkan pemecah teori di bawah beberapa kondisi yang cukup. Kita akan menggunakan contoh sebelumnya untuk mengilustrasikan ide-ide Nelson dan Oppen.

Langkah pertama adalah memurnikan input, yang merupakan himpunan literal SMT, dengan memperkenalkan konstanta antarmuka dan membagi input dengan teori.

Untuk rumus input pada Contoh 12.4.3, terdapat tiga teori dan satu kasus menggunakan kerangka kerja DPLL(T) adalah untuk memeriksa apakah konjungsi berikut dapat dipenuhi atau tidak:

$$a \doteq b + 1 \wedge f(a - 1) \neq f(b) \wedge \text{read}(A, b + 2) \doteq 5$$

Input dibagi menjadi tiga kelompok, F_1, F_2 , dan F_3 , ditambah kelompok umum CA atas himpunan C konstanta antarmuka. Misalnya, literal $(f(a - 1) \neq f(b))$ dikonversi menjadi $\{a - 1 \doteq e_1, f(e_1) \doteq e_2, f(b) \doteq e_3, e_2 \neq e_3\}$ dengan memperkenalkan tiga konstanta. Literal $(\text{read}(A, b + 2) \doteq 5)$ dikonversi menjadi $\{\text{read}(A, e_4) \doteq e_5, b + 2 \doteq e_4, 5 \doteq e_5\}$. Untuk seluruh input, kita membutuhkan tujuh konstanta antarmuka, yaitu $C = \{a, b, e_1, e_2, e_3, e_4, e_5\}$.

F_1	F_2	F_3
$b + 1 \doteq a$	$f(e_1) \doteq e_2$	$read(A, e_4) \doteq e_5$
$a - 1 \doteq e_1$	$f(b) \doteq e_3$	
$b + 2 \doteq e_4$		
$5 \doteq e_5$		

dan $CA = \{e_2 \neq e_3\}$ pada awalnya. Jelas bahwa $F_1 \cup CA$ termasuk dalam T_{LIA} , $F_2 \cup CA$ termasuk dalam T_{EUF} , dan $F_3 \cup CA$ termasuk dalam T_{arr} .

Langkah kedua dari prosedur kombinasi adalah memanggil setiap penyelesai teori pada $F_i \cup CA$ untuk (1) memeriksa apakah $F_i \cup CA$ dapat dipenuhi dan (2) menghasilkan persamaan baru di antara konstanta antarmuka dan menambahkannya ke CA .

Sebagai contoh, karena dari $b + 1 \doteq a$ kita menyimpulkan $a - 1 \doteq b$ dalam T_{LIA} , kita menambahkan $b \doteq e_1$ ke CA (karena $a - 1 \doteq b$ dan $a - 1 \doteq e_1$). Sekarang, $CA = \{e_2 = e_3, b \doteq e_1\}$.

Mudah untuk melihat bahwa $F_2 \cup CA$ tidak dapat dipenuhi dalam T_{EUF} (karena $b \doteq e_1$ menyiratkan $f(e_1) \doteq f(b)$, sehingga $e_2 \doteq e_3$). Oleh karena itu, langkah kedua berakhir dengan "tidak dapat dipenuhi."

Himpunan CA disebut susunan konstan. Jika $F_i \cup CA$ tidak dapat dipenuhi dalam teori T_i , maka inputnya tidak dapat dipenuhi. Jika tidak, kita menambahkan kesamaan baru ke dalam CA yang benar dalam T_i sampai tidak ada kesamaan baru lagi. Dalam hal ini, kita katakan CA jenuh. Ketika susunan konstan jenuh ditemukan dan untuk setiap i , $F_i \cup CA$ dapat dipenuhi dalam T_i , kita menyatakan bahwa inputnya dapat dipenuhi. Ide-ide ini dapat dijelaskan dengan prosedur berikut.

```

proc NelsonOppen( $S$ ): Boolean
  //  $S$  is a conjunction of SMT literals.
  Purify  $S$  and split  $S$  into  $F_i$ ,  $1 \leq i \leq n$ , and  $CA$ .
  while ( $true$ ) do
    for  $i := 1$  to  $n$  do
      if ( $F_i \cup CA$ ) is  $T_i$  – unsatisfiable return false
     $X := \emptyset$ 
    for  $i := 1$  to  $n$  do
      if  $F_i \models (x \doteq y)$  for  $(x \doteq y) \notin CA$ 
         $X := X \cup \{x \doteq y\}$ 
      if  $X = \emptyset$  return true //  $CA$  is saturated.
    else  $CA := CA \cup X$ 

```

Nelson dan Oppen menunjukkan bahwa prosedur di atas bekerja dengan benar di bawah dua kondisi yang cukup.

Definisi 12.4.4 Suatu teori T dikatakan stabil tak hingga jika setiap rumus bebas kuantifikasi yang dapat dipenuhi T memiliki model tak hingga.

Banyak teori menarik yang stabil tak hingga, termasuk teori struktur tak hingga (misalnya, aritmatika bilangan bulat). T_{EUF} juga dikenal stabil tak hingga. Namun, ada teori menarik yang tidak stabil tak hingga. Teori-teori tersebut adalah teori struktur terbatas, misalnya, teori vektor bit berukuran terbatas, atau aritmatika modulo n , atau teori string dengan panjang terbatas.

Definisi 12.4.5 Suatu teori T dikatakan cembung jika untuk setiap himpunan literal S , $S \models a_1 \doteq b_1 \vee \dots \vee a_n \doteq b_n$, maka $S \models a_i \doteq b_i$ untuk beberapa $1 \leq i \leq n$.

Teorema 12.4.6 Jika konjungsi literal S dapat dipisahkan menjadi teori T_i , $1 \leq i \leq n$ yang saling lepas tanda tangan, stabil tak hingga, dan cembung, maka kepuasan $(T_1 \cup \dots \cup T_n)$ dari S dapat diperiksa dengan algoritma Nelson-Oppen.

Beberapa teori populer seperti aritmatika linear bilangan bulat T_{LIA} tidak cembung. Misalnya, dalam T_{LIA} , $1 \leq x \leq 2$ setara dengan $x \doteq 1 \vee x \doteq 2$. Namun, baik T_{LIA} , $1 \leq x \leq 2 \models (x = 1)$ tidak benar, maupun T_{LIA} , $1 \leq x \leq 2 \models (x \doteq 2)$.

Upaya untuk menghilangkan kondisi tak hingga stabil atau kondisi cembung telah dilaporkan berhasil. Untuk melihat teori-teori apa saja yang terlibat dalam permasalahan SMT, tempat terbaiknya adalah situs web SMT-LIB:

smtlib.cs.uiowa.edu/index.shtml

Contoh 12.4.7 Perhatikan input $S = \{f(1) \doteq a, f(x) \doteq b, f(2) \doteq c, 1 \leq x, x \leq 2, b \doteq a - 1, b \doteq c + 1\}$. Untuk menunjukkan bahwa S tidak dapat dipenuhi, kita dapat menambahkan analisis kasus dalam algoritma kombinasi Nelson dan Oppen dengan mempertimbangkan dua kasus ketika $x \doteq 1$ bernilai benar atau salah. Detailnya diserahkan sebagai latihan.

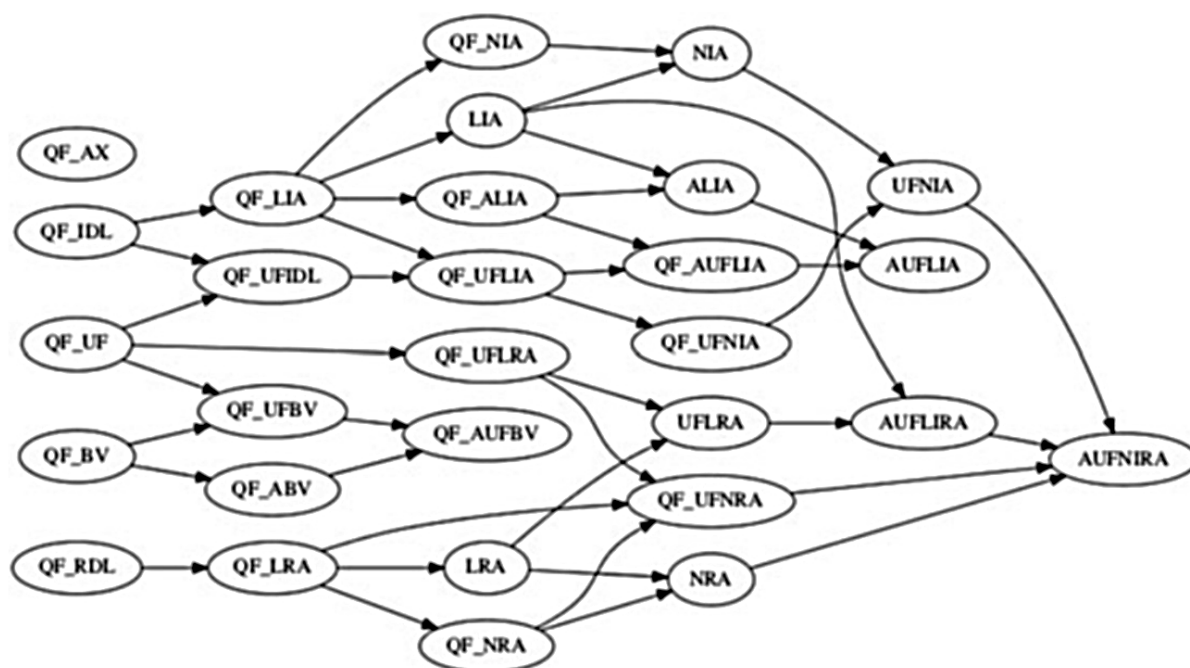
Teori kepuasan modulo, adalah inisiatif internasional yang bertujuan untuk memfasilitasi penelitian dan pengembangan di bidang SMT. Sejak didirikan pada tahun 2003, inisiatif ini telah mengejar tujuan-tujuan tersebut dengan berfokus pada tujuan-tujuan konkret berikut:

- Menyediakan deskripsi standar yang ketat tentang teori-teori latar belakang yang digunakan dalam sistem SMT.
- Mengembangkan dan mempromosikan bahasa input dan output umum untuk pemecah SMT.
- Menghubungkan pengembang, peneliti, dan pengguna SMT, dan mengembangkan komunitas di sekitarnya.
- Membangun dan menyediakan bagi komunitas penelitian perpustakaan besar benchmark untuk pemecah SMT.

- Mengumpulkan dan mempromosikan perangkat lunak yang bermanfaat bagi komunitas SMT.

Di SMT-LIB, masalah dikelompokkan berdasarkan teori, dan Gambar 12.4 menunjukkan struktur kelompok masalah. Setiap kelompok memiliki nama yang mencerminkan teori-teori yang digunakan oleh kelompok tersebut dan beberapa batasan utama dari kelompok tersebut, dengan nomenklatur berikut:

- **QF** untuk batasan pada rumus bebas kuantifikasi
- **A** atau **AX** untuk teori array
- **BV** untuk teori vektor bit ukuran tetap
- **IA** untuk teori aritmatika bilangan bulat
- **RA** untuk teori aritmatika bilangan riil
- **IRA** untuk teori aritmatika bilangan bulat dan riil campuran
- **IDL** untuk teori logika perbedaan bilangan bulat
- **UF** untuk simbol fungsi yang tidak diinterpretasikan (selain operasi dari aritmatika dan kesamaan)
- **L** sebelum **IA**, **RA**, atau **IRA** untuk fragmen linier dari aritmatika tersebut
- **N** sebelum **IA**, **RA**, atau **IRA** untuk fragmen nonlinier dari aritmatika tersebut



Gambar 12.4 SMT-LIB: Struktur kelompok masalah SMT berdasarkan teori. Panah menunjukkan relasi subgrup.

Nama-nama singkatan di atas dapat digabungkan untuk membentuk subgrup atau supergrup. Misalnya, supergrupnya adalah **AUFNIRA**, yang merupakan kombinasi dari **A** – **UF** – **N** – **IRA** dan menunjukkan kelompok masalah dari fragmen nonlinier (**N**) dari aritmatika bilangan bulat dan riil campuran (**IRA**), dengan array (**A**) dan pembatasan pada fungsi yang tidak diinterpretasikan (**UF**).

Situs web tersebut mencatat bahwa alasan penamaan ini sebagian besar bersifat historis. Misalnya, himpunan rumus bebas kuantifikasi (**QF**) adalah gabungan dari rumus dasar dan rumus non-dasar dengan variabel bebas. Namun, situs web tersebut tidak memiliki grup bernama "dasar". Bila diperlukan, situs web tersebut menggunakan kata "tertutup" dalam deskripsi suatu grup. Memang, rumus QF tertutup persis sama dengan rumus dasar.

Upaya besar SMT-LIB adalah menyediakan bahasa input/output standar untuk pemecah SMT. Para peneliti menggunakan bahasa ini untuk membuat masalah baru dan mengirimkannya ke SMT-LIB. Para pengembang pemecah SMT menggunakan SMT-LIB untuk bereksperimen dan menguji algoritma baru. SMT-LIB juga menyediakan tautan ke pemecah SMT dan alat serta utilitas terkait.

12.5 LATIHAN SOAL

1. Silakan lanjutkan eksekusi DPLL(X) secara manual pada Contoh 12.1.6 hingga model ditemukan dan catat setiap langkah eksekusinya.
2. Pada Contoh 12.2.1, aturan penulisan ulang dibuat dari

$$\{(x_0 \doteq a[3], x_1 \doteq f(a[2], x_0), x_2 \doteq f(a[1], x_1), y \doteq f(a[1], f(a[2], a[3]))\}$$

dengan mengorientasikan persamaan-persamaan tersebut dari kiri ke kanan. Tunjukkan $x_2 \doteq y$ dengan aturan penulisan ulang yang diperoleh dengan mengorientasikan persamaan-persamaan di atas dari kanan ke kiri dan sebutkan setiap langkah penulisan ulangnya.

3. Jalankan algoritma $NOS(A)$ secara manual untuk $A = \{f^3a \doteq a, f^5a \doteq a\}$. Gunakan Contoh 12.2.6 sebagai model untuk menggambarkan perubahan $up(v), h(v), use(v)$ untuk node v apa pun.
4. Jika Anda memiliki alat perangkat lunak untuk menyelesaikan program linier untuk memaksimalkan fungsi tujuan, bagaimana cara menggunakan alat ini untuk menyelesaikan program linier di mana tujuannya adalah untuk meminimalkan fungsi tujuan?
5. Diberikan himpunan E persamaan dasar dari Contoh 12.2.10

$$\{f(a) \doteq c_1, f(c_1) \doteq c_2, f(c_2) \doteq a, f(c_2) \doteq c_3, f(c_3) \doteq c_4, f(c_4) \doteq a\}$$

Silakan jalankan prosedur Knuth-Bendix pada E dengan urutan $f > a > c_1 > c_2 > c_3 > c_4$, dan daftarkan semua aturan penulisan ulang yang dibuat dari prosedur tersebut sesuai urutan pembuatannya.

6. Misalkan $E_0 = \{f(a, g(a)) \cdot g(g(a))\}$ (Contoh 12.2.9). Silakan jalankan $KBG(flattenSlicing(E_0))$ secara manual dan tunjukkan isi $nf(c), class(c)$, dan $use(c)$ untuk setiap pengidentifikasi c sebelum dan sesudah loop utama KBG .

7. Dengan asumsi setiap fungsi hash membutuhkan waktu konstan, buktikan bahwa Teorema 12.2.14 tetap benar jika operasi flatten dihilangkan. Dengan kata lain, operasi slicing sudah cukup untuk Teorema 12.2.14.
8. Matriks akhir yang diberikan pada Bagian 12.3.1 adalah

x_0	x_1	x_2	x_3	x_4	x_5	rhs
1	0	0	0	1	1	33
0	0	0	1	-4	1	4
0	0	1	0	3	-1	3
0	1	0	0	-2	1	6

Harap tuliskan sistem persamaan linear yang sesuai dalam bentuk standar dan tunjukkan langkah demi langkah bagaimana matriks awal menjadi matriks akhir melalui operasi pada matriks (seperti mengalikan satu baris dengan konstanta atau mengurangi satu baris dengan baris lainnya).

9. Harap lanjutkan eksekusi versi baru DPLLT(X) secara manual pada Contoh 12.4.2 hingga model ditemukan dan catat setiap langkah eksekusinya.
10. Harap berikan detail menjalankan algoritma kombinasi Nelson dan Oppen pada input berikut yang melibatkan T_{LIA} dan T_{EUF} :
 - a. $S_1 = \{f(f(x-1) - f(y)) \doteq a+1, f(0) > a+2, x-y \doteq 1\}$.
 - b. $S_2 = \{f(g(x-1)) \doteq x+1, f(g(y+1)) \doteq y-1, x \doteq y+2\}$.
 - c. $S_3 = \{f(x) \geq 1, f(y) \doteq 0, x < y+1, x \geq y\}$.
11. Mohon tunjukkan detail eksekusi algoritma kombinasi Nelson dan Open pada input berikut:

$$a \doteq b+1 \wedge A \doteq \text{write}(B, a+1, 4) \wedge \text{read}(A, b+2) \doteq 5$$

Mohon tambahkan analisis kasus pada algoritma kombinasi Nelson dan Open dan tunjukkan bagaimana input S terbukti tidak dapat dipenuhi (lihat Contoh 12.4.7):

12. Mohon tambahkan analisis kasus pada algoritma kombinasi Nelson dan Open dan tunjukkan bagaimana input S terbukti tidak dapat dipenuhi (lihat Contoh 12.4.7):

$$S = \{f(1) \doteq a, f(x) \doteq b, f(2) \doteq c, 1 \leq x, x \leq 2, b \doteq a-1, b \doteq c+1\}.$$

DAFTAR PUSTAKA

- Al Muhtadi, J., Alamri, R. A., Khan, F. A., & Saleem, K. (2021). Subjective logic-based trust model for fog computing. *Computer Communications*, 178, 221-233.
- Ali, H. S., Rout, R. R., Parimi, P., & Das, S. K. (2021, January). Real-time task scheduling in fog-cloud computing framework for iot applications: a fuzzy logic based approach. In *2021 International Conference on COMMunication Systems & NETworkS (COMSNETS)* (pp. 556-564). IEEE.
- Alnawayseh, S. E., Muhammad, M. H. G., Hassan, Z., Fatima, M., Aslam, M. S., Ibrahim, A., & Ateeq, K. (2023, March). Resource provisioning in cloud computing using fuzzy logic control system: An adaptive approach. In *2023 International Conference on Business Analytics for Technology and Security (ICBATS)* (pp. 1-9). IEEE.
- Barnoy, E. A., Popovtzer, R., & Fixler, D. (2020). Fluorescence for biological logic gates. *Journal of Biophotonics*, 13(9), e202000158.
- Belnap, N. D. (2020). How a computer should think. In *New essays on Belnap-Dunn logic* (pp. 35-53). Cham: Springer International Publishing.
- Carvalho, A. R., Ball, H., Biercuk, M. J., Hush, M. R., & Thomsen, F. (2021). Error-robust quantum logic optimization using a cloud quantum computer interface. *Physical Review Applied*, 15(6), 064054.
- Cheng, L., Li, Y., Yin, K. S., Hu, S. Y., Su, Y. T., Jin, M. M., ... & Miao, X. S. (2019). Functional demonstration of a memristive arithmetic logic unit (MemALU) for in-memory computing. *Advanced Functional Materials*, 29(49), 1905660.
- Cho, J. H., Okuma, A., Sofjan, K., Lee, S., Collins, J. J., & Wong, W. W. (2021). Engineering advanced logic and distributed computing in human CAR immune cells. *Nature communications*, 12(1), 792.
- Dumitrescu, C., Ciotirnae, P., & Vizitiu, C. (2021). Fuzzy logic for intelligent control system using soft computing applications. *Sensors*, 21(8), 2617.
- Fan, D., Wang, J., Wang, E., & Dong, S. (2020). Propelling DNA computing with materials' power: recent advancements in innovative DNA logic computing systems and smart bio-applications. *Advanced Science*, 7(24), 2001766.
- Gola, A., & Kłosowski, G. (2019). Development of computer-controlled material handling model by means of fuzzy logic and genetic algorithms. *Neurocomputing*, 338, 381-392.

- Jiao, S., Liu, J., Zhang, L., Yu, F., Zuo, G., Zhang, J., ... & Shao, L. (2022). All-optical logic gate computing for high-speed parallel information processing. *Opto-Electronic Science*, 1(9), 220010-1.
- Jot Singh, J., Dhawan, D., & Gupta, N. (2020). All-optical photonic crystal logic gates for optical computing: an extensive review. *Optical Engineering*, 59(11), 110901-110901.
- Kesarwani, A., & Khilar, P. M. (2022). Development of trust based access control models using fuzzy logic in cloud computing. *Journal of King Saud University-Computer and Information Sciences*, 34(5), 1958-1967.
- Kim, Y. S., Son, M. W., & Kim, K. M. (2021). Memristive stateful logic for edge Boolean computers. *Advanced Intelligent Systems*, 3(7), 2000278.
- Krivoulya, G., Ilina, I., Tokariev, V., & Shcherbak, V. (2020, October). Mathematical Model for Finding Probability of Detecting Victims of Man-Made Disasters Using Distributed Computer System with Reconfigurable Structure and Programmable Logic. In *2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T)* (pp. 573-576). IEEE.
- Lee, J., Park, B. G., & Kim, Y. (2019). Implementation of Boolean logic functions in charge trap flash for in-memory computing. *IEEE Electron Device Letters*, 40(9), 1358-1361.
- Ling, S., Zhang, C., Ma, C., Li, Y., & Zhang, Q. (2023). Emerging MXene-based memristors for in-memory, neuromorphic computing, and logic operation. *Advanced Functional Materials*, 33(1), 2208320.
- Nguyen, H. D., Sakama, C., Sato, T., & Inoue, K. (2018, October). Computing logic programming semantics in linear algebra. In *International conference on multi-disciplinary trends in artificial intelligence* (pp. 32-48). Cham: Springer International Publishing.
- Nourani, C. F. (2022). *Artificial Intelligence and Computing Logic: Cognitive Technology for AI Business Analytics*. Apple Academic Press.
- O'Meara, W., & Lennon, R. G. (2020, June). Serverless computing security: Protecting application logic. In *2020 31st Irish signals and systems conference (ISSC)* (pp. 1-5). IEEE.
- Ozsahin, D. U., Uzun, B., Ozsahin, I., Mustapha, M. T., & Musa, M. S. (2020). Fuzzy logic in medicine. In *Biomedical Signal Processing and Artificial Intelligence in Healthcare* (pp. 153-182). Academic Press.
- Priyadarshi, N., Bhoi, A. K., Sharma, A. K., Mallick, P. K., & Chakrabarti, P. (2020). An efficient fuzzy logic control-based soft computing technique for grid-tied photovoltaic system.

In *Cognitive Informatics and Soft Computing: Proceeding of CISC 2019* (pp. 131-139). Singapore: Springer Singapore.

- Qiu, C., Xiao, H., Wang, L., & Tian, Y. (2022). Recent advances in integrated optical directed logic operations for high performance optical computing: a review. *Frontiers of Optoelectronics*, 15(1), 1.
- Reuben, J. (2020). Rediscovering majority logic in the post-CMOS era: A perspective from in-memory computing. *Journal of Low Power Electronics and Applications*, 10(3), 28.
- Rezaeipannah, A., Mojarad, M., & Fakhari, A. (2022). Providing a new approach to increase fault tolerance in cloud computing using fuzzy logic. *International Journal of Computers and Applications*, 44(2), 139-147.
- Santoro, G., Turvani, G., & Graziano, M. (2019). New logic-in-memory paradigms: An architectural and technological perspective. *Micromachines*, 10(6), 368.
- Sheng, Z., Dong, J., Hu, W., Wang, Y., Sun, H., Zhang, D. W., ... & Zhang, Z. (2023). Reconfigurable logic-in-memory computing based on a polarity-controllable two-dimensional transistor. *Nano Letters*, 23(11), 5242-5249.
- Smithson, S. C., Onizawa, N., Meyer, B. H., Gross, W. J., & Hanyu, T. (2019). Efficient CMOS invertible logic using stochastic computing. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(6), 2263-2274.
- Testa, E., Soeken, M., Amar, L. G., & De Micheli, G. (2018). Logic synthesis for established and emerging computing. *Proceedings of the IEEE*, 107(1), 165-184.
- Wang, J., Wang, X., Eckert, C., Subramaniyan, A., Das, R., Blaauw, D., & Sylvester, D. (2019). A 28-nm compute SRAM with bit-serial logic/arithmetic operations for programmable in-memory vector computing. *IEEE Journal of Solid-State Circuits*, 55(1), 76-86.
- Xu, N., Park, T., Yoon, K. J., & Hwang, C. S. (2021). In-memory stateful logic computing using memristors: gate, calculation, and application. *physica status solidi (RRL)–Rapid Research Letters*, 15(9), 2100208.
- Ying, Z., Feng, C., Zhao, Z., Dhar, S., Dalir, H., Gu, J., ... & Chen, R. T. (2020). Electronic-photonic arithmetic logic unit for high-speed computing. *Nature communications*, 11(1), 2154.
- Zanotti, T., Puglisi, F. M., & Pavan, P. (2020). Smart logic-in-memory architecture for low-power non-von neumann computing. *IEEE Journal of the Electron Devices Society*, 8, 757-764.
- Zhang, H., Zhu, D., Kang, W., Zhang, Y., & Zhao, W. (2020). Stochastic computing implemented by skyrmionic logic devices. *Physical Review Applied*, 13(5), 054049.

LOGIKA KOMPUTER

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

BIO DATA PENULIS



Penulis memiliki berbagai disiplin ilmu yang diperoleh dari Universitas Diponegoro (UNDIP) Semarang. dan dari Universitas Kristen Satya Wacana (UKSW) Salatiga. Disiplin ilmu itu antara lain teknik elektro, komputer, manajemen dan ilmu sosiologi. Penulis memiliki pengalaman kerja pada industri elektronik dan sertifikasi keahlian dalam bidang Jaringan Internet, Telekomunikasi, Artificial Intelligence, Internet Of Things (IoT), Augmented Reality (AR), Technopreneurship, Internet Marketing dan bidang pengolahan dan analisa data (komputer statistik).

Penulis adalah pendiri dari Universitas Sains dan Teknologi Komputer (Universitas STEKOM) dan juga seorang dosen yang memiliki Jabatan Fungsional Akademik Lektor Kepala (Associate Professor) yang telah menghasilkan puluhan Buku Ajar ber ISBN, HAKI dari beberapa karya cipta dan Hak Paten pada produk IPTEK. Sejak tahun 2023 penulis tercatat sebagai Dosen luar biasa di Fakultas Ekonomi & Bisnis (FEB) Universitas Diponegoro Semarang. Penulis juga terlibat dalam berbagai organisasi profesi dan industri yang terkait dengan dunia usaha dan industri, khususnya dalam pengembangan sumber daya manusia yang unggul untuk memenuhi kebutuhan dunia kerja secara nyata.



YAYASAN PRIMA AGUS TEKNIK

PENERBIT :

YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-634-7695-16-1 (PDF)



9

786347

695161