Paradigma Pemrograman untuk Dummies: What Every Programmer Harus Tahu

Peter Van Roy

Bab ini memberikan pengantar untuk semua paradigma pemrograman utama, konsep dasarnya, dan hubungan di antara mereka. Kami memberikan pandangan luas untuk membantu programmer memilih konsep yang tepat yang mereka butuhkan untuk menyelesaikan masalah yang dihadapi. Kami memberikan taksonomi dari hampir 30 paradigma pemrograman yang berguna dan bagaimana mereka terkait. Kebanyakan dari mereka hanya berbeda dalam satu atau beberapa konsep, tapi ini bisa membuat dunia berbeda dalam pemrograman. Kami menjelaskan secara singkat bagaimana paradigma pemrograman memengaruhi desain bahasa, dan kami menunjukkan dua titik manis: bahasa paradigma ganda dan bahasa definitif. Kami memperkenalkan konsep utama bahasa pemrograman: catatan, penutupan, independensi (konkurensi), dan status bernama. Kami menjelaskan prinsip utama abstraksi data dan bagaimana hal itu memungkinkan kami mengatur program besar. Akhirnya, kami menyimpulkan dengan fokus pada konkurensi, yang secara luas dianggap sebagai konsep yang paling sulit untuk diprogram. Kami menyajikan empat paradigma yang kurang diketahui tetapi penting yang sangat menyederhanakan pemrograman konkuren sehubungan dengan bahasa utama: konkurensi deklaratif (baik yang lebih mudah maupun yang malas), pemrograman reaktif fungsional, pemrograman sinkron diskrit, dan pemrograman batasan. Paradigma ini tidak memiliki kondisi ras dan dapat digunakan dalam kasus di mana tidak ada paradigma lain yang berfungsi. Kami menjelaskan mengapa untuk prosesor multi-core dan kami memberikan beberapa contoh dari musik komputer, yang sering menggunakan paradigma ini. pemrograman sinkron diskrit, dan pemrograman kendala. Paradigma ini tidak memiliki kondisi ras dan dapat digunakan dalam kasus di mana tidak ada paradigma lain yang berfungsi. Kami menjelaskan mengapa untuk prosesor multi-core dan kami memberikan beberapa contoh dari musik komputer, yang sering menggunakan paradigma ini. pemrograman sinkron diskrit, dan pemrograman kendala. Paradigma ini tidak memiliki kondisi ras dan dapat digunakan dalam kasus di mana tidak ada paradigma lain yang berfungsi. Kami menjelaskan mengapa untuk prosesor multi-core dan kami memberikan beberapa contoh dari musik komputer, yang sering menggunakan paradigma ini.

Lebih banyak tidak lebih baik (atau lebih buruk) daripada kurang, hanya saja berbeda.
- Paradoks paradigma.

1. Perkenalan

Pemrograman adalah disiplin yang kaya dan bahasa pemrograman praktis biasanya cukup rumit. Untungnya, ide-ide penting bahasa pemrograman itu sederhana. Bab ini dimaksudkan untuk memberikan pengalaman pemrograman kepada pembaca untuk memulai ide-ide ini. Meskipun kami tidak memberikan definisi formal, kami memberikan intuisi yang tepat dan referensi yang baik sehingga pembaca yang tertarik dapat dengan cepat mulai menggunakan konsep dan bahasa yang menerapkannya. Kami menyebutkan *semua* paradigma penting tetapi kami menyukai beberapa paradigma yang kurang diketahui yang pantas untuk digunakan lebih luas. Kami sengaja mengabaikan penjelasan rinci tentang beberapa paradigma yang lebih terkenal

(seperti pemrograman fungsional dan berorientasi objek), karena mereka sudah memiliki literatur yang sangat besar.

Memecahkan masalah pemrograman membutuhkan pemilihan konsep yang tepat. Semua masalah kecuali mainan terkecil memerlukan rangkaian konsep yang berbeda untuk bagian yang berbeda. Inilah mengapa bahasa pemrograman harus mendukung banyak paradigma. SEBUAH *paradigma pemrograman* adalah pendekatan untuk memprogram komputer berdasarkan teori matematika atau sekumpulan prinsip yang koheren. Setiap paradigma mendukung sekumpulan konsep yang menjadikannya yang terbaik untuk jenis masalah tertentu. Misalnya, pemrograman berorientasi objek paling baik untuk masalah dengan sejumlah besar abstraksi data terkait yang diatur dalam hierarki. Pemrograman logika paling baik untuk mengubah atau menavigasi struktur simbolik yang kompleks sesuai dengan aturan logis. Pemrograman sinkronis diskrit paling baik untuk masalah reaktif, yaitu masalah yang terdiri dari reaksi terhadap rangkaian peristiwa eksternal. Bahasa yang mendukung ketiga paradigma tersebut masing-masing adalah Java, Prolog, dan Esterel.

Bahasa arus utama populer seperti Java atau C ++ hanya mendukung satu atau dua paradigma terpisah. Hal ini sangat disayangkan, karena masalah pemrograman yang berbeda memerlukan konsep pemrograman yang berbeda untuk menyelesaikannya dengan rapi, dan satu atau dua paradigma tersebut seringkali tidak mengandung konsep yang benar. Sebuah bahasa idealnya mendukung banyak konsep dengan cara yang difaktorkan dengan baik, sehingga pemrogram dapat memilih konsep yang tepat kapan pun mereka dibutuhkan tanpa dibebani oleh yang lain. Gaya pemrograman ini terkadang disebut *multiparadigma* pemrograman, menyiratkan bahwa itu adalah sesuatu yang eksotis dan di luar kebiasaan. Sebaliknya, menurut pengalaman kami, jelas bahwa itu harus menjadi cara pemrograman yang normal. Bahasa arus utama masih jauh dari mendukung ini. Meskipun demikian, memahami konsep yang tepat dapat membantu meningkatkan gaya pemrograman bahkan dalam bahasa yang tidak mendukungnya secara langsung, sama seperti pemrograman berorientasi objek dimungkinkan dalam C dengan sikap programmer yang tepat.

Bab ini sebagian didasarkan pada buku [50], yang dikenal sebagai CTM, yang memberikan lebih banyak informasi tentang banyak paradigma dan konsep yang disajikan di sini. Tapi bab ini melangkah lebih jauh dan menyajikan ide dan paradigma yang tidak tercakup dalam CTM. Contoh kode dalam bab ini ditulis dalam bahasa Oz, yang juga digunakan dalam CTM. Keunggulan Oz mendukung multi paradigma dengan baik, sehingga kita tidak perlu memasukkan lebih dari satu notasi. Contoh-contoh ini harus cukup jelas; setiap kali terjadi sesuatu yang tidak biasa, kami menjelaskannya dalam teks.

Isi bab ini

Bahasa, paradigma, dan konsep Bagian 2 menjelaskan apa itu paradigma pemrograman dan memberikan taksonomi dari paradigma utama. Jika pengalaman Anda terbatas pada satu atau hanya beberapa bahasa atau paradigma pemrograman (misalnya, pemrograman berorientasi objek di Java), maka Anda akan menemukan sudut pandang yang lebih luas di sini. Kami juga menjelaskan bagaimana kami mengatur paradigma untuk menunjukkan bagaimana mereka terkait. Kami menemukan bahwa tidak benar bahwa hanya ada satu paradigma "terbaik", dan fortiori ini bukanlah pemrograman berorientasi objek! Sebaliknya, ada banyak paradigma yang bermanfaat. Setiap paradigma memiliki tempatnya: masing-masing memiliki masalah yang memberikan solusi terbaik (paling sederhana, termudah untuk dipikirkan, atau paling efisien). Karena sebagian besar program harus memecahkan lebih dari satu masalah, maka program tersebut paling baik ditulis dalam paradigma yang berbeda.

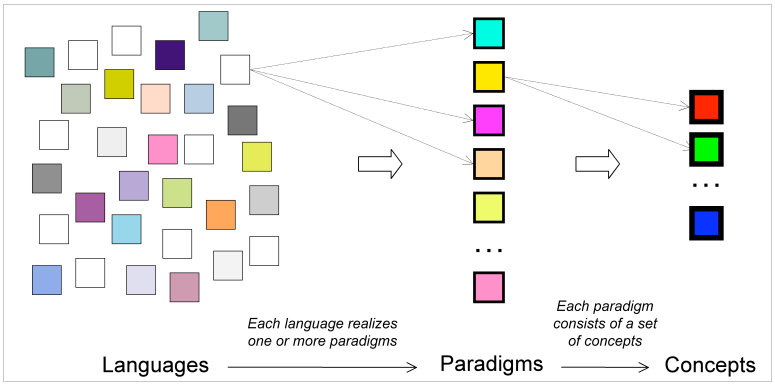
Merancang bahasa dan programnya Bagian 3 menjelaskan bagaimana merancang bahasa untuk mendukung beberapa paradigma. Bahasa yang baik untuk program besar harus mendukung beberapa paradigma. Salah satu pendekatan yang bekerja dengan sangat baik adalah *paradigma ganda* bahasa: bahasa yang mendukung satu paradigma untuk pemrograman dalam skala kecil dan paradigma lain untuk pemrograman dalam skala besar. Pendekatan lain adalah ide merancang a *definitif* bahasa. Kami menyajikan contoh desain yang telah terbukti dalam empat bidang yang berbeda. Desainnya memiliki struktur berlapis dengan satu paradigma di setiap lapisan. Setiap paradigma dipilih dengan hati-hati untuk menyelesaikan masalah yang muncul berturut-turut. Kami menjelaskan mengapa desain ini bagus untuk membuat perangkat lunak berskala besar.

Konsep pemrograman Bagian 4 menjelaskan empat konsep terpenting dalam pemrograman: catatan, penutupan yang dibatasi secara leksikal, independensi (konkurensi), dan status bernama. Masing-masing konsep ini memberikan pemrogram ekspresi esensial yang tidak dapat diperoleh dengan cara lain. Konsep-konsep ini sering digunakan dalam paradigma pemrograman.

Abstraksi data Bagian 5 menjelaskan bagaimana mendefinisikan bentuk data baru dengan operasinya dalam sebuah program. Kami menunjukkan empat jenis abstraksi data: objek dan tipe data abstrak adalah dua yang paling populer, tetapi ada dua lainnya, objek deklaratif dan tipe data abstrak berstatus. Abstraksi data memungkinkan untuk mengatur program menjadi bagian-bagian yang dapat dimengerti, yang penting untuk kejelasan, pemeliharaan, dan skalabilitas. Hal ini memungkinkan untuk meningkatkan ekspresi bahasa dengan mendefinisikan bahasa baru di atas bahasa yang sudah ada. Ini membuat abstraksi data menjadi bagian penting dari kebanyakan paradigma.

Pemrograman konkuren deterministik Bagian 6 menyajikan pemrograman konkuren deterministik, model konkuren yang memperdagangkan ekspresi untuk kemudahan pemrograman. ini *banyak* lebih mudah untuk memprogram daripada paradigma konkuren biasa, yaitu konkurensi status bersama dan konkurensi penyampaian pesan. Ini juga merupakan cara termudah untuk menulis program paralel, yaitu program yang berjalan pada banyak prosesor seperti prosesor multi-inti. Kami menyajikan tiga paradigma penting dari konkurensi deterministik yang pantas untuk lebih dikenal. Harga untuk menggunakan konkurensi deterministik adalah bahwa program tidak dapat mengekspresikan nondeterminisme, yaitu, di mana eksekusi tidak sepenuhnya ditentukan oleh spesifikasi. Misalnya, aplikasi klien / server dengan dua klien tidak bersifat deterministik karena server tidak tahu dari klien mana perintah berikutnya akan datang. Ketidakmampuan untuk mengekspresikan nondeterminisme di dalam program seringkali tidak relevan, karena nondeterminisme juga tidak diperlukan, berasal dari luar program, atau dapat dibatasi pada sebagian kecil program. Dalam aplikasi klien / server, hanya komunikasi dengan server yang nondeterministic. Implementasi klien dan server sendiri dapat sepenuhnya bersifat deterministik.

Pemrograman kendala Bagian 7 menyajikan paradigma paling deklaratif dari taksonomi kita, dalam arti asli deklaratif: memberi tahu komputer apa yang dibutuhkan alih-alih bagaimana menghitungnya. Paradigma ini memberikan abstraksi tingkat tinggi untuk menyelesaikan masalah dengan kondisi global. Ini telah digunakan di masa lalu untuk masalah kombinatorial, tetapi juga dapat digunakan untuk aplikasi yang lebih umum seperti komposisi dengan bantuan komputer. Pemrograman kendala telah mencapai tingkat kematangan yang tinggi sejak itu



Gambar 1. Bahasa, paradigma, dan konsep

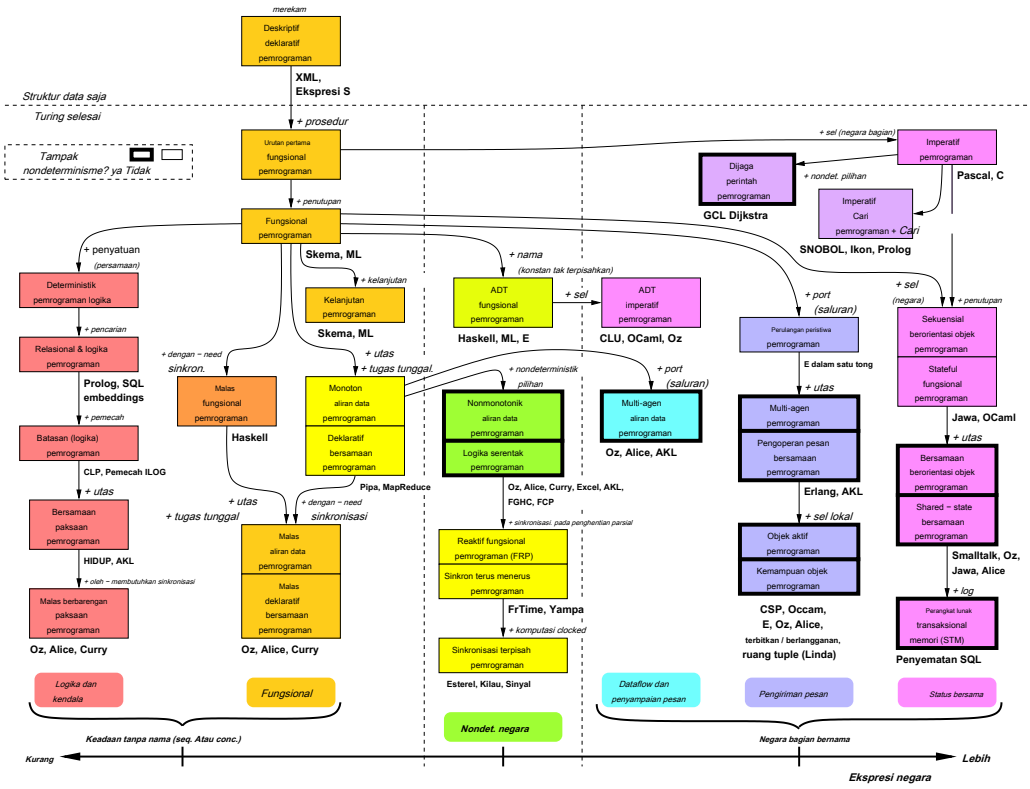
berasal dari tahun 1970-an. Ia menggunakan algoritma canggih untuk menemukan solusi yang memenuhi kondisi global. Ini berarti bahwa ia benar-benar memenuhi klaimnya yang ambisius.

Kesimpulan dan saran untuk melangkah lebih jauh Bagian 8 menyimpulkan dengan mengulangi mengapa bahasa pemrograman harus mendukung beberapa paradigma. Untuk memahami "jiwa" dari setiap paradigma dan untuk mendapatkan pengalaman pemrograman dengan paradigma yang berbeda, kami merekomendasikan penggunaan bahasa multiparadigma. Bahasa multiparadigma memungkinkan pemrograman di setiap paradigma tanpa gangguan dari paradigma lain. Dua bahasa multiparadigma yang paling luas adalah bahasa yang diketik secara dinamis Oz [50] dan bahasa yang diketik secara statis, Alice [38].

2 Bahasa, paradigma, dan konsep

Bagian ini memberikan gambaran besar tentang paradigma pemrograman, bahasa yang merealisasikannya, dan konsep yang dikandungnya. Ada banyak paradigma pemrograman yang lebih sedikit daripada bahasa pemrograman. Itulah mengapa menarik untuk fokus pada paradigma daripada bahasa. Dari sudut pandang ini, bahasa-bahasa seperti Java, Javascript, C #, Ruby, dan Python semuanya hampir identik: semuanya menerapkan paradigma berorientasi objek dengan hanya perbedaan kecil, setidaknya dari sudut pandang paradigma.

Gambar 1 menunjukkan jalur dari bahasa ke paradigma dan konsep. Setiap bahasa pemrograman menyadari satu atau lebih paradigma. Setiap paradigma didefinisikan oleh sekumpulan konsep pemrograman, diatur ke dalam bahasa inti sederhana yang disebut paradigma *bahasa ker- nel*. Ada banyak sekali bahasa pemrograman, tetapi paradigma yang lebih sedikit. Tapi masih banyak paradigma. Bab ini menyebutkan 27 paradigma berbeda yang sebenarnya digunakan. Semuanya memiliki implementasi yang baik dan aplikasi praktis. Untungnya, paradigma bukanlah pulau: mereka memiliki banyak kesamaan. Kami menyajikan taksonomi yang menunjukkan bagaimana paradigma terkait.



Gambar 2. Taksonomi paradigma pemrograman

2.1 Taksonomi paradigma pemrograman

Gambar 2 memberikan taksonomi dari semua paradigma pemrograman utama, diatur dalam grafik yang menunjukkan bagaimana mereka terkait [55]. Gambar ini mengandung banyak informasi dan berkaitan dengan pemeriksaan yang cermat. Ada 27 kotak, masing-masing mewakili paradigma sebagai sekumpulan konsep pemrograman. Dari 27 kotak ini, delapan berisi dua paradigma dengan nama yang berbeda tetapi konsep yang sama. Sebuah panah di antara dua kotak mewakili konsep atau konsep yang harus ditambahkan untuk berpindah dari satu paradigma ke paradigma berikutnya. Konsep adalah elemen primitif dasar yang digunakan untuk membangun paradigma. Seringkali dua paradigma yang tampaknya cukup berbeda (misalnya, pemrograman fungsional dan program berorientasi objek) berbeda hanya dengan satu konsep. Dalam bab ini kami fokus pada konsep pemrograman dan bagaimana paradigma muncul darinya. Dengan n konsep, secara teoritis mungkin untuk membangun 2^n paradigma. Tentu saja, banyak dari paradigma ini yang tidak berguna dalam praktiknya, seperti paradigma kosong (tanpa konsep).[1] atau paradigma dengan hanya satu konsep. Paradigma harus lengkap Turing agar praktis. Ini menjelaskan mengapa pemrograman fungsional sangat penting: itu didasarkan pada konsep fungsi kelas satu,

[1] Alasan serupa menjelaskan mengapa Baskin-Robbins memiliki tepat 31 rasa es krim. Kami mendalilkan bahwa mereka hanya memiliki 5 rasa, yang menghasilkan $2^5 - 1 = 31$ kombinasi dengan setidaknya satu rasa. 32nd kombinasi adalah rasa kosong. Rasa dari rasa kosong adalah pertanyaan penelitian terbuka.

atau *penutupan,* yang membuatnya setara dengan file λ- kalkulus yang Turing lengkap. Dari 2^n kemungkinan paradigma, jumlah paradigma praktis berguna jauh lebih kecil. Tapi itu masih jauh lebih besar dari *n.*

Ketika suatu bahasa disebutkan di bawah paradigma pada Gambar 2, itu berarti bahwa bagian dari bahasa tersebut dimaksudkan (oleh perancangnya) untuk mendukung paradigma tersebut tanpa campur tangan dari paradigma lain. Ini tidak berarti bahwa ada kesesuaian yang sempurna antara bahasa dan paradigma. Tidaklah cukup bahwa perpustakaan telah ditulis dalam bahasa untuk mendukung paradigma tersebut. Bahasa kernel bahasa harus mendukung paradigma. Ketika ada rumpun bahasa terkait, biasanya hanya satu anggota rumpun tersebut yang disebutkan untuk menghindari kekacauan. Tidak adanya bahasa tidak menyiratkan penilaian nilai apa pun. Ada terlalu banyak bahasa yang bagus untuk disebutkan semuanya.

Gambar 2 menunjukkan dua sifat penting dari paradigma: apakah mereka memiliki nondeterminisme yang dapat diamati atau tidak dan seberapa kuat mereka mendukung negara. Sekarang kita membahas masing-masing properti ini secara bergiliran.

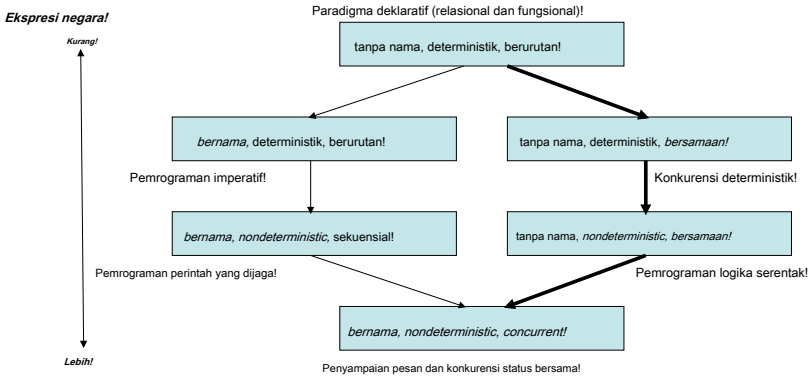
Nondeterminisme yang dapat diamati

Properti kunci pertama dari sebuah paradigma adalah apakah ia dapat mengekspresikan non-terminisme yang dapat diamati atau tidak. Ini diidentifikasikan pada Gambar 2 dengan kotak dengan batas tebal atau tipis. Kita ingat bahwa nondeterminisme adalah ketika eksekusi sebuah program tidak sepenuhnya ditentukan oleh spesifikasinya, yaitu, pada titik tertentu selama eksekusi, spesifikasi memungkinkan program untuk memilih apa yang akan dilakukan selanjutnya. Selama eksekusi, pilihan ini dibuat oleh bagian dari sistem run-time yang disebut *penjadwal.* Non-determinisme adalah *tampak* jika seorang pengguna dapat melihat hasil yang berbeda dari eksekusi yang dimulai pada konfigurasi internal yang sama. Ini sangat tidak diinginkan. Efek tipikal adalah a *kondisi balapan,* di mana hasil dari sebuah program bergantung pada perbedaan waktu yang tepat antara bagian-bagian yang berbeda dari sebuah program (sebuah "perlombaan"). Ini dapat terjadi jika pengaturan waktu memengaruhi pilihan yang dibuat oleh penjadwal. Tetapi paradigma yang memiliki kekuatan untuk mengekspresikan nondeterminisme yang dapat diamati dapat digunakan untuk memodelkan situasi dunia nyata dan untuk memprogram aktivitas independen.

Kami menyimpulkan bahwa nondeterminisme yang dapat diamati harus didukung hanya jika kekuatan ekspresifnya diperlukan. Ini terutama berlaku untuk pemrograman bersamaan. Misalnya, bahasa Java dapat mengekspresikan nondeterminisme yang dapat diamati karena ia memiliki nama negara dan konkurensi (lihat di bawah). Hal ini membuat pemrograman konkuren di Java cukup sulit [29]. Pemrograman bersamaan jauh lebih mudah dengan paradigma konkuren deklaratif, di mana semua program bersifat deterministik. Bagian 6 dan 7 menyajikan empat paradigma konkuren penting yang tidak memiliki nondeterminisme yang dapat diamati.

Negara bagian bernama

Properti kunci kedua dari sebuah paradigma adalah seberapa kuat ia mendukung negara. Negara adalah kemampuan untuk mengingat informasi, atau lebih tepatnya, untuk menyimpan urutan nilai pada waktunya. Kekuatan ekspresifnya sangat dipengaruhi oleh paradigma yang memuatnya. Kami membedakan tiga sumbu ekspresif, tergantung pada apakah negara tidak bernama atau dinamai, deterministik atau nondeterministik, dan berurutan atau bersamaan. Ini menghasilkan delapan kombinasi secara keseluruhan. Nanti di bab ini kami memberikan contoh dari banyak kombinasi ini. Tidak semua kombinasi berguna. Gambar 3 menunjukkan beberapa yang berguna diatur dalam kisi;



Gambar 3. Tingkat dukungan yang berbeda untuk negara

kotak yang berdekatan berbeda dalam satu koordinat. [2] Satu kotak menarik yang ditampilkan adalah bahasa perintah yang dijaga Dijkstra (GCL) [14]. Ini telah menamai pilihan negara dan nondeterministik dalam bahasa berurutan. Ini menggunakan pilihan nondeterministic untuk menghindari algoritma yang menentukan spesifikasi berlebihan (terlalu banyak bicara tentang bagaimana mereka harus mengeksekusi).

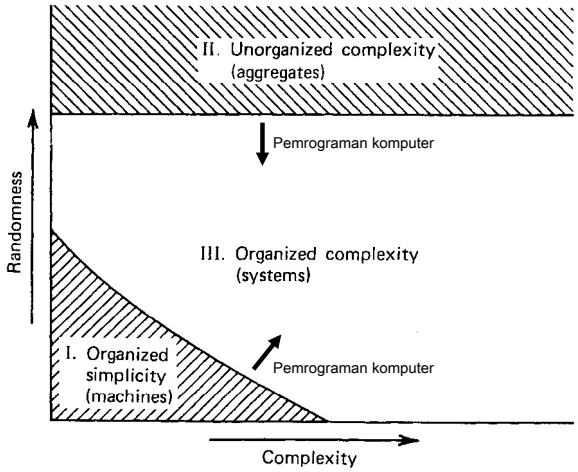
Paradigma pada Gambar 2 diklasifikasikan pada sumbu horizontal menurut seberapa kuat mereka mendukung status. Sumbu horizontal ini sesuai dengan garis tebal pada Gambar 3. Mari kita ikuti garis dari atas ke bawah. Kombinasi yang paling tidak ekspresif adalah pemrograman fungsional (status ulir, misalnya, DCG dalam Prolog dan monad dalam pemrograman fungsional: tanpa nama, deterministik, dan sekuensial). Menambahkan konkurensi memberikan pemrograman konkurensi deklaratif (misalnya, sel sinkron: tidak bernama, deterministik, dan konkuren). Menambahkan pilihan nondeterministik memberikan pemrograman logika bersamaan (yang menggunakan penggabungan aliran: tanpa nama, nondeterministik, dan bersamaan). Menambahkan port atau sel, masing-masing, memberikan pesan yang lewat atau status bersama (keduanya bernama, nondeterministic, dan concurrent). Nondeterminisme penting untuk interaksi dunia nyata (misalnya, klien / server).

Baik nondeterminisme yang dapat diamati maupun keadaan bernama adalah kasus di mana penting untuk memilih paradigma yang cukup ekspresif, tetapi tidak terlalu ekspresif (lihat epigram di kepala bab). Masing-masing dari dua konsep ini terkadang diperlukan tetapi harus ditinggalkan jika tidak diperlukan. Intinya adalah memilih paradigma dengan konsep yang tepat. Terlalu sedikit dan program menjadi rumit. Terlalu banyak dan penalaran menjadi rumit. Kami akan memberikan banyak contoh dari asas ini di sepanjang bab ini.

2.2 Pemrograman komputer dan desain sistem

Gambar 4 memberikan tampilan pemrograman komputer dalam konteks desain sistem umum. Gambar ini menambahkan pemrograman komputer ke diagram yang diambil dari Weinberg [56]. Dua sumbu mewakili sifat utama sistem: kompleksitas (jumlah komponen dasar yang berinteraksi) dan keacakan (seberapa nondeterministik perilaku sistem). Ada dua jenis sistem yang dipahami oleh sains: agregat (mis., Gas

[2] Dua dari delapan kemungkinan kombinasi tidak ditampilkan pada gambar. Kami serahkan kepada pembaca untuk menemukannya dan mencari tahu apakah itu masuk akal!



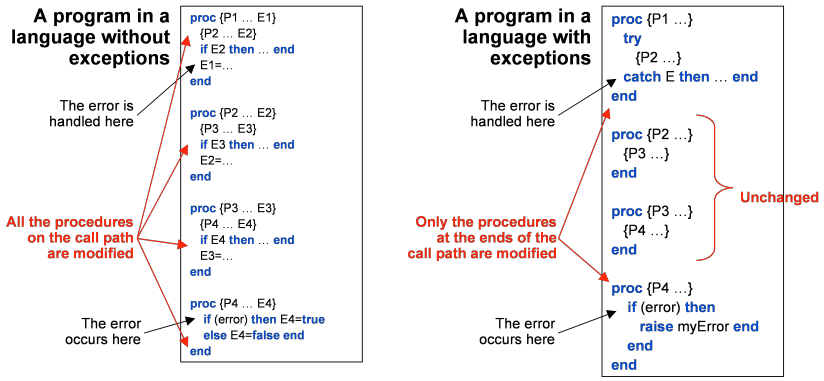
Gambar 4. Pemrograman komputer dan desain sistem (diadaptasi dari Weinberg [56])

molekul dalam kotak, dipahami oleh mekanika statistik) dan mesin (misalnya, jam dan mesin cuci, sejumlah kecil komponen yang berinteraksi dengan cara yang sebagian besar bersifat deterministik). Area putih besar di tengah sebagian besar tidak dipahami. Ilmu pemrograman komputer mendorong ke dalam dua batas ilmu sistem: program komputer dapat bertindak sebagai mesin yang sangat kompleks dan juga sebagai agregat melalui simulasi. Pemrograman komputer memungkinkan pembangunan sistem yang paling kompleks.

Bahasa pemrograman modern telah berkembang selama lebih dari lima dekade pengalaman dalam membangun solusi terprogram untuk masalah dunia nyata yang kompleks. Program modern bisa sangat kompleks, mencapai ukuran yang diukur dalam jutaan baris kode sumber, ditulis oleh tim program yang besar selama bertahun-tahun. Dalam pandangan kami, bahasa yang berskala ke tingkat kompleksitas ini sebagian berhasil karena mereka memodelkan beberapa faktor penting tentang bagaimana membangun sistem yang kompleks. Dalam pengertian ini, bahasa-bahasa ini bukan hanya konstruksi pikiran manusia yang sewenang-wenang. Mereka mengeksplorasi batasan kompleksitas dengan cara yang lebih obyektif. Oleh karena itu kami ingin memahaminya dengan cara ilmiah, yaitu dengan memahami konsep dasar yang menyusun paradigma yang mendasari dan bagaimana konsep-konsep ini dirancang dan digabungkan.

2.3 Prinsip penyuluhan kreatif

Konsep tidak digabungkan secara sembarangan untuk membentuk paradigma. Mereka dapat diatur menurut *prinsip penyuluhan kreatif*. Prinsip ini pertama kali didefinisikan oleh Felleisen [18] dan secara independen ditemukan kembali [50]. Ini memberi kita panduan untuk menemukan keteraturan dalam kumpulan luas kemungkinan paradigma. Dalam paradigma tertentu, dapat terjadi bahwa program menjadi rumit karena alasan teknis yang tidak memiliki hubungan langsung dengan masalah spesifik yang sedang dipecahkan. Ini pertanda bahwa ada konsep baru yang menunggu untuk ditemukan. Untuk menunjukkan bagaimana prinsip bekerja, asumsikan kita memiliki paradigma pemrograman fungsional sekuensial sederhana. Berikut adalah tiga skenario bagaimana konsep baru dapat ditemukan dan ditambahkan untuk membentuk paradigma baru:



Gambar 5. Bagaimana menambahkan pengecualian ke suatu bahasa dapat menyederhanakan program

- Jika kita perlu memodelkan beberapa aktivitas independen, maka kita harus mengimplementasikan beberapa tumpukan eksekusi, penjadwal, dan mekanisme untuk mendahului eksekusi dari satu aktivitas ke aktivitas lainnya. Semua kerumitan ini tidak diperlukan jika kita menambahkan satu konsep ke bahasa: *konkurensi*.

- Jika kita perlu memodelkan memori yang dapat diupdate, yaitu entitas yang mengingat dan memperbarui masa lalunya, maka kita harus menambahkan dua argumen ke semua pemanggilan fungsi relatif terhadap entitas itu. Argumen mewakili nilai input dan output dari memori. Ini berat dan juga tidak modular karena memori bergerak ke seluruh program. Semua kecanggungan ini tidak diperlukan jika kita menambahkan satu konsep ke bahasa: *negara bernama*.

- Jika kita perlu memodelkan deteksi dan koreksi kesalahan, di mana setiap fungsi dapat mendeteksi kesalahan kapan saja dan mentransfer kontrol ke rutinitas koreksi kesalahan, maka kita perlu menambahkan kode kesalahan ke semua output fungsi dan persyaratan untuk menguji semua panggilan fungsi untuk dikembalikan kode kesalahan. Semua kerumitan ini tidak diperlukan jika kita menambahkan satu konsep ke bahasa: *pengecualian*. Gambar 5 menunjukkan cara kerjanya.

Tema umum dalam ketiga skenario ini (dan banyak lainnya!) Adalah bahwa kita perlu melakukan modifikasi program secara pervasif (nonlokal) untuk menangani konsep baru. Jika kebutuhan akan modifikasi yang meluas terwujud dengan sendirinya, kita dapat menganggapnya sebagai tanda bahwa ada konsep baru yang menunggu untuk ditemukan. Dengan menambahkan konsep ini ke bahasa, kami tidak lagi membutuhkan modifikasi yang meluas ini dan kami memulihkan kesederhanaan program. Kompleksitas satu-satunya dalam program adalah yang dibutuhkan untuk memecahkan masalah. Tidak ada kerumitan tambahan yang diperlukan untuk mengatasi kekurangan teknis bahasa tersebut. Baik Gambar 2 dan [50] diatur menurut prinsip penyuluhan materi iklan.

3 Merancang bahasa dan programnya

Bahasa pemrograman tidak dirancang dalam ruang hampa, tetapi untuk memecahkan jenis masalah tertentu. Setiap masalah memiliki paradigma yang terbaik untuk itu. Tidak ada satu paradigma yang terbaik untuk semua masalah. Oleh karena itu, penting untuk memilih dengan cermat paradigma yang didukung oleh

bahasa. Kami akan melihat dua kasus menarik: bahasa yang mendukung dua paradigma (Bagian 3.1) dan bahasa berlapis (Bagian 3.2). Bahasa berlapis yang kami sajikan sangat menarik karena struktur berlapis yang hampir sama muncul di empat area yang berbeda.

3.1 Bahasa yang mendukung dua paradigma

Banyak bahasa mendukung dua paradigma, biasanya satu untuk pemrograman dalam skala kecil dan satu lagi untuk pemrograman dalam skala besar. Paradigma pertama dipilih untuk jenis masalah yang paling sering menjadi sasaran bahasa. Paradigma kedua dipilih untuk mendukung abstraksi dan modularitas dan digunakan saat menulis program besar. Berikut beberapa contoh:

- *Prolog:* Paradigma pertama adalah mesin pemrograman logika yang didasarkan pada penyatuan dan pencarian kedalaman pertama. Paradigma kedua adalah keharusan: operasi assert dan retract yang memungkinkan program untuk menambah dan menghapus klausa program. Prolog berasal dari tahun 1972, yang menjadikannya bahasa lama. Perkembangan terkini dalam bahasa pemodelan berdasarkan algoritma pencarian lanjutan memajukan kedua sisi pemrograman logika dan pemrograman imperatif. Implementasi Prolog modern telah menambahkan beberapa kemajuan ini, misalnya, dukungan untuk pemrograman kendala dan sistem modul.

- *Bahasa pemodelan (misalnya, Comet, Numerica [48]):* Paradigma pertama adalah pemecah: pemrograman kendala (lihat Bagian 7), pencarian lokal (lihat bab oleh Philippe Codognet [8]), kepuasan (pemecah SAT), dan sebagainya. Paradigma kedua adalah pemrograman berorientasi objek.

- *Memecahkan perpustakaan (mis., Gecode):* Paradigma pertama adalah pustaka solver berdasarkan algoritma pencarian lanjutan, seperti Gecode [43, 47]. Paradigma kedua ditambahkan oleh bahasa host, misalnya, C ++ dan Java mendukung pemrograman berorientasi objek.

- *Penyematan bahasa (misalnya, SQL):* SQL sudah mendukung dua paradigma: mesin pemrograman relasional untuk kueri logis dari database dan antarmuka transaksional untuk pembaruan database secara bersamaan. Bahasa host melengkapi ini dengan mendukung pemrograman berorientasi objek, untuk organisasi program besar. Contoh ini melampaui dua paradigma untuk menunjukkan desain dengan tiga paradigma yang saling melengkapi.

3.2 Bahasa pemrograman yang pasti

Pada suatu saat, penelitian bahasa akan memberikan solusi yang cukup baik sehingga peneliti akan beralih ke pekerjaan pada tingkat abstraksi yang lebih tinggi. Ini telah tersedia untuk banyak subarea desain bahasa, seperti bahasa assembly dan algoritma parsing. Pada tahun 1970-an, kursus kompilator dibuat berdasarkan studi tentang algoritme parsing. Saat ini, penguraian dipahami dengan baik untuk sebagian besar tujuan praktis dan desain kompiler telah berpindah. Kursus penyusun hari ini dibangun di sekitar topik tingkat yang lebih tinggi seperti analisis aliran data, sistem tipe, dan konsep bahasa. Kami mendalilkan bahwa jenis evolusi ini terjadi dengan desain bahasa juga.

Lapisan	Proyek bahasa			
	Erlang [6, 5]	E [32, 31]	Distrib. Oz [10]	Didactic Oz [50]
Fungsional pemrograman (lihat Bagian 4.2)	Sebuah proses adalah fungsi rekursif di utasnya sendiri, menggunakan clo- jaminan untuk pembaruan kode panas	Objek adalah a fungsi rekursif koneksi dengan lokal negara	Fungsi, pro- pengobatan, kelas, dan komponen adalah penutupan dengan e ffi sien distrib. protokol	Penutupan adalah dasar dari semua paradigma
Deterministik konkurensi (lihat Bagian 6)	(tidak didukung)	Deterministik eksekusi semua objek dalam satu tong (proses)	Aliran data setuju rency dengan e ffi- protokol yang efisien untuk variabel aliran data	Konkurensi adalah sebagai semudah fungsional pemrograman, tidak kondisi balapan
Penyampaian pesan konkurensi (lihat Bagian 4.3)	Kesalahan toleransi dengan isolasi, kesalahan deteksi dengan pesan	Keamanan oleh iso- lation, messages antar objek di tong yang berbeda	Asinkron pesan pro- tocols untuk menyembunyikan latensi	Multi-agen pro- tata bahasa adalah mantan menekan dan mudah untuk memprogram
Shared-state konkurensi (lihat Bagian 4.4)	Database global (Mnesia) simpan negara yang konsisten	(tidak didukung)	Global yang koheren negara protokol; transaksi untuk latensi dan kesalahan toleransi	Negara yang dinamai untuk modularitas

Tabel 1. Struktur berlapis dari bahasa pemrograman definitif

Bagian ini menyajikan struktur dari satu kemungkinan bahasa definitif [52]. Kami mempelajari empat proyek penelitian yang dilakukan untuk memecahkan empat masalah yang sangat berbeda. Solusi yang dicapai oleh keempat proyek merupakan kontribusi yang signifikan untuk bidang masing-masing. Keempat proyek menganggap desain bahasa sebagai faktor kunci untuk mencapai kesuksesan. Yang mengejutkan adalah keempat proyek tersebut akhirnya menggunakan bahasa dengan struktur yang sangat mirip. Tabel 1 menunjukkan properti umum dari bahasa pemrograman yang ditemukan di masing-masing dari empat proyek. Bahasa umum memiliki struktur berlapis dengan empat lapisan: inti fungsional yang ketat, diikuti oleh konkurensi deklaratif, lalu penyampaian pesan asinkron, dan akhirnya status bernama global. Struktur berlapis ini secara alami mendukung empat paradigma. Kami secara singkat merangkum keempat proyek tersebut:

1. *Erlang* Pemrograman sistem tertanam yang sangat tersedia untuk telekomunikasi.
   Proyek ini dilakukan oleh Joe Armstrong dan rekan-rekannya di Ericsson Computer Science Laboratory mulai tahun 1986. Bahasa Erlang dirancang dan implementasi pertama yang efisien dan stabil diselesaikan pada tahun 1991 [5, 6]. Program Erlang terdiri dari proses ringan bernama terisolasi yang saling mengirim pesan. Karena isolasi, program Erlang dapat dijalankan hampir tidak berubah pada sistem terdistribusi dan prosesor multi-core. Sistem Erlang memiliki database yang direplikasi, Mnesia, untuk menjaga keadaan koheren global. Erlang dan platform pemrogramannya, sistem OTP (Open Telecom Platform), berhasil digunakan dalam sistem komersial oleh Ericsson dan perusahaan lain [57, 17].

2. *E* Memprogram sistem terdistribusi yang aman dengan banyak pengguna dan beberapa
   domain keamanan. Proyek ini dilakukan selama bertahun-tahun oleh lembaga yang berbeda. Ini dimulai dengan model kemampuan Dennis dan Van Horn pada tahun 1965 [13] dan model Aktor Carl Hewitt pada tahun 1973 [24] dan itu dipimpin melalui pemrograman logika bersamaan ke bahasa E yang dirancang oleh Doug Barnes, Mark Miller, dan rekan-rekan mereka [32 , 31]. Pendahulu E telah digunakan untuk mengimplementasikan berbagai virtual multiuser

lingkungan. Program E terdiri dari tong (proses) utas tunggal terisolasi yang menampung objek aktif yang saling mengirim pesan. Konkurensi deterministik penting di E karena nondeterminisme dapat mendukung saluran rahasia.

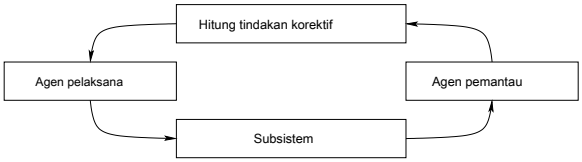
3. *Oz yang didistribusikan* Membuat pemrograman terdistribusi transparan jaringan menjadi praktis.
   Proyek ini dimulai pada tahun 1995 dalam proyek PERDIO di DFKI dengan kesadaran bahwa desain bahasa Oz yang difaktorkan dengan baik, pertama kali dikembangkan oleh Gert Smolka dan murid-muridnya pada tahun 1991 sebagai hasil dari proyek ACCLAIM, adalah awal yang baik. poin untuk membuat jaringan distribusi transparan praktis [45]. Hal ini menghasilkan Sistem Pemrograman Mozart yang mengimplementasikan Oz Terdistribusi dan pertama kali dirilis pada tahun 1999 [22, 34]. Karya terbaru telah menyederhanakan Mozart dan meningkatkan kekuatannya untuk membangun abstraksi toleransi kesalahan [10].

4. *Didactic Oz* Mengajar pemrograman sebagai disiplin terpadu yang mencakup semua yang populer
   paradigma pemrograman. Proyek ini dimulai pada tahun 1999 dengan realisasi oleh penulis dan Seif Haridi bahwa Oz sangat cocok untuk mengajar pemrograman karena memiliki banyak konsep pemrograman dalam desain yang difaktorkan dengan baik, memiliki semantik sederhana, dan memiliki implementasi berkualitas tinggi. . Buku teks [50], yang diterbitkan pada tahun 2004, "merekonstruksi" desain Oz menurut pendekatan berprinsip (lihat Bagian 2.3). Buku ini adalah dasar dari kursus pemrograman yang sekarang diajarkan di beberapa lusin universitas di seluruh dunia. Penulis telah menggunakannya di UCL sejak 2003 untuk kursus pemrograman tahun kedua yang diberikan kepada semua mahasiswa teknik dan kursus pemrograman bersamaan tahun ketiga. Kursus tahun kedua (sejak 2005 disebut FSAB1402) sangat menarik karena mencakup tiga paradigma terpenting, fungsional, berorientasi objek,

Dari struktur umum desain ini, seseorang dapat menyimpulkan beberapa konsekuensi yang masuk akal untuk desain bahasa. Pertama, bahwa pengertian pemrograman deklaratif merupakan inti dari bahasa pemrograman. Ini sudah terkenal; penelitian kami memperkuat kesimpulan ini. Kedua, pemrograman deklaratif itu akan tetap menjadi inti di masa mendatang, karena pemrograman yang terdistribusi, aman, dan toleran terhadap kesalahan adalah topik penting yang memerlukan dukungan dari bahasa pemrograman. Ketiga, konkurensi deterministik adalah bentuk penting dari pemrograman konkuren yang tidak boleh diabaikan. Kami berkomentar bahwa konkurensi deterministik adalah cara terbaik untuk mengeksploitasi paralelisme prosesor multi-inti karena semudah pemrograman fungsional dan tidak dapat memiliki kondisi balapan (lihat juga Bagian 6) [53].

3.3 Arsitektur sistem swasembada

Kami telah menyajikan beberapa kesimpulan awal tentang bahasa definitif; mari kita sekarang menjadi ambisius dan memperluas cakupan kita ke sistem perangkat lunak. Sistem perangkat lunak terakhir adalah sistem yang tidak memerlukan bantuan manusia, yaitu, ia dapat menyediakan setiap modifikasi perangkat lunak yang dibutuhkannya, termasuk pemeliharaan, deteksi dan koreksi kesalahan, dan adaptasi terhadap perubahan kebutuhan. Sistem seperti itu bisa disebut *swasembada [* 44]. Sistem swasembada bisa sangat kuat; misalnya jaringan peer-to-peer dapat mengelola



Gambar 6. Sebuah umpan balik tunggal

diri mereka sendiri untuk bertahan hidup di lingkungan yang sangat tidak bersahabat dengan melakukan transisi fase reversibel [44, 54]. Mari kita kesampingkan dulu kecerdasan buatan yang diperlukan untuk membangun sistem semacam itu, dan menyelidiki mekanisme bahasa yang dibutuhkannya saja. Sistem mungkin meminta bantuan manusia dalam beberapa kasus, tetapi pada prinsipnya sistem harus berisi semua mekanisme yang diperlukan untuk menyelesaikan tugasnya.

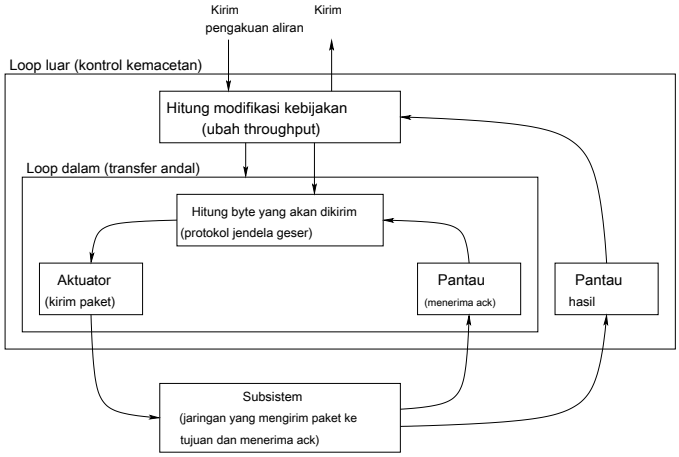
Apa arsitektur yang masuk akal untuk merancang sistem swasembada? Dari kesimpulan pada bagian sebelumnya dan pengalaman kami dalam membangun sistem terdistribusi, kami dapat mengusulkan arsitektur. Dalam hal paradigma pemrograman, yang pertama kita butuhkan adalah komponen sebagai entitas kelas pertama (ditentukan oleh closure) yang dapat dimanipulasi melalui program tingkat tinggi. Di atas level ini, komponen berperilaku sebagai agen serentak terisolasi yang berkomunikasi melalui penyampaian pesan. Terakhir, kita membutuhkan status bernama dan transaksi untuk konfigurasi ulang sistem dan pemeliharaan sistem. Status bernama memungkinkan kita untuk mengelola konten komponen dan mengubah interkoneksinya. Ini memberi kita bahasa yang memiliki struktur berlapis yang mirip dengan bagian sebelumnya.

Dengan bahasa ini kita bisa memprogram sistem kita. Untuk memungkinkan program menyesuaikan diri dengan lingkungannya, kami mengambil inspirasi dari sistem biologis dan mengatur komponennya sebagai putaran umpan balik. Sistem kemudian terdiri dari satu set loop umpan balik yang berinteraksi. Sebuah umpan balik tunggal terdiri dari tiga komponen bersamaan yang berinteraksi dengan subsistem (lihat Gambar 6): agen pemantau, agen koreksi, dan agen penggerak. Sistem realistis terdiri dari banyak putaran umpan balik. Karena setiap subsistem harus senyaman mungkin, harus ada putaran umpan balik di semua tingkatan. Putaran umpan balik ini dapat berinteraksi dalam dua cara mendasar:

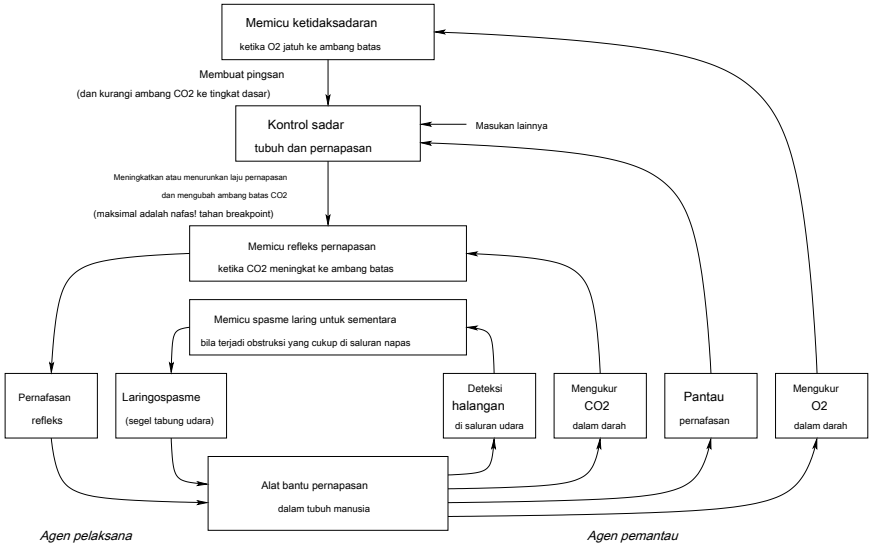
- Stigmergy: Dua loop berbagi satu subsistem.
- Manajemen: Satu loop mengontrol loop lain secara langsung.

Gambar 8 memberikan contoh dunia nyata dari biologi: sistem pernapasan manusia [49]. Sistem ini berisi empat loop. Tiga loop membentuk menara yang dihubungkan oleh manajemen. Putaran keempat berinteraksi dengan yang lain melalui stigmergy.

Gaya desain sistem yang diilustrasikan oleh sistem pernapasan manusia dapat diterapkan pada pemrograman. Sebuah program kemudian terdiri dari serangkaian putaran umpan balik yang berinteraksi melalui stigmergy dan manajemen. Gambar 7 menunjukkan bagian dari Transmission Control Protocol sebagai struktur loop umpan balik [49]. Loop dalam menerapkan transfer aliran byte yang andal menggunakan protokol jendela geser. Loop luar melakukan kontrol kongesti: jika terlalu banyak paket yang hilang, itu mengurangi kecepatan transfer dari loop dalam dengan mengurangi ukuran jendela. Dalam pandangan kami, struktur perangkat lunak berskala besar akan semakin banyak dikerjakan dengan gaya swasembada ini. Jika tidak dilakukan dengan cara ini, perangkat lunak akan menjadi terlalu rapuh dan runtuh dengan kesalahan atau masalah acak.



Gambar 7. TCP sebagai struktur loop umpan balik



Angka 8. Sistem pernapasan manusia sebagai struktur loop umpan balik

4 Konsep pemrograman

Paradigma pemrograman dibangun dari konsep pemrograman. Pada bagian ini kami menyajikan empat konsep pemrograman yang paling penting, yaitu catatan, penutupan yang tercakup secara leksikal, kemerdekaan (konkurensi), dan status bernama. Kami menjelaskan konsep dan mengapa itu penting untuk pemrograman.

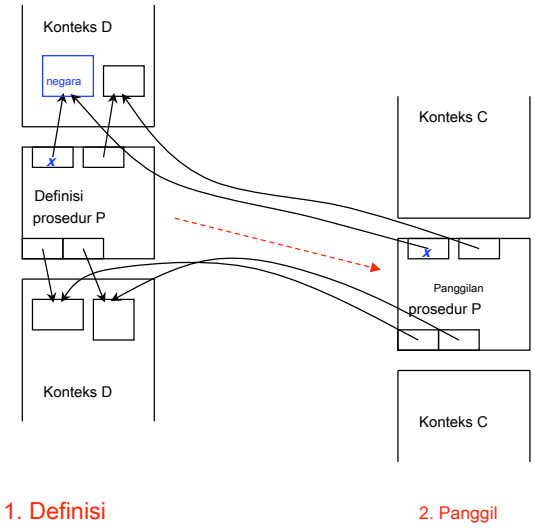
4.1 Rekam

Record adalah struktur data: sekelompok referensi ke item data dengan akses terindeks ke setiap item. Sebagai contoh:

```
R = chanson (nom: "Le Roi des Aulnes"
             artis: "Dietrich Fischer-Dieskau"
             kompositur: "Franz Schubert"
             bahasa: allemand)
```

Catatan tersebut direferensikan oleh pengenal R. Anggota dapat menjadi referensi melalui operasi titik, misalnya, R.nom mengembalikan referensi ke string " Le Roi des Aulnes ". Itu
catatan adalah dasar dari pemrograman simbolik. Bahasa pemrograman simbolik dapat menghitung dengan catatan: membuat catatan baru, menguraikannya, dan memeriksanya. Banyak struktur data penting seperti array, list, string, tree, dan tabel hash dapat diturunkan dari record. Ketika dikombinasikan dengan closure (lihat bagian selanjutnya), record bisa digunakan untuk pemrograman berbasis komponen.

4.2 Penutupan dengan cakupan leksikal



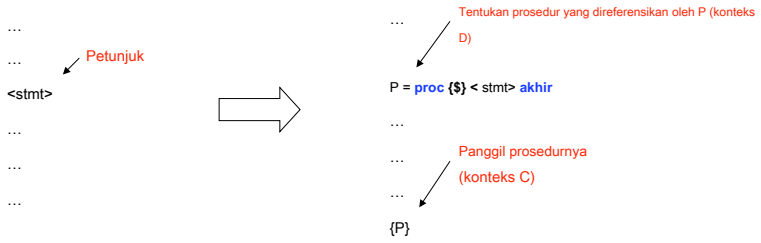
Gambar 9. Definisi dan seruan penutupan

Penutupan yang dilingkupi secara leksikal adalah konsep yang sangat kuat yang merupakan inti dari pemrograman. Pemrograman fungsional, yaitu pemrograman dengan penutupan, adalah a

paradigma sentral (lihat Gambar 2). Dari sudut pandang implementasi, closure menggabungkan prosedur dengan referensi eksternalnya (referensi yang digunakan pada definisinya). Dari sudut pandang programmer, closure adalah “paket kerja”: sebuah program dapat mengubah instruksi apapun menjadi sebuah closure pada satu titik dalam program, meneruskannya ke titik lain, dan memutuskan untuk mengeksekusinya pada titik tersebut. Hasil eksekusinya sama seperti jika instruksi dijalankan pada titik penutupan dibuat.

Gambar 9 menunjukkan secara skematis apa yang terjadi ketika sebuah closure didefinisikan dan kapan ia dipanggil. Prosedur P diimplementasikan dengan penutupan. Pada definisi (konteks D), P menyimpan referensi dari konteks definisi. Misalnya, menyimpan referensi *x*
ke beberapa negara bernama. Kami mengatakan bahwa lingkungan (kumpulan referensi) dari P adalah *Tutup* atas konteks definisi. Pada panggilan (konteks C), P menggunakan referensi dari konteks D.

Gambar 10 menunjukkan satu kemungkinan penggunaan penutupan: membuat struktur kontrol. Di sebelah kiri, kami menjalankan instruksi <stmt>. Di sebelah kanan, alih-alih menjalankan <stmt>, kami menempatkannya di dalam prosedur (penutupan) yang direferensikan oleh P (contoh menggunakan sintaks Oz). Kapan saja nanti dalam program, kita dapat memutuskan untuk memanggil P. Kita telah memisahkan definisi <stmt> dari eksekusinya. Dengan kemampuan ini kita dapat mendefinisikan struktur kendali seperti jika pernyataan atau sementara loop.



Gambar 10. Contoh: memodifikasi program untuk memisahkan pembuatan dan eksekusi

Contoh Gambar 9 dan 10 dapat dengan mudah digeneralisasikan ke prosedur dengan argumen. Lingkungan tertutup ada seperti sebelumnya. Argumen diteruskan selama setiap panggilan. Oleh karena itu closure memiliki dua set referensi: lingkungan tertutup (dari definisi) dan argumen (dari setiap panggilan). Hampir semua bahasa pemrograman (kecuali untuk beberapa leluhur terhormat seperti Pascal dan C) menggunakan closure semacam ini:

- fungsi adalah penutupan;
- prosedur adalah penutupan;
- benda adalah penutup;
- kelas adalah penutupan;
- komponen perangkat lunak adalah penutup.

Banyak kemampuan yang biasanya dikaitkan dengan paradigma spesifik didasarkan pada closure:

- Instansiasi dan genericity, biasanya terkait dengan pemrograman berorientasi objek, dapat dilakukan dengan mudah dengan menulis fungsi yang mengembalikan fungsi lain. Dalam pemrograman berorientasi objek, fungsi pertama disebut "kelas" dan yang kedua disebut "objek".

- Pemisahan masalah, biasanya terkait dengan pemrograman berorientasi aspek, dapat dilakukan dengan mudah dengan menulis fungsi yang menggunakan fungsi lain sebagai argumen. Misalnya, Erlang memiliki fungsi yang mengimplementasikan klien / server yang toleran terhadap kesalahan secara umum. Ini dipanggil dengan argumen fungsi yang mendefinisikan perilaku server. Pemrograman berorientasi aspek dalam bahasa berorientasi objek dijelaskan dalam bab oleh Pierre Cointe [9]. Ini biasanya dilakukan dengan transformasi sintaksis (disebut "tenun") yang menambahkan kode aspek ke sumber aslinya. Bahasa AspectJ adalah contoh yang baik dari pendekatan ini. Weaving sulit digunakan karena rapuh: mudah untuk memasukkan kesalahan dalam program (mengubah kode sumber mengubah semantik program).

- Pemrograman berbasis komponen adalah gaya pemrograman di mana program diatur sebagai komponen, di mana setiap komponen dapat bergantung pada komponen lain. SEBUAH *komponen* adalah blok bangunan yang menentukan bagian dari sebuah program. Sebuah contoh dari sebuah komponen disebut a *modul,* yang merupakan rekor yang berisi penutupan. Modul baru dibuat oleh fungsi yang mengambil modul dependennya sebagai input. Komponen adalah fungsinya.

Bahasa Erlang menerapkan semua kemampuan ini secara langsung dengan closure. Ini praktis dan terukur: produk komersial yang berhasil dengan lebih dari satu juta baris kode Erlang telah dikembangkan (misalnya, saklar ATM AXD-301 [57]). Dalam kebanyakan bahasa lain, penggunaan closure tersembunyi di dalam implementasi bahasa dan tidak tersedia langsung untuk programmer. Jika dilakukan dengan hati-hati, ini bisa menjadi keuntungan, karena penerapannya dapat menjamin bahwa closure digunakan dengan benar.

4.3 Independensi (konkurensi)

Konsep kunci lainnya adalah kemandirian: membangun program sebagai bagian independen. Ini tidak sesederhana kelihatannya. Misalnya, pertimbangkan program yang terdiri dari instruksi yang dijalankan satu demi satu. Instruksi tidak independen karena diperintahkan tepat waktu. Untuk mengimplementasikan kemandirian kita membutuhkan konsep pemrograman baru yang disebut konkurensi. Ketika dua bagian tidak berinteraksi sama sekali, kami mengatakannya *bersamaan.* [3] (Ketika urutan eksekusi dari dua bagian diberikan, kami katakan demikian *sekuensial.)* Bagian yang bersamaan dapat diperpanjang untuk memiliki beberapa interaksi yang terdefinisi dengan baik, yang disebut komunikasi.

Konkurensi tidak boleh disamakan dengan paralelisme. Concurrency adalah konsep bahasa dan paralelisme adalah konsep perangkat keras. Dua bagian menjadi paralel jika dijalankan secara bersamaan pada banyak prosesor. Konkurensi dan paralelisme bersifat ortogonal: dimungkinkan untuk menjalankan program bersamaan pada satu prosesor (menggunakan penjadwalan preemptive dan pembagian waktu) dan menjalankan program sekuensial pada beberapa prosesor (dengan memparalelkan penghitungan). Eksekusi paralel pada prosesor multi-core dijelaskan di halaman 38.

Dunia nyata itu serentak: terdiri dari aktivitas yang berkembang secara mandiri. Dunia komputasi juga serentak. Ini memiliki tiga tingkat konkurensi:

[3] Secara teknis, eksekusi program terdiri dari urutan parsial peristiwa transisi status dan dua peristiwa *bersamaan* jika tidak ada urutan di antara mereka.

- Sistem terdistribusi: sekumpulan komputer yang terhubung melalui jaringan. Aktivitas bersamaan disebut komputer. Ini adalah struktur dasar Internet.
- Sistem operasi: perangkat lunak yang mengelola komputer. Aktivitas bersamaan disebut proses. Proses memiliki ingatan independen. Sistem operasi menangani tugas memetakan eksekusi proses dan memori ke komputer. Misalnya, setiap aplikasi yang berjalan biasanya dijalankan dalam satu proses.
- Aktivitas di dalam satu proses. Aktivitas bersamaan disebut utas. Utas dijalankan secara independen tetapi berbagi ruang memori yang sama. Misalnya, jendela yang berbeda di browser Web biasanya dijalankan dalam utas terpisah.

Perbedaan mendasar antara proses dan utas adalah bagaimana alokasi sumber daya dilakukan. Konkurensi tingkat proses terkadang disebut *konkurensi kompetitif:* setiap proses mencoba memperoleh semua sumber daya sistem untuk dirinya sendiri. Peran utama sistem operasi adalah untuk menengahi permintaan sumber daya yang dilakukan oleh semua proses dan mengalokasikan sumber daya dengan cara yang adil. Konkurensi tingkat utas terkadang disebut *konkurensi kooperatif:*

utas dalam suatu proses berbagi sumber daya dan berkolaborasi untuk mencapai hasil dari proses tersebut. Untaian berjalan dalam aplikasi yang sama dan karenanya dipandu oleh program yang sama.

Ada dua paradigma populer untuk konkurensi. Yang pertama adalah *persetujuan negara-bersama:* thread mengakses item data bersama menggunakan struktur kontrol khusus yang disebut monitor untuk mengelola akses bersamaan. Paradigma ini adalah yang paling populer. Ini digunakan oleh hampir semua bahasa utama, seperti Java dan C #. Cara lain untuk melakukan konkurensi status bersama adalah melalui transaksi: utas memperbarui item data bersama secara atomis. Pendekatan ini digunakan oleh database dan oleh memori transaksional perangkat lunak. Paradigma kedua adalah *konkurensi penyampaian pesan:* agen serentak masing-masing berjalan dalam satu utas yang saling mengirim pesan. Bahasa CSP (Communicating Sequential Processes) [25] dan Erlang [6] menggunakan penyampaian pesan. Proses CSP mengirim pesan sinkron (proses pengiriman menunggu hingga proses penerimaan telah mengambil pesan) dan proses Erlang mengirim pesan asinkron (proses pengiriman tidak menunggu).

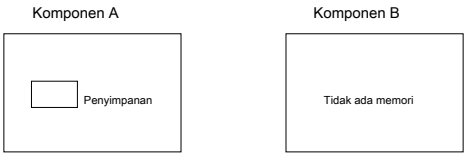
Terlepas dari popularitas mereka, monitor adalah primitif konkurensi yang paling sulit untuk diprogram [29]. Transaksi dan penyampaian pesan lebih mudah, tetapi tetap sulit. Ketiga pendekatan tersebut berasal dari ekspresi mereka: mereka dapat mengekspresikan program nondeterministik (yang pelaksanaannya tidak sepenuhnya ditentukan oleh spesifikasinya), oleh karena itu sulit untuk bernalar tentang kebenarannya. Pemrograman konkuren akan jauh lebih sederhana jika nondeterminisme dikendalikan dengan cara tertentu, sehingga tidak terlihat oleh pemrogram. Bagian 6 dan 7 menyajikan empat paradigma penting yang mengimplementasikan ide ini untuk membuat pemrograman bersamaan jauh lebih sederhana.

4.4 Negara yang diberi nama

Konsep kunci terakhir yang akan kami perkenalkan bernama negara. Negara memperkenalkan gagasan abstrak tentang waktu dalam program. Dalam program fungsional, tidak ada pengertian tentang waktu. Fungsi adalah fungsi matematika: ketika dipanggil dengan argumen yang sama, mereka selalu memberikan hasil yang sama. Fungsi tidak berubah. Di dunia nyata, banyak hal berbeda. Ada beberapa entitas dunia nyata yang memiliki perilaku fungsi yang abadi. Organisme tumbuh dan belajar. Ketika stimulus yang sama diberikan pada suatu organisme pada waktu yang berbeda, reaksinya biasanya akan berbeda. Bagaimana kita bisa memodelkan ini di dalam program? Kita perlu memodelkan entitas dengan identitas unik (namanya) yang perilakunya berubah selama eksekusi

dari program. Untuk melakukan ini, kami menambahkan gagasan abstrak tentang waktu ke program. Waktu abstrak ini hanyalah a *urutan nilai dalam waktu* yang memiliki a *nama tunggal.* Kami menyebut urutan ini negara bernama. Status tanpa nama juga dimungkinkan (monad dan DCG, lihat Bagian 2.1), tetapi tidak memiliki properti modularitas dari status bernama.

Gambar 11 menunjukkan dua komponen, A dan B, di mana komponen A memiliki status internal (memori) dan komponen B tidak. Komponen B selalu memiliki perilaku yang sama: setiap kali dipanggil dengan argumen yang sama, akan memberikan hasil yang sama. Komponen A dapat memiliki perilaku yang berbeda setiap kali dipanggil, jika mengandung nilai yang berbeda dalam status namanya. Memiliki nama negara adalah berkah dan kutukan. Merupakan berkah karena memungkinkan komponen beradaptasi dengan lingkungannya. Itu bisa tumbuh dan belajar. Ini adalah kutukan karena komponen dengan status bernama dapat mengembangkan perilaku yang tidak menentu jika konten dari status bernama tidak diketahui atau salah. Sebuah komponen tanpa status bernama, setelah terbukti benar, selalu tetap benar. Benar tidak begitu mudah untuk dipertahankan untuk sebuah komponen dengan status bernama. Aturan yang baik adalah bahwa negara yang bernama tidak boleh tidak terlihat:



Gambar 11. Komponen dengan status bernama dan komponen tanpa status bernama

Status dan modularitas bernama

Status bernama penting untuk modularitas sistem. Kami mengatakan bahwa sistem (fungsi, prosedur, komponen, dll.) Adalah *modular* jika pembaruan dapat dilakukan ke bagian sistem tanpa mengubah bagian sistem lainnya. Kami memberikan skenario untuk menunjukkan bagaimana kami dapat merancang sistem modular dengan menggunakan status bernama. Tanpa status bernama, ini tidak mungkin.

Asumsikan bahwa kami memiliki tiga pengembang, P, U1, dan U2. P telah mengembangkan modul M yang berisi dua fungsi F dan G. U1 dan U2 adalah pengguna M: program mereka sendiri menggunakan modul M. Berikut adalah satu kemungkinan definisi dari M:

```
kesenangan { ModuleMaker}
    kesenangan { F ...}
        . . .       % Definisi F
    akhir
    kesenangan { G ...}
        . . .       % Definisi G
    akhir
di
    modul (f: F g: G)
akhir
M = {ModuleMaker}% Penciptaan M
```

Fungsinya ModuleMaker adalah komponen perangkat lunak, yaitu mendefinisikan perilaku bagian dari sistem. Kami membuat contoh komponen ini dengan memanggil ModuleMaker. Salah satu contohnya adalah modul M. Perhatikan bahwa antarmuka modul hanyalah sebuah catatan, di mana setiap bidang adalah salah satu operasi modul. Modul M memiliki dua operasi F dan G.

Sekarang asumsikan bahwa pengembang U2 memiliki aplikasi yang menghabiskan banyak waktu kalkulasi. U2 ingin menyelidiki dimana selama ini dihabiskan, sehingga ia bisa menulis ulang lamarannya agar lebih murah. U2 mencurigai hal itu F dipanggil terlalu sering dan dia ingin memverifikasi ini. U2 menginginkan versi baru M yang menghitung berapa kali F disebut. Jadi U2 menghubungi P dan memintanya untuk membuat versi baru M yang melakukan ini, tetapi tanpa mengubah antarmuka (yang mendefinisikan operasi

M dan bagaimana mereka dipanggil) karena jika tidak U2 harus mengubah semua programnya (belum lagi U1!).

Mengherankan! Ini tidak mungkin tanpa status bernama. Jika F tidak memiliki status bernama maka tidak dapat mengubah perilakunya. Secara khusus, itu tidak dapat menyimpan penghitung berapa kali itu dipanggil. Satu-satunya solusi dalam program tanpa status bernama adalah mengubah F antarmuka (argumennya):

```
kesenangan { F ... Fin Fout} Fout =
      Fin + 1
      . . .
akhir
```

Kami menambahkan dua argumen ke F, yaitu Sirip dan Fout. Saat menelepon F, Fin memberikan hitungan berapa kali F dipanggil, dan F menghitung hitungan baru dalam Fout dengan menambahkan satu ke Sirip. Saat menelepon F, kita harus menghubungkan semua argumen baru ini. Misalnya, tiga panggilan berturut-turut ke F akan terlihat seperti ini:

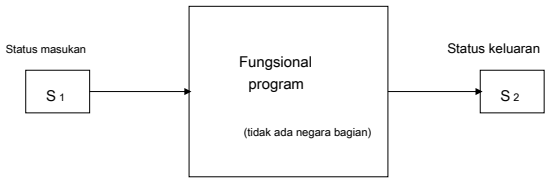
```
A = {F ... F1 F2}
B = {F ... F2 F3}
C = {F ... F3 F4}
```

F1 adalah hitungan awal. Panggilan pertama menghitung F2, yang diteruskan ke panggilan kedua, dan seterusnya. Panggilan terakhir mengembalikan hitungan F4. Kami melihat bahwa ini adalah solusi yang sangat buruk, karena U2 harus mengganti programnya dimanapun F disebut. Lebih parah lagi: U1 juga harus merubah programnya, padahal U1 tidak pernah meminta perubahan apapun. Semua pengguna M, bahkan U1, harus mengubah program mereka, dan mereka sangat tidak senang dengan biaya birokrasi ekstra ini.

Solusi untuk masalah ini adalah dengan menggunakan status bernama. Kami memberikan memori internal ke modul M. Dalam Oz, memori internal ini disebut a *sel* atau a *sel variabel*. Ini sesuai dengan apa yang oleh banyak bahasa disebut variabel. Inilah solusinya:

```
kesenangan { ModuleMaker}
    X = {NewCell 0}% Buat sel yang direferensikan oleh X
    kesenangan { F ...}
        X: = @ X +1          % Konten baru X sudah lama ditambah 1% Definisi asli F
        . . .
    akhir
    kesenangan { F ...}
        . . .                % Definisi asli dari G
    akhir
    kesenangan { Hitung} @X akhir% Kembalikan konten X
di
    modul (f: F g: G c: Hitung)
akhir
M = {ModuleMaker}
```

Modul baru M berisi sel di dalamnya. Kapanpun F disebut, sel bertambah. Operasi tambahan Hitung (diakses oleh Mc) mengembalikan hitungan sel saat ini. Antarmuka F dan G tidak berubah. Sekarang semua orang senang: U2 memiliki modul barunya dan tidak ada yang perlu mengubah program mereka sama sekali F dan G dipanggil dengan cara yang sama. Ini menunjukkan bagaimana status bernama memecahkan masalah modularitas.



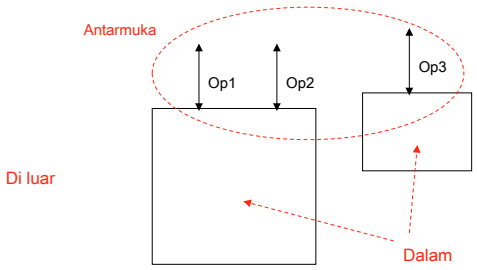
Gambar 12. Sebuah program sebagai transformator negara

Keuntungan utama dari status bernama adalah program menjadi modular. Kerugian utamanya adalah sebuah program bisa menjadi tidak benar. Tampaknya kita perlu memiliki dan tidak menamai negara pada saat yang bersamaan. Bagaimana kita mengatasi dilema ini? [4] Salah satu solusinya adalah dengan memusatkan penggunaan status bernama di satu bagian program dan menghindari status bernama di bagian lain. Gambar 12 menunjukkan bagaimana desain ini bekerja. Sebagian besar program adalah fungsi murni tanpa status bernama. Sisa dari program ini adalah transformator keadaan: ia memanggil fungsi murni untuk melakukan pekerjaan yang sebenarnya. Ini memusatkan status bernama di sebagian kecil program.

5 Abstraksi data

Abstraksi data adalah cara untuk mengatur penggunaan struktur data sesuai dengan aturan yang tepat yang menjamin bahwa struktur data digunakan dengan benar. Abstraksi data memiliki bagian dalam, bagian luar, dan antarmuka di antara keduanya. Semua struktur data disimpan di dalam. Bagian dalam tersembunyi dari luar. Semua operasi pada data harus melewati antarmuka. Gambar 13 menunjukkan ini secara grafis. Ada tiga keuntungan bagi organisasi ini:

[4] Dilema semacam ini ada di jantung penemuan. Ini disebut a *kontradiksi teknis* dalam Theory of Inventive Problem Solving (TRIZ) Altshuller, yang menyediakan teknik untuk solusinya [2].



Gambar 13. Abstraksi data

1. Pertama, ada jaminan bahwa abstraksi data akan selalu berfungsi dengan baik. Antarmuka mendefinisikan operasi resmi pada struktur data dan tidak ada operasi lain yang mungkin dilakukan.

2. Kedua, program lebih mudah dipahami. Pengguna abstraksi data tidak perlu memahami bagaimana abstraksi diimplementasikan. Program dapat dipartisi menjadi banyak abstraksi, diimplementasikan secara independen, yang sangat mengurangi kompleksitas program. Ini dapat lebih ditingkatkan dengan menambahkan properti komposisionalitas: memungkinkan abstraksi data untuk didefinisikan di dalam abstraksi data lainnya.

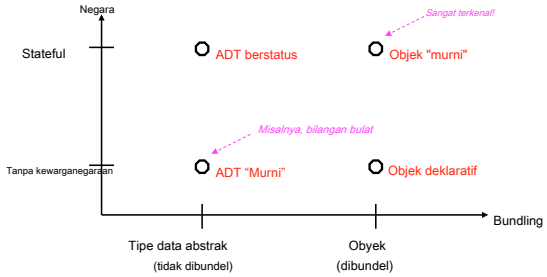
3. Ketiga, menjadi mungkin untuk mengembangkan program yang sangat besar. Kami dapat membagi implementasi di antara tim orang. Setiap abstraksi memiliki satu orang yang bertanggung jawab untuk itu: dia mengimplementasikan dan memeliharanya. Orang itu hanya harus mengetahui antarmuka yang digunakan oleh abstraksinya.

Di sisa bagian ini, kami pertama kali menjelaskan empat cara berbeda untuk mengatur abstraksi data. Kami kemudian memperkenalkan dua prinsip, polimorfisme dan pewarisan, yang sangat meningkatkan kekuatan abstraksi data untuk mengatur program. Pemrograman berorientasi objek, seperti biasanya dipahami, didasarkan pada abstraksi data dengan polimorfisme dan pewarisan.

5.1 Objek dan tipe data abstrak

Ada empat cara utama untuk mengatur abstraksi data, yang diatur dalam dua sumbu. Sumbu pertama adalah *negara:* apakah abstraksi menggunakan status bernama atau tidak. Sumbu kedua adalah *bundling:* apakah abstraksi menggabungkan data dan operasi menjadi satu kesatuan (ini disebut file *obyek* atau a *abstraksi data prosedural (PDA)),* atau apakah abstraksi memisahkan mereka (ini disebut file *tipe data abstrak (ADT)).* Mengalikan dua sumbu menghasilkan empat kemungkinan, yang ditunjukkan pada Gambar 14.

Dua dari empat kemungkinan ini sangat populer dalam bahasa pemrograman modern. Kami memberikan contoh keduanya dalam bahasa Jawa. Integer di Java direpresentasikan sebagai nilai (1, 2, 3, dll.) Dan operasi (+, -, *, dll.). Nilai diteruskan sebagai argumen ke operasi, yang mengembalikan nilai baru. Ini adalah contoh tipe data abstrak tanpa status bernama. Objek di Java menggabungkan data (atributnya) dan operasinya (metodenya) ke dalam satu entitas. Ini adalah contoh objek dengan status bernama.



Gambar 14. Empat cara untuk mengatur abstraksi data

Dua kemungkinan lain, tipe data abstrak dengan status bernama dan objek deklarasi, juga bisa berguna. Tapi mereka jarang digunakan dalam bahasa saat ini.

5.2 Polimorfisme dan prinsip tanggung jawab

Prinsip terpenting dari pemrograman berorientasi objek, setelah abstraksi data itu sendiri, adalah polimorfisme. Dalam bahasa sehari-hari, kita katakan suatu entitas adalah polimorfik jika dapat mengambil bentuk yang berbeda. Dalam pemrograman komputer, kita katakan suatu entitas adalah polimorfik jika dapat mengambil argumen dari tipe yang berbeda. Kemampuan ini sangat penting untuk mengorganisir program-program besar sehingga tanggung jawab rancangan program terkonsentrasi di tempat-tempat yang sudah didefinisikan dengan baik dan tidak tersebar di seluruh program. Untuk menjelaskan ini, kami menggunakan contoh dunia nyata. Seorang pasien yang sakit pergi ke dokter. Pasien tidak perlu menjadi seorang dokter, tetapi hanya memberitahu dokter satu pesan: "Sembuhkan aku!". Dokter memahami pesan ini dan melakukan hal yang benar tergantung pada spesialisasinya. Program "GetCured" yang dijalankan oleh pasien bersifat polimorfik: dibutuhkan seorang dokter sebagai argumen dan bekerja dengan semua jenis dokter yang berbeda. Ini karena semua dokter memahami pesan "Sembuhkan saya!".

Untuk pemrograman, gagasan polimorfisme serupa: jika sebuah program bekerja dengan satu abstraksi data sebagai argumen, ia dapat bekerja dengan yang lain, *jika* yang lainnya memiliki antarmuka yang sama. Keempat jenis abstraksi data yang kita lihat sebelumnya mendukung polimorfisme. Tetapi ini sangat sederhana untuk objek, yang merupakan salah satu alasan keberhasilan pemrograman berorientasi objek.

Gambar 15 memberikan contoh. Pertimbangkan paket grafik yang menyertakan rutinitas untuk menggambar berbagai jenis gambar. Kami mendefinisikan ini menggunakan deklarasi kelas. Lihat definisi dari CompoundFigure. Angka-angka ini mendefinisikan yang terdiri dari daftar angka-angka lain (bahkan angka majemuk lainnya!). Metode seri di CompoundFigure tidak tahu cara menggambar salah satu dari angka-angka ini. Tetapi karena bersifat polimorfik, ia dapat memanggil seri di gambar lainnya. Setiap sosok tahu cara menggambar dirinya sendiri. Ini adalah pembagian tanggung jawab yang benar.

```
kelas Angka
    . . .
akhir

kelas Lingkaran
    attr xyr
    sabu seri ... akhir
    . . .
akhir

kelas Garis
    attr x1 y1 x2 y2
    sabu seri ... akhir
    . . .
akhir

kelas CompoundFigure
    attr figlist
    sabu seri
        untuk F di @ figlist lakukan { F draw} akhir
    akhir
    . . .
akhir
```

Gambar 15. Contoh polimorfisme dalam paket grafik

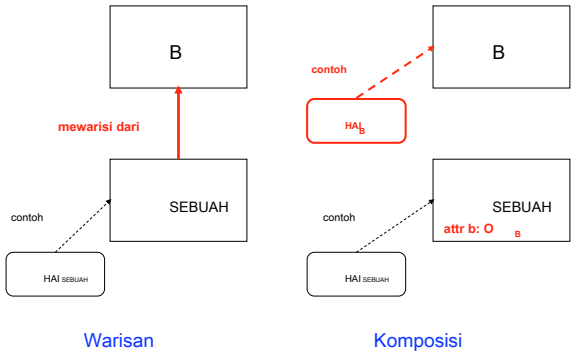
5.3 Prinsip pewarisan dan substitusi

Prinsip penting kedua dari pemrograman berorientasi objek adalah pewarisan. Banyak abstraksi memiliki banyak kesamaan, dalam apa yang mereka lakukan tetapi juga dalam implementasinya. Ide bagus untuk mendefinisikan abstraksi untuk menekankan hubungan bersama mereka dan tanpa mengulangi kode yang mereka bagikan. Kode yang berulang merupakan sumber kesalahan: jika satu salinan diperbaiki, semua salinan harus diperbaiki. Terlalu mudah untuk melupakan beberapa salinan atau memperbaikinya dengan cara yang salah.

Pewarisan memungkinkan untuk mendefinisikan abstraksi secara bertahap. Definisi A dapat mewarisi dari definisi B lain: definisi A menggunakan definisi B sebagai dasarnya dan menunjukkan bagaimana definisi tersebut dimodifikasi atau diperluas. Definisi tambahan A disebut a *kelas*. Namun,
abstraksi yang menghasilkan definisi penuh, bukan yang parsial.

Pewarisan bisa menjadi alat yang berguna, tapi harus digunakan dengan hati-hati. Kemungkinan untuk memperluas definisi B dengan pewarisan dapat dilihat sebagai antarmuka lain ke B. Antarmuka ini perlu dipertahankan selama masa B. Ini adalah sumber bug tambahan. Rekomendasi kami adalah menggunakan warisan sesedikit mungkin. Saat mendefinisikan kelas, kami merekomendasikan untuk mendefinisikannya sebagai tidak dapat diperluas jika memungkinkan. Di Jawa ini disebut a terakhir kelas.

Alih-alih warisan, kami merekomendasikan untuk menggunakan komposisi sebagai gantinya. Komposisi adalah teknik alami: ini berarti bahwa atribut suatu objek merujuk ke objek lain. Benda-benda itu disusun bersama. Dengan cara ini, tidak perlu memperluas kelas dengan warisan. Kami menggunakan objek seperti yang didefinisikan untuk digunakan. Gambar 16 mengilustrasikan pewarisan dan komposisi berdampingan.



Gambar 16. Warisan versus komposisi

Jika harus menggunakan warisan, maka cara yang benar untuk menggunakannya adalah dengan mengikuti substitusi prinsip. Misalkan kelas A mewarisi dari kelas B dan kita memiliki dua objek, *HAI SEBUAH* dan *HAI B*. Prinsip substitusi menyatakan bahwa prosedur apa pun yang bekerja dengan objek *HAI B* dari kelas B juga harus bekerja dengan objek *HAI SEBUAH* kelas A. Dengan kata lain, warisan tidak boleh merusak apa pun. Kelas A harus merupakan perpanjangan konservatif dari kelas B.

Kami mengakhiri diskusi kami tentang warisan dengan kisah peringatan. Pada 1980-an, sebuah perusahaan multinasional yang sangat besar [5] memulai proyek ambisius berdasarkan pemrograman berorientasi objek. Meski memiliki anggaran beberapa miliar dolar, proyek tersebut gagal total. Salah satu alasan utama kegagalan ini adalah penggunaan warisan yang salah. Dua kesalahan utama dilakukan:

- Melanggar prinsip substitusi. Prosedur yang bekerja dengan objek kelas tidak lagi bekerja dengan objek subkelas. Akibatnya, banyak prosedur yang hampir identik perlu ditulis.

- Menggunakan subclass untuk menutupi bug. Alih-alih mengoreksi bug, subclass dibuat untuk menutupi bug, yaitu untuk menguji dan menangani kasus-kasus di mana bug terjadi. Akibatnya, hierarki kelas menjadi sangat dalam, rumit, lambat, dan penuh dengan bug.

6 Pemrograman konkuren deterministik

Salah satu masalah utama dari pemrograman konkuren adalah nondeterminisme. Eksekusi suatu program bersifat nondeterministik jika pada titik tertentu selama eksekusi ada pilihan apa yang harus dilakukan selanjutnya. Nondeterminisme muncul secara alami ketika ada konkurensi: karena dua aktivitas bersamaan adalah independen, spesifikasi program tidak dapat mengatakan mana yang dijalankan pertama kali. Jika ada beberapa utas yang siap dijalankan, maka di setiap status eksekusi, sistem harus memilih utas mana yang akan dijalankan berikutnya. Pilihan ini dapat dilakukan dengan berbagai cara; biasanya ada bagian dari sistem yang disebut *penjadwal* yang membuat pilihan.

Nondeterminisme sangat sulit ditangani jika dapat diamati oleh pengguna program. Nondeterminisme yang dapat diamati kadang-kadang disebut a *kondisi balapan*. Misalnya, jika masing-masing dari dua utas menetapkan sel variabel ke nilai yang berbeda, maka masing-masing dari dua nilai dapat diamati:

[5] Yang akan tetap anonim.

menyatakan C = {NewCell 0}
benang C: = 1 **akhir**
benang C: = 2 **akhir**

Sel variabel C dapat berisi nilai 1 atau 2 setelah kedua utas dijalankan. Ini adalah kasus sederhana dari kondisi balapan. Banyak kasus rumit yang mungkin terjadi, ketika dua utas berbagi beberapa sel variabel dan melakukan lebih banyak penghitungan dengannya. Debugging dan penalaran tentang program dengan kondisi balapan sangat sulit.

6.1 Menghindari nondeterminisme dalam bahasa yang berbarengan

Cara termudah untuk menghilangkan kondisi ras adalah merancang bahasa yang tidak memiliki nondeterminisme. Tapi ini akan membuang bayi dengan air mandi karena konkurensi secara alami menyiratkan nondeterminisme. Bagaimana kita bisa menghindari efek buruk non-determinisme dan masih memiliki konkurensi? [6] Kita dapat menyelesaikan masalah ini dengan membuat perbedaan yang jelas antara nondeterminisme *dalam* sistem, yang tidak dapat dihindari, dan

tampak nondeterminisme, yang mungkin bisa dihindari. Kami menyelesaikan masalah dalam dua langkah:

- Pertama, kami membatasi nondeterminisme yang dapat diamati pada bagian-bagian program yang benar-benar membutuhkannya. Bagian lain seharusnya tidak memiliki nondeterminisme yang dapat diamati.
- Kedua, kami mendefinisikan bahasa sehingga memungkinkan untuk menulis program bersamaan tanpa nondeterminisme yang dapat diamati.

Paradigma serentak	Balapan bisa jadi?	Masukan bisa nondeterm.?	Contoh bahasa
Deklaratif konkurensi	Tidak	Tidak	Oz [34], Alice [38]
Paksaan pemrograman	Tidak	Tidak	Gecode [43], Numerika [48] FrTime
Reaktif fungsional pemrograman	Tidak	Iya	[12], Yampa [27]
Sinkronisasi terpisah pemrograman	Tidak	Iya	Esterel [7], Kilau [21], Sinyal [26] Erlang [6], E
Penyampaian pesan konkurensi	Iya	Iya	[32]

Meja 2. Empat paradigma konkuren deterministik dan satu yang tidak

Apakah mungkin memiliki bahasa bersamaan tanpa nondeterminisme yang dapat diamati? Pemeriksaan superfisial dari bahasa pemrograman populer mungkin membuat orang mengatakan tidak: Java dan C # menggunakan konkurensi status bersama dan Erlang menggunakan konkurensi penyampaian pesan, yang semuanya memiliki nondeterminisme yang dapat diamati. Untungnya, kesan superfisial ini sepenuhnya salah. Setidaknya ada empat paradigma pemrograman berguna yang bersamaan tetapi tidak memiliki nondeterminisme yang dapat diamati (tidak ada kondisi balapan). Tabel 2 mencantumkan keempat ini bersama dengan konkurensi penyampaian pesan. Mari kami jelaskan lebih detail.

[6] Ini adalah contoh lain dari kontradiksi teknis. Lihat catatan kaki di halaman 29.

Konkurensi deklaratif (disebut juga aliran data monotonik) Dalam paradigma ini, input terministik diterima dan digunakan untuk menghitung output deterministik. Paradigma ini hidup seutuhnya dalam dunia deterministik. Jika ada beberapa aliran input, mereka harus deterministik, yaitu, program harus tahu persis elemen input apa yang akan dibaca untuk menghitung setiap output (misalnya, mungkin ada konvensi bahwa tepat satu elemen dibaca dari setiap aliran input) . Dua bahasa yang menerapkan paradigma ini adalah Oz [50, 34] dan Alice [38]. Paradigma ini bisa dijadikan malas tanpa kehilangan sifat baiknya. Paradigma dan ekstensi malasnya dijelaskan lebih rinci di Bagian 6.2. Pemrograman batasan terkait dengan konkurensi deklaratif dan dijelaskan di Bagian 7.

Ada juga a *nonmonotonik* paradigma aliran data, di mana perubahan pada input apa pun segera disebarkan melalui program. Perubahan tersebut dapat dikonseptualisasikan sebagai *token aliran data* bepergian melalui program. Paradigma ini dapat menerima masukan nondeterministik, tetapi memiliki kelemahan yaitu terkadang menambahkan nondeterminisme sendiri yang tidak ada dalam masukan (disebut "kesalahan" di bawah). Itulah sebabnya kami tidak membahas paradigma ini lebih lanjut dalam bab ini. Pemrograman reaktif fungsional mirip dengan aliran data nonmonotonik tetapi tanpa gangguan.

Pemrograman reaktif fungsional (disebut juga pemrograman sinkron berkelanjutan) Dalam paradigma ini, program berfungsi tetapi argumen fungsi dapat diubah dan perubahan disebarkan ke keluaran. Paradigma ini dapat menerima masukan nondeterministik dan tidak menambahkan nondeterminismenya sendiri. Secara semantik, argumen adalah fungsi kontinu dari variabel terurut total (yang dapat sesuai dengan besaran yang berguna seperti *waktu* atau *ukuran).* Penerapan biasanya menghitung ulang nilai hanya jika nilai tersebut berubah dan dibutuhkan. Diskritisasi diperkenalkan hanya jika hasil dihitung [16]. Ini berarti bahwa penskalaan sewenang-wenang dimungkinkan tanpa kehilangan akurasi karena perkiraan. Jika perubahan disebarkan dengan benar, maka program fungsional tidak menambahkan nondeterminisme apapun. Misalnya, fungsi sederhana

ekspresi x + (x * y) dengan x = 3 dan y = 4 memberi 15. Jika x diubah menjadi 5, kemudian hasil ekspresi berubah dari 15 untuk 25. Menerapkan ini secara naif dengan aliran bersamaan menghubungkan agen waktu ke agen plus tidak benar. Implementasi ini dapat memberikan a *kesalahan,* misalnya jika nilai baru x mencapai penjumlahan sebelum hasil perkalian baru. Ini memberikan hasil sementara dari 17, mana yang salah. Glitch adalah sumber nondeterminisme yang harus dihindari penerapannya, misalnya dengan praproses waktu kompilasi (melakukan operasi topologis) atau batasan penjadwalan thread. Beberapa bahasa yang menerapkan paradigma ini adalah Yampa (tertanam di Haskell) [27] dan FrTime (tertanam dalam Skema) [12].

Pemrograman sinkronis terpisah Dalam paradigma ini, program menunggu kejadian masukan, melakukan kalkulasi internal, dan mengeluarkan kejadian keluaran. Ini disebut a *sistem reaktif.* Sistem reaktif harus deterministik: urutan input yang sama menghasilkan urutan output yang sama. Seperti pemrograman reaktif fungsional, paradigma ini dapat menerima masukan nondeterministik dan tidak menambahkan nondeterminismenya sendiri. Perbedaan utamanya adalah bahwa waktu bersifat diskrit, bukan kontinu: waktu bergerak maju secara bertahap dari satu peristiwa masukan ke peristiwa berikutnya. Peristiwa keluaran dipancarkan pada waktu logis yang sama

instants sebagai peristiwa masukan. [7] Semua perhitungan yang dilakukan untuk menentukan peristiwa keluaran berikutnya dianggap sebagai bagian dari waktu instan yang sama. Inilah yang sebenarnya terjadi dalam logika digital clock: rangkaian kombinasional bersifat "seketika" (terjadi dalam satu siklus) dan rangkaian sekuensial "memerlukan waktu": menggunakan memori clock (terjadi selama beberapa siklus). Sinyal jam adalah urutan peristiwa input. Menggunakan waktu diskrit sangat menyederhanakan pemrograman untuk sistem reaktif. Misalnya, ini berarti bahwa subprogram dapat disusun dengan mudah: peristiwa keluaran dari satu subkomponen secara instan tersedia sebagai peristiwa masukan di subkomponen lain. Beberapa bahasa yang menerapkan paradigma ini adalah Esterel [7], Luster [21], dan Signal [26]. Esterel adalah bahasa penting, Lustre adalah bahasa aliran data fungsional, dan Signal adalah bahasa aliran data relasional. Dimungkinkan untuk menggabungkan paradigma kendala sinkron dan konkuren diskrit untuk mendapatkan keuntungan dari keduanya. Ini memberikan model Timed CC, yang dijelaskan dalam bab oleh Carlos Olarte *dkk [*35].

Ketiga paradigma memiliki aplikasi praktis yang penting dan telah diwujudkan dengan bahasa yang memiliki implementasi yang baik. Ini di luar cakupan bab ini untuk membahas secara rinci ketiga paradigma. Karena kesederhanaan dan pentingnya, kami hanya memberikan gagasan dasar dari paradigma pertama, konkurensi deklaratif, di Bagian 6.2.

Konkurensi deterministik dan musik komputer

Konkurensi deterministik ada di mana-mana dalam musik komputer. Kami memberikan empat contoh:

- Studio komposisi musik OpenMusic menyediakan seperangkat alat grafis untuk komposer [1]. Ia memiliki bahasa visual interaktif untuk mendefinisikan grafik aliran data komponen musik (disebut "patch"). Ini memiliki semantik yang mirip dengan pemrograman sinkronis diskrit. Perbedaan utama adalah pemicuan eksplisit. Pemicuan eksplisit digunakan untuk antarmuka antara komposer manusia dan sistem. Evaluasi grafik dipicu oleh komposer yang secara eksplisit meminta nilai suatu komponen. Hal ini menyebabkan rangkaian penghitungan yang digerakkan oleh permintaan (malas): komponen meminta evaluasi komponen yang bergantung padanya, dan seterusnya secara transitif hingga mencapai komponen yang tidak memiliki dependensi. Komponen memiliki memori: hasil evaluasi disimpan di dalam komponen.

- Pengikut skor Antescofo memungkinkan perangkat lunak musik mengikuti musisi manusia saat mereka memainkan sebuah karya [11]. Ini menerjemahkan waktu jam (detik) menjadi waktu manusia (tempo, yaitu, ketukan per menit). Antescofo diperluas dengan bahasa yang memungkinkan komposer menganotasi skor dengan instruksi kontrol. Bahasa ini memiliki semantik yang mirip dengan pemrograman sinkronis diskrit, di mana peristiwa masukan adalah catatan dan peristiwa keluaran adalah petunjuk pembuatnya. Perbedaan utamanya adalah Antescofo memiliki osilator tempo yang dapat menjadwalkan kejadian pada not pecahan. Ini menambah aspek berkelanjutan pada eksekusi Antescofo.

[7] Secara teknis, program dalam bahasa sinkron seperti Esterel mendefinisikan deterministik *Mesin mealy,* yang merupakan otomat keadaan terbatas di mana setiap transisi keadaan diberi label dengan masukan dan keluaran.

- Alat pengolah sinyal Faust disajikan dalam bab oleh Yann Orlarey *dkk*
  menyediakan bahasa aliran data visual untuk mendefinisikan plug-in atau aplikasi pemrosesan sinyal audio [37]. Bahasa aliran data memiliki semantik sinkron diskrit dengan rasa fungsional, mirip dengan Lustre [36]. Faust dioptimalkan untuk kinerja tinggi: ia mendukung frekuensi clock tinggi dan kompilasi yang efisien ke C ++.

- Max / MSP pemrograman dan lingkungan eksekusi menyediakan seperangkat alat grafis untuk kinerja musik menggunakan bahasa visual interaktif untuk mendefinisikan grafik aliran data [39]. Max / MSP memiliki tiga bagian yang berbeda: bahasa aliran data Max, yang menyediakan aliran kontrol keseluruhan, perpustakaan pemrosesan sinyal digital MSP, yang menghasilkan audio, dan perpustakaan Jitter untuk pemrosesan video dan 3D. Grafik aliran data agak mirip dengan OpenMusic, meskipun semantiknya cukup berbeda. Bahasa Max dijalankan secara real-time dan dengan gaya yang lebih awal mulai dari metronom atau generator lainnya. Bahasa ini dirancang untuk membuatnya mudah untuk menulis program aliran data fungsional (menghormati persamaan fungsional setiap saat, mirip dengan pemrograman reaktif fungsional), meskipun implementasinya tidak memaksakan ini. [8] Bahasa ini memiliki semantik berurutan, karena paling banyak satu pesan dapat melintasi grafik aliran data setiap saat. Urutannya tidak langsung terlihat oleh pengguna, tetapi penting untuk eksekusi deterministik.

Sungguh luar biasa bahwa contoh-contoh ini ada pada tiga tingkat abstraksi yang berbeda: komposisi musik (OpenMusic), pertunjukan musik (skala waktu manusia, Max / MSP dan Antescofo), dan kinerja musik (skala waktu pemrosesan sinyal, Max / MSP dan Faust).).

OpenMusic, Max / MSP, dan Antescofo memberikan konfirmasi menggoda dari Tabel 1. OpenMusic memiliki bahasa yang matang yang diatur sebagai tiga lapisan: fungsional, konkurensi deterministik, dan status bersama. Max / MSP memiliki inti sekuensial dengan lapisan konkuren deterministik di atas. Bahasa Antescofo masih dirancang: sejauh ini, Antescofo hanya memiliki lapisan konkuren deterministik, tetapi lapisan lain telah direncanakan. Bahasa Hermes / dl, dijelaskan dalam bab oleh Alexandre François, juga membedakan antara lapisan deterministik dan stateful [19].

6.2 Konkurensi deklaratif

Kami menjelaskan secara singkat bagaimana melakukan konkurensi deklaratif, yang merupakan bentuk konkurensi deterministik pertama dan paling sederhana (lihat bab 4 dari [50] untuk informasi lebih lanjut). Konkurensi deklaratif memiliki keunggulan utama dari pemrograman fungsional, yaitu pengaruh, dalam model konkuren. Artinya semua urutan evaluasi memberikan hasil yang sama, atau dengan kata lain tidak ada ketentuan balapan. Ia menambahkan dua konsep ke paradigma fungsional: variabel benang dan aliran data. Sebuah utas mendefinisikan urutan instruksi, dijalankan secara independen dari utas lain. Utas memiliki satu operasi:

- {NewThread P}: buat utas baru yang menjalankan prosedur 0-argumen P.

Variabel aliran data adalah variabel tugas tunggal yang digunakan untuk sinkronisasi. Variabel aliran data memiliki tiga operasi primitif:

[8] Kesalahan aneh terkadang muncul pada waktu yang tidak tepat selama eksekusi program Max / MSP. Beberapa di antaranya mungkin karena kesalahan program yang mengakibatkan perilaku non-deterministik sementara, yaitu gangguan. Kesalahan seperti itu dapat dihindari dengan mengubah desain bahasa atau sistem run-time-nya, dengan cara yang mirip dengan pemrograman sinkron.

- X = {NewVar}: membuat variabel aliran data baru yang direferensikan oleh X.
- {Bind XV}: mengikat X untuk V, dimana V. adalah nilai atau variabel aliran data lainnya.
- {Tunggu X}: utas saat ini menunggu sampai X terikat pada sebuah nilai.

Dengan menggunakan operasi primitif ini, kami memperluas semua operasi bahasa untuk menunggu hingga argumennya tersedia dan mengikat hasilnya. Misalnya, kami mendefinisikan operasi Menambahkan dalam hal variabel aliran data dan operasi penjumlahan primitif PrimAdd:

```
proc { Tambahkan XYZ}
    {Tunggu X} {Tunggu Y}
    lokal R di { PrimAdd XYR} {Ikat ZR} akhir
akhir
```

Panggilan Z = {Tambahkan 2 3} penyebab Z untuk terikat 5 (output fungsi adalah argumen ketiga prosedur). Kami melakukan hal yang sama untuk semua operasi termasuk kondisional (**jika)** pernyataan (yang menunggu sampai kondisi terikat) dan pemanggilan prosedur (yang menunggu sampai variabel prosedur terikat). Hasilnya adalah bahasa aliran data deklaratif.

Konkurensi deklaratif malas

Kita bisa menambahkan eksekusi malas ke konkurensi deklaratif dan tetap mempertahankan properti baik dari pengaruh dan determinisme. Dalam eksekusi malas, konsumen hasil yang memutuskan apakah akan melakukan kalkulasi atau tidak, bukan produsen hasil. Dalam satu lingkaran, kondisi terminasi ada di konsumen, bukan produsen. Produser bahkan dapat diprogram sebagai loop tak terbatas. Eksekusi malas melakukan perhitungan paling sedikit yang diperlukan untuk mendapatkan hasil. Kita membuat konkurensi deklaratif menjadi malas dengan menambahkan satu konsep, sinkronisasi sesuai kebutuhan, yang diimplementasikan oleh satu operasi:

- {WaitNeeded X}: utas saat ini menunggu hingga utas melakukan { Tunggu X}.

Paradigma ini menambahkan evaluasi malas dan konkurensi ke pemrograman fungsional dan masih bersifat deklaratif. Ini adalah paradigma deklaratif paling umum berdasarkan pemrograman fungsional yang diketahui sejauh ini. [9] Dengan Diperlukan kita bisa mendefinisikan versi malas dari Menambahkan:

```
proc { LazyAdd XYZ}
    utas { WaitNeeded Z} {Tambahkan XYZ} akhir
akhir
```

Ini praktis jika utasnya efisien, seperti di Mozart [34]. Panggilan Z = {LazyAdd 2 3} menunda penambahan sampai nilai Z dibutuhkan. Kami mengatakan bahwa itu menciptakan file *penundaan malas.* Jika utas lain dijalankan Z2 = {Tambahkan Z 4}, maka suspensi akan dijalankan, mengikat Z untuk 5. Jika utas lainnya dijalankan Z2 = {LazyAdd Z 4} sebagai gantinya, dua suspensi malas dibuat. Jika kebutuhan utas ketiga Z2, lalu keduanya akan dieksekusi.

Konkurensi deklaratif dan prosesor multi-core

Dengan munculnya prosesor multi-core, pemrograman paralel akhirnya mencapai arus utama. Prosesor multi-inti menggabungkan dua atau lebih elemen pemrosesan (disebut inti) dalam satu paket, pada satu cetakan atau beberapa cetakan. Inti berbagi interkoneksi

[9] Pemrograman batasan lebih umum tetapi didasarkan pada pemrograman relasional.

ke seluruh sistem dan sering berbagi memori cache on-chip. Karena kerapatan transistor terus meningkat menurut Hukum Moore (berlipat ganda kira-kira setiap dua tahun, yang diperkirakan akan berlanjut setidaknya hingga 2020) [33], jumlah inti juga akan meningkat. Untuk menggunakan semua kekuatan pemrosesan ini, kita perlu menulis program paralel.

Penelitian selama puluhan tahun menunjukkan bahwa pemrograman paralel tidak dapat sepenuhnya disembunyikan dari pemrogram: tidak mungkin secara umum mengubah program sewenang-wenang secara otomatis menjadi program paralel. Tidak ada peluru ajaib. Hal terbaik yang bisa kita lakukan adalah membuat pemrograman paralel semudah mungkin. Bahasa pemrograman dan perpustakaannya harus membantu dan tidak menghalangi programmer. Bahasa tradisional seperti Java atau C ++ tidak dilengkapi dengan baik untuk ini karena konkurensi keadaan bersama sulit.

Konkurensi deklaratif adalah paradigma yang baik untuk pemrograman paralel [53]. Ini karena ia menggabungkan konkurensi dengan properti pemrograman fungsional yang baik. Program adalah fungsi matematika: fungsi yang benar tetap benar tidak peduli bagaimana dipanggil (yang tidak benar untuk objek). Program tidak memiliki ketentuan balapan: bagian mana pun dari program yang benar dapat dijalankan secara bersamaan tanpa mengubah hasil. Setiap program yang benar dapat diparalelkan hanya dengan menjalankan bagian-bagiannya secara bersamaan pada inti yang berbeda. Jika serangkaian instruksi yang akan dieksekusi tidak sepenuhnya dipesan, maka ini dapat memberikan percepatan. Paradigma yang memiliki nama negara (sel variabel) membuat ini lebih sulit karena setiap sel variabel memberlakukan urutan (urutan nilainya). Gaya pemrograman yang umum adalah memiliki agen serentak yang dihubungkan oleh aliran.

7 Pemrograman kendala

Dalam pemrograman kendala, kami mengungkapkan masalah yang akan dipecahkan sebagai masalah kepuasan kendala (CSP). CSP dapat dinyatakan sebagai berikut: dengan serangkaian variabel yang berkisar di domain yang terdefinisi dengan baik dan satu set batasan (hubungan logis) pada variabel tersebut, temukan penetapan nilai ke variabel yang memenuhi semua batasan. Pemrograman batasan adalah yang paling deklaratif dari semua paradigma pemrograman praktis. Programmer menentukan hasil dan sistem mencarinya. Penggunaan search harness ini

kesempatan buta untuk menemukan solusi: sistem dapat menemukan solusi yang sama sekali tidak diharapkan oleh programmer. Bab oleh Philippe Codognet menjelaskan mengapa ini berguna untuk penemuan artistik [8].

Pemrograman batasan berada pada tingkat abstraksi yang jauh lebih tinggi daripada semua paradigma lain dalam bab ini. Ini muncul dalam dua cara. Pertama, pemrograman kendala dapat memaksakan a *kondisi global* pada masalah: kondisi yang benar untuk solusi. Kedua, pemrograman kendala sebenarnya bisa *menemukan* solusi dalam waktu yang wajar, karena dapat menggunakan algoritma yang canggih untuk kendala yang diterapkan dan algoritma pencarian. Ini memberi pemecah banyak kekuatan. Misalnya, batasan pencarian jalur dapat menggunakan algoritme jalur terpendek, batasan perkalian dapat menggunakan algoritme faktorisasi prima, dan sebagainya. Karena kekuatannya dalam memaksakan kondisi lokal dan global, pemrograman kendala telah digunakan dalam komposisi yang dibantu komputer [3, 41].

Pemrograman dengan batasan sangat berbeda dari pemrograman dalam paradigma lain pada bab ini. Alih-alih menulis sekumpulan instruksi yang akan dieksekusi, programmer *model* masalah: merepresentasikan masalah menggunakan variabel dengan fungsi mereka, mendefinisikan masalah sebagai batasan pada variabel, memilih penyebar yang menerapkan batasan, dan mendefinisikan distribusi dan strategi pencarian. Untuk kecil

masalah kendala, model yang naif bekerja dengan baik. Untuk masalah besar, model dan heuristik harus dirancang dengan hati-hati, untuk mengurangi pencarian sebanyak mungkin dengan mengeksploitasi struktur dan properti masalah. Seni pemrograman kendala terdiri dari merancang model yang membuat masalah besar bisa dikerjakan.

Kekuatan dan of eksibilitas dari sistem pemrograman batasan bergantung pada ekspresi domain variabelnya, ekspresif dan kekuatan pemangkasan penyebar, dan kecerdasan pemecah CSP-nya. Sistem kendala awal didasarkan pada domain sederhana seperti pohon terbatas dan bilangan bulat. Sistem kendala modern telah menambahkan bilangan real dan baru-baru ini juga mengarahkan grafik sebagai domain.

Pemrograman batasan terkait erat dengan konkurensi deklaratif. Secara semantik, keduanya adalah aplikasi dari kerangka kerja pemrograman konkuren Saraswat [42]. Seperti konkurensi deklaratif, pemrograman kendala bersifat bersamaan dan deterministik. Ia hidup dalam dunia deterministik: untuk masukan tertentu ia menghitung keluaran yang diberikan. Ini berbeda dari konkurensi deklaratif dalam dua cara utama. Pertama, ini menggantikan variabel aliran data dengan batasan umum. Mengikat variabel aliran data, misalnya, X = V, dapat dilihat sebagai kendala kesetaraan: X adalah sama dengan V. Kedua, ia memiliki aliran kontrol yang lebih fleksibel: setiap kendala dijalankan di utasnya sendiri, yang membuatnya menjadi agen konkuren yang disebut *mualim*

(lihat Bagian 7.2). Hal ini memungkinkan pembatas untuk memangkas ruang pencarian dengan lebih baik.

7.1 Beberapa aplikasi pemrograman kendala

Pemrograman kendala memiliki aplikasi di banyak bidang, seperti kombinatorik, perencanaan, penjadwalan, pengoptimalan, dan pemrograman berorientasi tujuan. Aplikasi yang memungkinkan sangat bergantung pada domain variabel dan batasan yang diimplementasikan dalam solver. Masalah kombinatorial sederhana dapat diselesaikan dengan bilangan bulat. Domain variabel yang sesuai dengan integer disebut a *domain terbatas* karena mengandung satu set bilangan bulat yang terbatas. Saat kita mengatakan, misalnya, itu $x \in \{0, \cdots, 9\}$, yang kami maksud adalah solusi untuk x adalah elemen dari himpunan terbatas {0, · · ·, 9}. Jika kita memiliki delapan variabel *s, e, n, d, m, o, r, y,* semua di set {0, · · ·, 9}, maka kita dapat memodelkan teka-teki KIRIM + LEBIH BANYAK = UANG (di mana setiap huruf mewakili sebuah digit) dengan batasan tunggal $1000\,s + 100\,e + 10\,n + d + 1000\,m + 100\,o + 10\,r + e = 10000\,m + 1000\,o + 100\,n + 10\,e + y$.

Kami menambahkan kendala $s > 0$ dan $m > 0$ untuk memastikan digit pertama bukan nol dan pembatas *alldi ff ({ s, e, n, d, m, o, r, y})* untuk memastikan bahwa semua digit berbeda. Untuk memecahkan masalah ini secara cerdas, pemecah kendala hanya membutuhkan satu informasi lagi: heuristik yang dikenal sebagai strategi distribusi (lihat Bagian 7.2). Untuk contoh ini, heuristik sederhana yang disebut first-fail sudah cukup.

Domain terbatas adalah contoh sederhana dari domain diskrit. Sistem kendala juga telah dibangun menggunakan domain berkelanjutan. Misalnya, sistem Numerika menggunakan interval nyata dan dapat menyelesaikan masalah dengan persamaan diferensial [48]. Perbedaan antara teknik Numerica dan solusi numerik biasa dari persamaan diferensial (misalnya, Runge-Kutta atau metode prediktor-korektor) adalah bahwa pemecah kendala memberikan

menjamin: solusinya, jika ada dijamin berada dalam interval yang dihitung oleh pemecah. Metode biasa tidak memberikan jaminan tetapi hanya perkiraan kesalahan yang terikat.

Batasan grafik dan musik komputer

Penelitian terbaru sejak 2006 telah memperkenalkan domain diskrit yang sangat kuat, yaitu grafik arah. Variabel berkisar pada grafik yang diarahkan dan batasan yang menentukan kondisi

pada grafik. Ini dapat mencakup kondisi sederhana seperti ada atau tidak ada edge atau node. Tetapi yang membuat domain benar-benar menarik adalah domain ini juga dapat mencakup kondisi kompleks seperti penutupan transitif, keberadaan jalur dan dominator, dan isomorfisme subgraf [15, 40, 58]. Kondisi kompleks diimplementasikan oleh algoritma grafik yang canggih. Perpustakaan Gecode untuk batasan grafik sedang dalam persiapan sebagai bagian dari proyek MANCOOSI [20, 43].

Batasan grafik dapat digunakan dalam masalah apa pun yang solusinya melibatkan grafik. Proyek MANCOOSI menggunakannya untuk memecahkan masalah penginstalan paket untuk distribusi perangkat lunak sumber terbuka yang besar. Spiessens telah menggunakan batasan grafik untuk alasan propagasi otoritas dalam sistem aman [46]. Node dari grafik otoritas adalah subjek dan objek. Tepi dalam grafik otoritas menggambarkan izin: entitas memiliki hak untuk melakukan tindakan pada entitas lain. Jalur dalam grafik otoritas menggambarkan otoritas: entitas dapat melakukan tindakan, baik secara langsung maupun tidak langsung. Masalah propagasi otoritas dapat dirumuskan sebagai masalah grafik. Karena program kendala bersifat relasional, ini berfungsi di kedua arah:

Sebuah karya musik memiliki tatanan global. Skor musik dapat direpresentasikan sebagai grafik. Karena dua fakta ini, kami berhipotesis bahwa batasan grafik dapat menjadi primitif yang berguna untuk komposisi yang dibantu komputer. Misalnya, isomorfisma subgraf dapat digunakan untuk menemukan atau menerapkan tema di seluruh komposisi. Mungkin perlu merancang batasan grafik baru untuk musik komputer. Misalnya, dalam partitur musik, tema yang sama sering kali dapat ditemukan di tempat yang berbeda dan pada skala waktu yang berbeda, mungkin memberikan struktur fraktal pada partitur. Batasan global dapat dirancang untuk memaksakan kondisi ini.

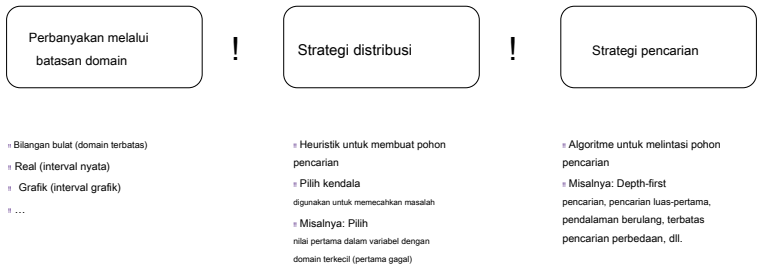
7.2 Bagaimana pemecah kendala bekerja

Pada prinsipnya, menyelesaikan CSP itu mudah: cukup hitung semua kemungkinan nilai untuk semua variabel dan uji apakah setiap enumerasi adalah solusi. Pendekatan naif ini sangat tidak praktis. Pemecah kendala praktis menggunakan teknik yang jauh lebih cerdas seperti pencarian lokal (dijelaskan dalam bab oleh Philippe Codognet [8]) atau algoritme penyebaran-penyebaran (dijelaskan di bagian ini). Yang terakhir ini mengurangi jumlah pencarian dengan menyebarkan dan mendistribusikan langkah-langkah (untuk informasi lebih lanjut lihat [47], yang menjelaskan perpustakaan Gecode):

- *Menyebarkan langkah:* Kurangi domain variabel dalam ukuran sebanyak mungkin menurut penyebar. SEBUAH *mualim* adalah agen konkuren yang mengimplementasikan batasan. Ini dipicu ketika domain dari salah satu argumennya berubah. Kemudian mencoba untuk lebih mengurangi domain argumennya sesuai dengan batasan yang diimplementasikannya. Para penyebar bisa saling memicu melalui argumen bersama. Mereka mengeksekusi sampai tidak ada lagi pengurangan yang mungkin (titik tetap). Ini mengarah pada tiga kemungkinan: solusi, kegagalan (tidak ada solusi), atau solusi yang tidak lengkap.

- *Distribusikan langkah:* Untuk setiap solusi yang tidak lengkap, pilih batasan *C* dan membagi masalah *P.* menjadi dua submasalah *P.* ∧ *C* dan *P.* ∧ ¬ *C.* Ini meningkatkan jumlah masalah yang harus diselesaikan, tetapi setiap masalah mungkin lebih mudah diselesaikan karena memiliki informasi tambahan (*C* atau ¬ *C).* Langkah ini adalah bentuk penelusuran paling primitif.

Algoritme kemudian dilanjutkan dengan langkah-langkah penyebaran untuk dua submasalah. Ini menciptakan pohon biner yang disebut pohon pencarian. Efisiensi algoritma distribusi-propagasi bergantung pada tiga faktor yang dapat dipilih secara independen (lihat Gambar 17):



Gambar 17. Pemecah kendala berdasarkan algoritme penyebaran-penyebaran

- *Propagasi melalui domain kendala.* Hal ini menentukan seberapa besar rambat (pruning) yang dilakukan oleh para dai. Ini tergantung pada dua faktor: kecanggihan para penyebar dan ekspresi domain kendala. Propagator dapat mengimplementasikan algoritma yang sangat canggih yang bergantung pada teori yang dalam

  hasil. Misalnya, penyebar perkalian A * B =: C dapat menggunakan algoritma faktorisasi untuk meningkatkan propagasi. Untuk bilangan bulat positif SEBUAH dan B, perkalian
  mualim A * B =: 12 akan berkurang menjadi A, B $\in \{$ 1, $\cdots$, 12} atau A, B $\in \{$ 1, 2, 3, 4, 6, 12}, bergantung pada apakah domain kendala dapat memiliki "lubang" atau tidak. Prop yang lebih baik
  agation dan domain kendala yang lebih ekspresif mengurangi jumlah langkah distribusi (lebih sedikit pencarian) dengan biaya lebih banyak propagasi (lebih banyak kesimpulan). Bergantung pada masalahnya, ini mungkin atau mungkin juga bukan pertukaran yang baik.

- *Strategi distribusi.* Heuristik ini mendefinisikan bagaimana kendala tersebut *C* dipilih untuk setiap langkah distribusi. Pilihan yang bagus *C* tergantung pada struktur masalah dan distribusi solusinya. Misalnya, heuristik gagal pertama menemukan variabel dengan domain terkecil dan memilih nilai pertama dalam domain ini.

- *Strategi pencarian.* Heuristik ini mendefinisikan bagaimana pohon pencarian dilintasi. Traversal tipikal adalah depth-first atau breadth-first, tetapi masih banyak traversal yang lebih canggih, seperti A *, pendalaman berulang, dan perbedaan terbatas. A * menemukan jalur terpendek dengan memandu pencarian dengan fungsi heuristik: jarak aktual yang ditempuh ditambah perkiraan jarak yang tersisa ke tujuan. Estimasi tidak boleh lebih besar dari jarak tersisa sebenarnya. Pendalaman berulang dan perbedaan terbatas melakukan penelusuran yang semakin luas, dimulai dengan batas 1 dan menambah batas setelah setiap traversal. Pendalaman berulang menggunakan batas kedalaman dan perbedaan terbatas menggunakan batas ketidaksesuaian (misalnya, jumlah perbedaan sehubungan dengan jalur kedalaman pertama).

8 Kesimpulan dan saran untuk melangkah lebih jauh

Bab ini memberikan gambaran singkat tentang paradigma pemrograman utama dan konsepnya. Bahasa pemrograman harus mendukung beberapa paradigma karena masalah yang berbeda membutuhkan konsep yang berbeda untuk menyelesaikannya. Kami menunjukkan beberapa cara untuk mencapai ini: bahasa paradigma ganda yang mendukung dua paradigma dan bahasa definitif dengan empat paradigma dalam struktur berlapis. Setiap paradigma memiliki “jiwa” -nya sendiri yang hanya bisa dipahami dengan benar-benar menggunakan paradigma tersebut. Kami merekomendasikan agar Anda menjelajahi paradigma tersebut

dengan benar-benar memprogramnya. Setiap paradigma memiliki bahasa pemrograman yang mendukungnya dengan baik dengan komunitas dan pendukungnya. Misalnya, kami merekomendasikan Haskell untuk pemrograman fungsional malas [28], Erlang untuk konkurensi penyampaian pesan [6], SQL untuk pemrograman transaksional, Esterel untuk pemrograman sinkron diskrit [7], dan Oz untuk pemrograman konkurensi dan batasan deklaratif [50].

Jika Anda ingin menjelajahi bagaimana menggunakan paradigma yang berbeda dalam satu program, kami merekomendasikan bahasa multiparadigma seperti Oz [34], Alice [38], Curry [4], atau CIAO [23]. Masing-masing dari keempat memiliki pendekatan yang berbeda; pilih yang paling kamu suka! Untuk Oz ada buku teks dan situs Web yang memiliki banyak materi termasuk kursus lengkap dalam bahasa Inggris dan Prancis [50, 51]. Ada perbedaan besar antara bahasa yang didesain dari awal menjadi multiparadigma (seperti Oz) dan bahasa yang mengandung banyak konsep pemrograman (seperti Common Lisp). Bahasa multiparadigma yang sebenarnya difaktorkan: dimungkinkan untuk memprogram dalam satu paradigma tanpa campur tangan dari paradigma lain.

Ucapan Terima Kasih

Bab ini ditulis selama cuti penulis di IRCAM. Bagian dari pekerjaan yang dilaporkan di sini didanai oleh Uni Eropa dalam proyek SELFMAN (kontrak program kerangka kerja keenam 34084) dan proyek MANCOOSI (perjanjian hibah program kerangka ketujuh 214898). Penulis berterima kasih kepada Arshia Cont, pengembang Antescofo, dan G´érard Assayag serta anggota grup RepMus lainnya di IRCAM atas diskusi menarik yang menghasilkan beberapa kesimpulan dalam bab ini.

Bibliografi

[1] Agon C., Assayag G. dan Bresson B., *OpenMusic 6.0.6,* IRCAM, Perwakilan Musik tations Research Group, 2008.

[2] Altshuller G., “Algoritma Inovasi: TRIZ, Inovasi Sistematis dan Teknologi- nical Creativity, ”Technical Innovation Center, Inc. (Diterjemahkan dari bahasa Rusia oleh Lev Shulyak dan Steven Rodman), 2005.

[3] Anders T., *Menulis Musik dengan Aturan Menulis: Desain dan Penggunaan Generik Sistem Batasan Musik,* Ph.D. disertasi, Queen's University, Belfast, Irlandia Utara, Februari 2007.

[4] Hanus A., Hanus S., dan Hanus M., *Kari: Pengantar Tutorial,* Desember 2007. Lihat www.curry-language.org.

[5] Armstrong J., *Membuat Sistem Terdistribusi yang Andal di Hadirat Software Er- rors,* Ph.D. disertasi, Royal Institute of Technology (KTH), Stockholm, Swedia, November 2003.

[6] Armstrong J., Williams M., Wikstr¨om C. dan Virding R., *Pemrograman Bersamaan di Erlang,* Prentice-Hall, 1996. Lihat www.erlang.org.

[7] Berry G., *Esterel v5 Language Primer – Versi 5.21 rilis 2.0,* École des Tambang dan INRIA, April 1999.

[8] Codognet P. "Kombinatorik, Keacakan, dan Seni Menciptakan", *Baru Com- Paradigma putasional untuk Musik Komputer,* Assayag G. dan Gerzso A. (eds.), IR- CAM / Delatour France, 2009.

[9] Cointe P. "Mendesain Bahasa Terbuka: Perspektif Historis", *Baru Paradigma Komputasi untuk Musik Komputer,* Assayag G. dan Gerzso A. (eds.), IRCAM / Delatour France, 2009.

[10] Collet R., *Batasan Transparansi Jaringan dalam Bahasa Pemrograman Terdistribusi- pengukur,* Ph.D. disertasi, Universit´é catholique de Louvain, Louvain-la-Neuve, Belgia, Desember 2007.

[11] Lanjutan A., “Antescofo: Sinkronisasi Antisipatif dan Kontrol Pa- rameters di Computer Music ”, *ICMC 2008,* Belfast, Irlandia Utara, Agustus 2008.

[12] Cooper GH, *Mengintegrasikan Evaluasi Aliran Data ke dalam Panggilan Praktis Tingkat Tinggi by-Value Language,* Ph.D. disertasi, Brown University, Providence, Rhode Is- land, Mei 2008.

[13] Dennis JB dan Van Horn EC, “Semantik Pemrograman untuk Multiprogrammed Perhitungan ”, *Komunikasi ACM,* 9 (3), Maret 1966.

[14] Dijkstra EW, *Sebuah Disiplin Pemrograman,* Prentice-Hall, 1976.

[15] Dooms G., *Domain Komputasi CP (Grafik) dalam Pemrograman Batasan,* Ph.D. disertasi, Universit´é catholique de Louvain, Louvain-la-Neuve, Belgia, 2006.

[16] Elliott C., *Reaktivitas Fungsional Cukup Efisien,* Laporan teknis LambdaPix 2008-01, April 2008.

[17] Ericsson, *Buka Telecom Platform — Panduan Pengguna, Manual Referensi, Instalasi Panduan, Bagian Khusus OS,* Telefonaktiebolaget LM Ericsson, Stockholm, Swedia, 1996.

[18] Felleisen M., "Tentang Kekuatan Ekspresif Bahasa Pemrograman", di *Euro ke-3 Pean Symposium on Programming (ESOP 1990),* Mei 1990, hlm.134-151.

[19] Fran¸cois, ARJ, "Waktu dan Persepsi dalam Musik dan Komputasi", *Baru Com- Paradigma putasional untuk Musik Komputer,* Assayag G. dan Gerzso A. (eds.), IR- CAM / Delatour France, 2009.

[20] Guti´érrez G., *Graph Constraint Library untuk Gecode,* Proyek MANCOOSI menyampaikan- bisa, lihat www.mancoosi.org. Dalam persiapan, 2009.

[21] Halbwachs N. dan Pascal R., *Tutorial Kilau,* Januari 2002.

[22] Seif H., Van Roy P., Merek P., dan Schulte C., “Bahasa Pemrograman untuk Dis- Aplikasi penghormatan ”, *Jurnal Komputasi Generasi Baru,* 16 (3), hlm.223-261, Mei 1998.

[23] MV Hermenegildo, Bueno F., Carro M., L´ópez P., Morales JF, dan Puebla G., “Tinjauan tentang Bahasa Multiparadigma Ciao dan Lingkungan Pengembangan Program dan Filosofi Desainnya”, Springer LNCS 5065 (Esai Didedikasikan untuk Ugo Montanari pada Acara Ulang Tahun ke-65), hlm. 209-237, 2008.

[24] Hewitt C., Uskup P. dan Steiger R., “Formalisme AKTOR Modular Universal untuk Intelijen Arti fi cial ”. Di *Konferensi Bersama Internasional ke-3 tentang Intelijen Arti (IJCAI),* hlm. 235-245, Agustus 1973.

[25] MOBIL Hoare, *Mengkomunikasikan Proses Berurutan,* Prentice-Hall, 1985.

[26] Houssais B., *Bahasa Pemrograman Sinkron SIGNAL: Tutorial,* IRISA, April 2002.

[27] Hudak P., Courtney A., Nilsson H., dan Peterson J., “Panah, Robot, dan Fungsi- Pemrograman Reaktif Nasional ”. Di *Sekolah musim panas tentang pemrograman Fungsional Lanjutan,* Springer LNCS 2638, hlm.159-187, 2003.

[28] Hudak P., Peterson J. dan Fasel J., *A Gentle Introduction to Haskell, Versi 98,* Lihat www.haskell.org/tutorial.

[29] Lea D., *Pemrograman Bersamaan di Java: Prinsip dan Pola Desain,* Pren- tice Hall, 1999.

[30] Lienhard M., Schmitt A. dan Stefani J.-B., *Oz / K: Bahasa Kernel untuk Pemrograman Terbuka Berbasis Komponen.* Di *Konferensi Internasional Keenam tentang Pemrograman Umum dan Rekayasa Komponen (GPCE'07),* Oktober 2007.

[31] Miller MS, “Komposisi Kuat: Menuju Pendekatan Terpadu untuk Kontrol Akses dan Kontrol Konkurensi, ”Ph.D. disertasi, Universitas Johns Hopkins, Balti- more, Maryland, Mei 2006.

[32] Miller M S., Stiegler M., Tutup T., Frantz B., Yee KP., Morningstar C., Shapiro J., Hardy N., Tribble E .D., Barnes D., Bornstien D., Wilcox-O'Hearn B., Stanley T., Reid K. dan Bacon D., *E: Kemampuan Terdistribusi Sumber Terbuka,* 2001. Lihat www.erights.org.

[33] Moore GE, *Hukum Moore,* Wikipedia, ensiklopedia gratis, 2009. [34] Mozart Consortium, *Sistem Pemrograman Mozart,* Versi 1.4.0 dirilis Juli 2008. Lihat www.mozart-oz.org.

[35] Olarte C., Rueda C. dan Valencia FD, “Concurrent Constraint Calculi: a Declar- Paradigma atif untuk Pemodelan Sistem Musik ”. *Paradigma Komputasi Baru untuk Musik Komputer,* Assayag G. dan Gerzso A. (eds.), IRCAM / Delatour France, 2009.

[36] Orlarey Y., Fober D. dan Letz S., "Aspek Sintaks dan Semantik Faust". *Komput Lunak. 8 (9),* November 2004, hlm.623-632.

[37] Orlarey Y., Fober D. dan Letz S., “FAUST: sebuah Pendekatan Fungsional yang Efisien untuk Pemrograman DSP ”. *Paradigma Komputasi Baru untuk Musik Komputer,* As- sayag G. dan Gerzso A. (eds.), IRCAM / Delatour France, 2009.

[38] Lab Sistem Pemrograman, Universitas Saarland, *Alice ML Versi 1.4.0* Lihat www.ps.uni-sb.de/alice.

[39] Puckette M., *dkk., Max / MSP Versi 5.0.6,* 2009. Lihat www.cycling74.com.

[40] Quesada L., *Memecahkan Masalah Grafik Batasan menggunakan Batasan Jangkauan berdasarkan Penutupan Transitif dan Dominator,* Ph.D. disertasi, Universit´é catholique de Louvain, Louvain-la-Neuve, Belgia, November 2006.

[41] Rueda C., Alvarez G., Quesada L., Tamura G., Valencia F., D˜iaz JF dan As- sayag G., "Mengintegrasikan Batasan dan Objek Bersamaan dalam Aplikasi Musik: Kalkulus dan Bahasa Visualnya", *Jurnal Kendala,* Kluwer Academic Pub-lishers, nomor 6, hlm. 21-52, 2001.

[42] Saraswat, VA *Pemrograman Batasan Bersamaan,* MIT Press, Cambridge, MA, 1993.

[43] Schulte C., Lagerkvist M. dan Tack G., *Gecode: Pengembangan Batasan Generik Lingkungan Hidup,* 2006. Lihat www.gecode.org.

[44] proyek SELFMAN. *Manajemen Mandiri untuk Sistem Terdistribusi Skala Besar berdasarkan Jaringan dan Komponen Overlay Terstruktur,* Program Kerangka Kerja Komisi Eropa ke-6, 2006-2009. Lihat www.ist-selfman.org.

[45] Smolka G., Schulte C. dan Van Roy P., *PERDIO – Persistent dan Distributed Pro- tata bahasa di Oz,* Proposal proyek BMBF, DFKI, Saarbrücken, Jerman. Feb. 1995.

[46] Spiessens A., *Pola Kolaborasi yang Aman,* Ph.D. disertasi, Universit´é catholique de Louvain, Louvain-la-Neuve, Belgia, Februari 2007.

[47] Taktik G., *Propagasi Kendala: Model, Teknik, Implementasi,* Ph.D. disertasi, Universitas Saarland, Saarbrücken, Jerman, Januari 2009.

[48] Van Hentenryck P., "Pengantar yang Lembut tentang NUMERICA", *Artif. Intell.* 103 (1-2), Agustus 1998, hlm.209-235.

[49] Van Roy P., "Manajemen Mandiri dan Masa Depan Desain Perangkat Lunak", *Interna-Lokakarya nasional tentang Aspek Formal Komponen Perangkat Lunak (FACS 2006),* ENTCS volume 182, Juni 2007, halaman 201-217.

[50] Van Roy P. dan Seif H., *Konsep, Teknik, dan Model Program Komputer-ming,* MIT Press, Cambridge, MA, 2004. Lihat ctm.info.ucl.ac.be.

[51] Van Roy P. dan Seif H., *Pemrograman: Konsep, Teknik, dan Mod`èles (* di Prancis), Dunod Éditeur, 2007. Lihat ctm.info.ucl.ac.be/fr.

[52] Van Roy P., "Konvergensi dalam Desain Bahasa: Kasus Petir Menyerang Empat Waktu di Tempat yang Sama ". Di *Simposium Internasional ke-8 tentang Pemrograman Fungsional dan Logika (FLOPS 2006),* Fuji Sosono, Jepang, Springer LNCS 3945, April 2006, hlm.2-12.

[53] Van Roy P., "Tantangan dan Peluang Banyak Prosesor: Mengapa Multi-Prosesor Inti Mudah dan Internet Itu Sulit ", Pernyataan posisi, dalam *ICMC 2008,* Belfast, Irlandia Utara, Agustus 2008.

[54] Van Roy P., "Mengatasi Kerapuhan Perangkat Lunak dengan Loop Umpan Balik yang Berinteraksi dan Transisi Fase Reversibel ", Dalam *Simposium BCS tentang Visi Ilmu Komputer,* London, Inggris, September 2008.

[55] Van Roy P., *Paradigma Pemrograman Utama,* Versi poster 1.08.1 Lihat www.info.ucl.ac.be/~pvr/paradigms.html, 2008.

[56] Weinberg G., *Pengantar Berpikir Sistem Umum,* Dorset House Publishing Co., 1977 (edisi ulang tahun perak 2001).

[57] Wiger U., "Peningkatan Empat Kali Lipat dalam Produktivitas dan Kualitas – Kekuatan-Industri Pemrograman Rungsional dalam Produk-Produk Kelas Telekomunikasi ", Dalam *Prosiding Lokakarya 2001 tentang Desain Formal Sistem Tertanam Kritis Keselamatan,* 2001.

[58] Zampelli S., *Pendekatan Pemrograman Kendala untuk Subgraf Isomorfisme,* Ph.D. disertasi, Universit´é catholique de Louvain, Louvain-la-Neuve, Belgia, Juni 2008.