Semântica de Linguagens de Programação

notas de aula – versão 32

Jerônimo C. Pellegrini

9 de fevereiro de 2019

Sumário

Sobre este texto

Este é um texto introdutório em Semântica de Linguagens de Programação. Os pré-requisitos são conhecimento básico em Lógica e Linguagens Formais, um pouco de maturidade para trabalhar com conceitos abstratos, e familiaridade com linguagens de programação. Há Apêndices tratando brevemente de Linguagens Formais e Compilação, mas muitíssimo resumidamente.

Nomenclatura

Neste texto usamos marcadores para final de definições (⧫), exemplos (◀) e demonstrações (□).

Exercícios acompanhados de uma engrenagem,  envolvem implementação.

$(\alpha)^m$	concatenação iterada de palavra, página 99
$(f \quad x)$	notação λ: aplicação da função f ao argumento x, página 7
$\approx$	relação de bissimulação, página 62
$\bot_X, \bot$	menor elemento de um domínio, página 31
$\langle \cdot, \cdot \rangle$	par ordenado (do tipo produto), página 84
$\langle c, \ldots \rangle$	configuração de um programa em execução, página 51
$\stackrel{\text{def}}{=}$	"é definido como", página 11
$\mathcal{P}$	significado de procedimentos (função semântica), página 41
$\mathbf{Decl}_P$	declarações de procedimentos (domínio sintático), página 41
$\mathcal{V}$	significado de variáveis (função semântica), página 40
$\mathbf{Decl}_V$	declarações de variáveis (domínio sintático), página 40
$\emptyset$	processo nulo (CCS), página 88
$\mathbf{Env}_P$	ambientes de procedimento (domínio semântico), página 41
$\mathbf{Env}_V$	ambientes de variáveis (domínio semântico), página 38
$\text{lfp}\, f$	Ponto fixo mínimo de f, página 33
$\text{gr}\, f$	gráfico da função f, página 7
$\lambda x \cdot \ldots$	notação λ: função mapeando $x \mapsto \ldots$, página 7
$\lambda^{\rightarrow}$	λ-Cálculo simplesmente tipado, página 81
$\rightarrowtail$	β-redução no λ-Cálculo, página 70
$\lceil M \rceil$	remoção de tipo do termo M, página 83
$\mathbf{Loc}$	locais de memória (domínio semântico), página 38
$\gg$	passo de execução de máquina abstrata, página 60

$\Downarrow$ aplicação de regra de semântica natural, página 51

νaP restrição de ação (CCS), página 88

ω elemento maior que qualquer número, página 31

π_i função projeção, página 34

π_i operador projeção, página 84

Σ Conjunto de estados de um programa, página 35

Σ Um alfabeto, página 27

$\sigma \times \tau$ tipo produto, página 84

$\sqcup$ supremo, página 30

$\sqsubseteq$ alguma relação de ordem parcial, página 26

Sto memórias (domínio semântico), página 38

$\rightarrow$ aplicação de regra de semântica estrutural, página 51

sup supremo, página 30

ε palavra vazia, página 99

$\varphi[x/y]$ substituição de y por x na fórmula φ, página 8

$\text{wlp}(s, Q)$ pré-condição libera mais fraca, página 67

$A + B$ união disjunta de domínios, página 35

$a.P$ ação a prefixada no processo P (CCS), página 88

$A \rightarrow B$ domínio de funções, página 35

a^*, A^* fecho de Kleene, página 100

a^+, A^+ fecho de Kleene sem palavra vazia, página 100

$P + Q$ escolha entre processos (CCS), página 88

$P[b/a]$ renomeação de ação (CCS), página 88

$t \uparrow$ o λ-termo t diverge, página 71

$t \equiv_\alpha t'$ α-equivalência, página 70

$VLG(t)$ conjunto de variáveis ligadas em um λ-termo, página 70

$VLV(t)$ conjunto de variáveis livres em um λ-termo, página 70

X^* Fecho transitivo. Se X é alfabeto, X^* é o conjunto de palavras sobre X., página 27

$X_\perp$ ordem parcial X elevada com menor elemento $\perp$, página 31

$\mathcal{G}$ Conjunto de todos os grafos., página 28

AFD autômato finito determinístico, página 105

AFN autômato finito não-determinístico, página 105

Capítulo 1

Introdução

A *sintaxe* de uma linguagem de programação determina como a estrutura gramatical de um programa deve ser – sem qualquer preocupação com o significado de cada trecho, com compatibilidade de tipos. A *semântica* de uma linguagem determina o significado formal de cada trecho de código. Como exemplo extremamente simples, tomamos a linguagem que usamos usualmente para expressões. A expressão

$$) + 3\{/ + /5$$

certamente não faz sentido – e o problema com ela é *sintático*, uma vez que há delimitadores (parênteses e chaves) dispostos de forma incorreta, e operadores em sequência, sem operandos. A expressão

$$2 + x = \textbf{true}$$

também não faz sentido, mas está *sintaticamente* correta. O problema dela é que o lado esquerdo só pode denotar números, e o lado direito denota um valor-verdade[1]. A expressão está *semanticamente* errada.

Este texto é sobre a descrição da semântica de linguagens de programação – deixaremos de lado os aspectos sintáticos das linguagens.

1.1 Abordagens para semântica

Trataremos neste texto de três abordagens diferentes para descrever a semântica de linguagens. Nesta seção damos apenas uma idéia de como é cada uma delas.

- **Semântica Operacional** – a linguagem é descrita através de sua interpretação em uma máquina virtual hipotética. Esta máquina virtual é construída de forma a facilitar a descrição formal de linguagens de programação e as demonstrações de suas propriedades. Damos um exemplo. O par $\langle c, \sigma \rangle$ é uma *configuração* de uma máquina, e significa que temos o comando (ou trecho de programa c para ser executado, no estado σ (um *estado* consiste dos valores atribuídos às variáveis). A seguir temos uma regra:

$$\frac{\langle c_1, \sigma \rangle \to \sigma', \langle c_2, \sigma' \rangle \to \sigma''}{\langle c_1; c_2, \sigma \rangle \to \sigma''}$$

  A regra diz que se:

  - quando executamos c_1 no estado σ terminamos no estado σ';
  - quando executamos c_2 no estado σ' terminamos no estado σ'',

  então executar a sequência $c_1; c_2$ no estado σ nos levará ao estado σ''.

[1] Há linguagens de programação que permitem misturar tipos de dados desta forma, a ainda de outras, surpreendentes: em Perl, `"2"+ "3x"` vale cinco, assim como `3+"2abc10"`.

- **Semântica Denotacional** – cada trecho de programa é descrito como uma função. Um programa, portanto, mapeia um estado (valores de variáveis e outros elementos relevantes à execução do programa) em um novo estado.

$$\mathcal{C}[\![c_1; c_2]\!] = \text{se } c_1 \text{ para, } c_2(c_1),$$
$$\text{senão } \bot$$

  Aqui, $\mathcal{C}[\![c]\!]$ denota o significado semântico do comando c; este significado é uma função que leva estados do programa em outros estados. A regra diz que o significado de $c_1; c_2$ é $\bot$ quando c_1 entra em *loop*; caso contrário, o estado é a função composta $c_1 \circ c_2$. A semântica denotacional de uma linguagem imperativa, por exemplo, é a descrição da função associada a cada comando.

- **Semântica Axiomática** – o significado de cada trecho de um programa é descrito pelo seu efeito no estado do programa – o estado é composto pelos valores das variáveis. O efeito de cada trecho é descrito por pré-condições e pós-condições. A notação é $\{P\}c\{Q\}$, onde P são as pré-condições e Q são as pós-condições. Damos um pequeno exemplo:

$$\{x \leq y\} y := y + 1 \{x < y\}$$

  Aqui declaramos que em qualquer estado onde valha $x \leq y$, se executarmos "$y := y + 1$", passará a valer $x < y$. A semântica axiomática de uma linguagem é um conjunto de regras de pré-condição e pós-condição.

Nenhuma das diferentes abordagens para semântica é melhor que as outras. Elas têm características diferentes, e são aplicáveis em situações diferentes – são, portanto, complementares[2].

1.2 Sintaxe: concreta e abstrata

A *sintaxe concreta* de uma linguagem de programação é dada por uma gramática livre de contexto, e é útil na análise sintática de programas da linguagem. Na sintaxe concreta incluímos elementos cuja função é remover ambiguidades, como parênteses, delimitadores do tipo "begin/end", e regras para forçar precedência de operadores. Usualmente, a sintaxe concreta de uma linguagem de programação é dada pela especificação de uma gramática livre de contexto, usando a notação Backus-Naur (BNF).

A *sintaxe abstrata* de uma linguagem é normalmente mais simples, e pode não ser suficiente para realizar a análise sintática (*parsing*) da linguagem. Isto porque não inclui elementos extras que normalmente existem na gramática concreta para eliminar ambiguidades. A estrutura de dados que descreve a árvore abstrata construída por um compilador é um exemplo de sintaxe abstrata.

Exemplo 1.1. Um comando de repetição com sintaxe concreta

⟨*while*⟩ ::= while ⟨*exp-bool*⟩ do begin ⟨*cmd*⟩ end

poderia ter sua sintaxe abstrata descrita pela gramática a seguir.

⟨*while*⟩ ::= while ⟨*exp-bool*⟩ do ⟨*cmd*⟩ ◀

Exemplo 1.2. A seguir temos a gramática concreta de um pequeno trecho de linguagem, que inclui apenas sequência de comandos, atribuições e decisões simples.

⟨*cmds*⟩ ::= ⟨*cmd*⟩ | { ⟨*cmd*⟩ ; ⟨*cmds*⟩ }

⟨*cmd*⟩ ::= id := ⟨*expr*⟩
| ⟨*if*⟩ ⟨*bool*⟩ ⟨*then*⟩ ⟨*cmds*⟩

[2] Michael Gordon [Gor79] deixa este ponto tão claro quanto possível, quando diz que "tentar determinar, por exemplo, se a semântica denotacional é melhor do que a axiomática é como tentar determinar se Físico-Química é melhor que química orgânica".

⟨*expr*⟩ ::= ⟨*expr*⟩ + ⟨*ter*⟩ | ⟨*ter*⟩

⟨*ter*⟩ ::= ⟨*ter*⟩ * ⟨*fa*⟩ | ⟨*fa*⟩

⟨*fa*⟩ ::= (⟨*epxr*⟩) | id | num

Veja que há regras para termos e fatores, para que se possa determinar, durante a análise sintática, a precedência de operadores. Também há duas regras para comandos, *cmd* e *cmds*, para permitir o agrupamento de comandos.

A gramática abstrata para a mesma linguagem pode ser descrita como segue.

⟨*cmd*⟩ ::= ⟨*cmd*⟩ ; ⟨*cmd*⟩
| id := ⟨*expr*⟩
| ⟨*if*⟩ ⟨*bool*⟩ ⟨*then*⟩ ⟨*cmd*⟩

⟨*expr*⟩ ::= ⟨*expr*⟩ + ⟨*expr*⟩
| ⟨*expr*⟩ * ⟨*expr*⟩
| id
| num

Observamos que há uma regra para sequenciamento de comandos (a primeira para o símbolo ⟨*cmd*⟩), mas não há delimitadores do tipo "begin/end". As regras para expressões não definem precedência de operadores, e não incluem parênteses para agrupamento.

Como não nos interessa a grafia exata de palavras reservadas, podemos também escrever a mesma gramática abstrata da seguinte forma.

⟨*C*⟩ ::= ⟨*C*⟩; ⟨*C*⟩
| id ← ⟨*E*⟩
| ⟨*B*⟩ ⇒ ⟨*C*⟩

⟨*E*⟩ ::= ⟨*E*⟩+⟨*E*⟩
| ⟨*E*⟩*⟨*E*⟩
| id
| num

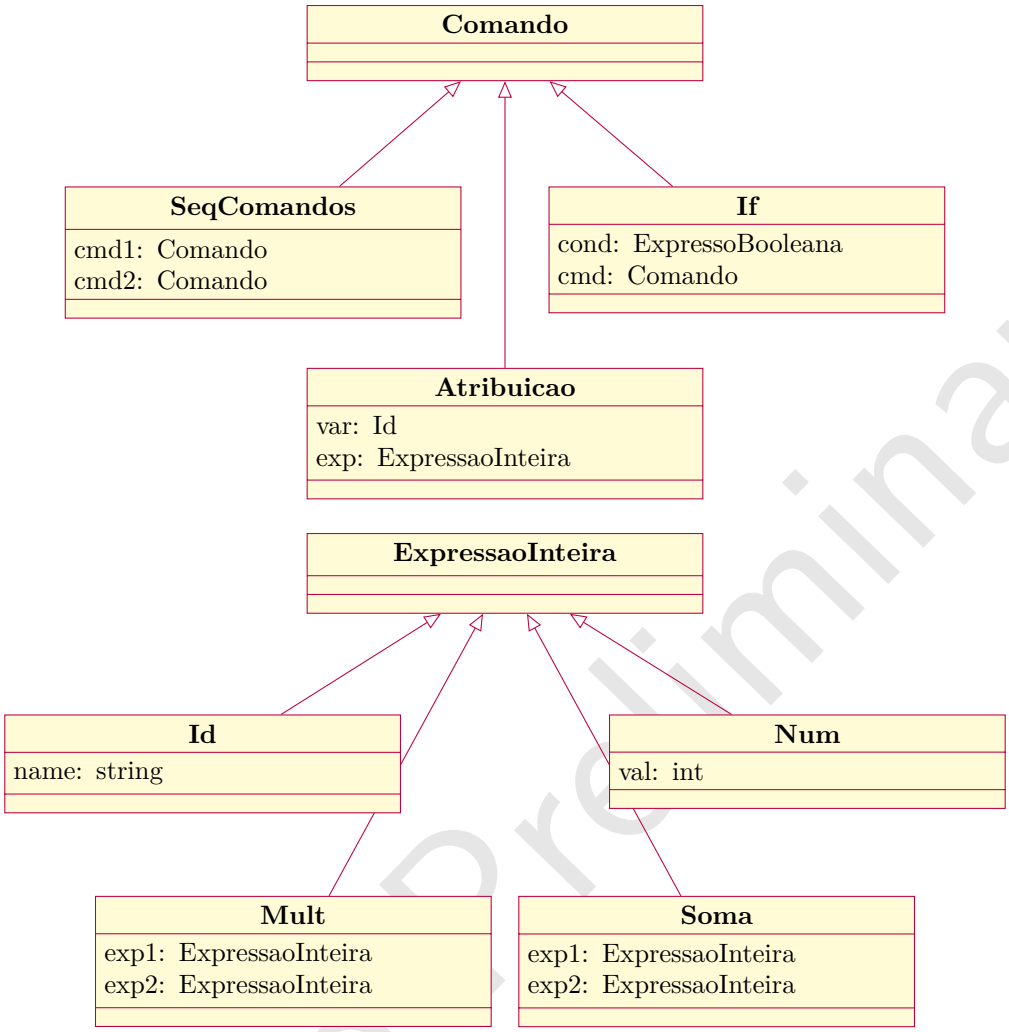
A sintaxe abstrata é também descrita pela estrutura de dados que poderia ser usada para construir a árvore durante a análise sintática. A seguir traduzimos esta mesma sintaxe abstrata para a linguagem Haskell e para um diagrama de classes UML.

Em Haskell:

```
data Comando = Sequencia Comando Comando
             | Atribuicao String ExpressaoInt
             | Condicional ExpressaoBool Comando

data ExpressaoInt = Soma     ExpressaoInt ExpressaoInt
                  | Mult     ExpressaoInt ExpressaoInt
                  | Valor    Int
                  | Variavel String
```

A seguir temos o diagrama UML, que definiria a estrutura para linguagens orientadas a objetos..



Tanto a estrutura de dados descrita em Haskell como o diagrama UML são descrições da sintaxe abstrata da linguagem dada anteriormente.

O exercício 1 pede a descrição da parte faltante desta estrutura de dados (expressões booleanas). ◀

A semântica de linguagens de programação é definida tendo como base a gramática abstrata da linguagem. Ambiguidades sintáticas são preocupação exclusiva da análise sintática, de que não tratamos neste texto.

Queremos definir funções que dão significado para construtos (cada uma das classes no diagrama UML) de uma linguagem. Diagramas visuais não facilitam esse trabalho, portanto usaremos uma descrição formal da sintaxe abstrata para definir os domínios das funções semânticas.

1.3 Sintaxe abstrata, estilo Scott-Strachey

Neste texto usaremos o estilo de descrição de gramática abstrata usual sem semântica formal, chamado de estilo *Scott-Strachey*[3].

Definição 1.3 (sintaxe abstrata, estilo Scott-Strachey)**.** A sintaxe abstrata no estilo Scott-Strachey é composta de

[3]Dana Scott e Christopher Strachey foram pioneiros da semântica denotacional.

i) uma lista de domínios sintáticos. Cada domínio sintático é uma parte da linguagem, gerada por um símbolo diferente.

ii) Uma meta-variável para cada domínio sintático, representando seus elementos.

iii) uma gramática livre de contexto, cujos símbolos não-terminais são as meta-variáveis sintáticas. Cada meta-variável representando um domínio deve ter pelo menos uma regra para sua expansão. ♦

Exemplo 1.4. Suponha que queiramos representar expressões booleanas e numéricas. Temos dois domínios simples: **Id**, contendo identificadores; e **Num**, contendo valores numéricos[4]. Os outros domínios sintáticos, **Aexp** e **Bexp** são definidos pela gramática abstrata.

$$A \in \textbf{Aexp}$$
$$B \in \textbf{Bexp}$$
$$I \in \textbf{Id}$$
$$N \in \textbf{Num}$$

As meta-variáveis que usamos são A, B, I, N. A gramática é mostrada a seguir.

$\langle A \rangle$ ::= N
| I
| $\langle A \rangle$ * $\langle A \rangle$
| $\langle A \rangle$ / $\langle A \rangle$
| $\langle A \rangle$ + $\langle A \rangle$
| $\langle A \rangle$ - $\langle A \rangle$

$\langle B \rangle$ ::= true | false
| $\langle A \rangle$ `=` $\langle A \rangle$
| $\langle A \rangle$ `<=` $\langle A \rangle$
| $\langle A \rangle$ `>=` $\langle A \rangle$
| $\langle B \rangle$ `and` $\langle B \rangle$
| $\langle B \rangle$ `or` $\langle B \rangle$
| `not` $\langle B \rangle$ ◄

Há notações ligeiramente diferentes, que tornam a gramática mais compacta. Por exemplo, pode-se definir os domínios sintáticos junto com a gramática:

$A \in \textbf{Aexp}$ ::=N
$|I$
$|A * A$
$|A/A$
$|A + A$
$|A - A$

1.4 Significado semântico

Ao tratarmos das semânticas denotacional e operacional, será central o conceito de *função semântica*.

Em diversas linguagens de programação há, por exemplo, diferentes expressões do mesmo número. Em C, por exemplo,

```
15,   (12 + 3),   0xF,   017
```

[4]Neste momento *não* nos interessa que conjunto numérico pretendemos representar – se é $\mathbb{N}$, $\mathbb{Q}$, ponto flutuante, ou algum outro.

representam o *mesmo* número – o inteiro 15 (se o último valor parece estranho, observe que um literal numérico começando com `0` em *C* é lido como número em representação *octal*[5]). Presumimos portanto que *números existem sem depender de sua representação*: podemos usar 4, *iv*, ou $1+1+1+1$ para representar a mesma entidade. Cada domínio sintático que definimos em nossas linguagens de programação deve ser mapeado em um conjunto de *significados* – ou seja, em um *domínio semântico*.

Definição 1.5 (função semântica)**.** Uma *função semântica* é uma função cujo domínio são os elementos da linguagem descrita por uma sintaxe abstrata. ♦

Exemplo 1.6. **Num** é um conjunto de *símbolos*, que representam números

$$\mathbf{Num} = \{0,1,2,3,4,5,6,7,8,9\}^*$$

Podemos definir a função semântica

$$\mathcal{N} : \mathbf{Num} \to \mathbb{N}$$

que mapeia palavras de **Num** em números naturais:

$$\mathcal{N}[\![0]\!] = \mathbf{0}$$
$$\mathcal{N}[\![5]\!] = \mathbf{5}$$
$$\mathcal{N}[\![20]\!] = \mathbf{20}$$

Já a função semântica

$$\mathcal{X} : \mathbf{xNum} \to \mathbb{N}$$

mapeia palavras em $\mathbf{xNum} = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}^*$ em $\mathbb{N}$:

$$\mathcal{X}[\![0]\!] = \mathbf{0}$$
$$\mathcal{X}[\![2]\!] = \mathbf{2}$$
$$\mathcal{X}[\![1A]\!] = \mathbf{26}$$

A função semântica

$$\mathcal{B} : \mathbf{bNum} \to \mathbb{N}$$

mapeia as palavras em $\{0,1\}^*$ em $\mathbb{N}$.

$$\mathcal{B}[\![0]\!] = \mathbf{0}$$
$$\mathcal{B}[\![1]\!] = \mathbf{1}$$
$$\mathcal{B}[\![101]\!] = \mathbf{5}$$ ◄

A semântica denotacional trata de encontrar funções semânticas para *todos* os domínios sintáticos de um programa.

1.5 Descrição de funções

Há duas maneiras importantes de descrever funções que usaremos extensivamente neste texto. As duas são expostas brevemente nesta seção.

[5] Números em representação octal (base oito) não são mais tão comuns quanto eram quando a linguagem *C* foi desenvolvida, mas o padrão ainda define literais octais. Pode-se verificar que `017` = 15 com o pequeno programa a seguir:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    printf("Valor de 017: %d\n",017);
    exit(EXIT_SUCCESS);
}
```

1.5.1 Gráfico de funções

Uma relação entre A e B pode ser representada como um conjunto de pares ordenados (porque é um subconjunto de $A \times B$; como funções são relações, também admitem esta representação.

Definição 1.7 (gráfico de função)**.** Seja $f : A \to B$ uma função. O *gráfico* de f é o conjunto de pares

$$\text{gr}(f) = \{(a, b) : \ a \in A, b = f(a)\}$$ ♦

Exemplo 1.8. O gráfico da função $f : \mathbb{N} \to \mathbb{N}$ tal que $f(n) = 2n$ é

$$\begin{aligned} \text{gr } f(x) &= \{(a, b) : \ a \in \mathbb{N}, b = 2a\} \\ &= \{(0, 0), (1, 2), (2, 4), \ldots\}. \end{aligned}$$ ◀

Exemplo 1.9. O gráfico da função sinal $s : \mathbb{R}^* \to \{-1, +1\}$, com $f(x) = |x|/x$, é

$$\text{gr}(s) = \{(x, +1) : \ x \in \mathbb{R}_+\} \cup \{(x, -1) : \ x \in \mathbb{R}_-\}.$$ ◀

Exemplo 1.10. Seja $\Sigma = \{a, b\}$. Defina a função $p : \Sigma^3 \to \{\textbf{true}, \textbf{false}\}$ como $p(\alpha) = \textbf{true}$ se α é palíndroma, e $p(\alpha) = \textbf{false}$ caso contrário. Então

$$gr(p) = \left\{ \begin{array}{l} (aaa, \textbf{true}), (aab, \textbf{false}), (aba, \textbf{true}), (abb, \textbf{false}), \\ (baa, \textbf{false}), (bab, \textbf{true}), (bba, \textbf{false}), (bbb, \textbf{true}) \end{array} \right\}.$$ ◀

1.5.2 Notação λ

Tomamos do λ-Cálculo uma notação conveniente para funções. O λ-Cálculo será apresentado no Capítulo 8, sendo que nesta seção apresentamos somente parte de sua notação.

Definição 1.11. Se uma função mapeia x em α, denotamos

$$\lambda x \cdot \alpha$$

Se $f = \lambda x \cdot \alpha$, a *aplicação desta função a um parâmetro* y é denotada

$$(f \quad y).$$ ♦

Claramente, $(f \quad y)$ é o mesmo que usualmente denotamos $f(y)$.

A notação λ nos permite tratar de funções sem dar-lhes nome; permite que usemos funções como valores para variáveis (como em "$f = \lambda x \cdot \ldots$"); e é padrão no estudo de semântica de linguagens[6].

Exemplo 1.12. A função dobro é denotada $\lambda x \cdot 2x$. Podemos escrever, portanto, que

$$\Big((\lambda x \cdot 2x) \quad 5\Big) = 10.$$

Ou ainda, se definirmos que

$$d = \lambda x \cdot 2x$$

então

$$(d \quad 15) = 30.$$ ◀

Para descrevermos funções com mais de um argumento, separamos os argumentos com vírgula. Por exemplo,

$$\lambda x, y, z \cdot \sqrt{x^2 + y^2 + z^2}$$

é a norma usual em $\mathbb{R}^3$.

[6] Também é usada em linguagens de programação da família Lisp para descrever funções anônimas: `(lambda (x) (* 2 x))` é a função que toma um argumento (`x`) e o multiplica por dois.

1.6 Aplicação parcial

Definição 1.13 (aplicação parcial)**.** Seja f uma função com n argumentos. Se fixarmos parte dos argumentos, teremos uma nova função, com $k < n$ argumentos. Esta nova função é uma *aplicação parcial* de f. ♦

Exemplo 1.14. Seja $f = \lambda x, y, z \cdot\ x + y + z$. Então $g = \lambda x, z \cdot\ x + 5 + z$ é aplicação parcial do segundo argumento da função f. ◄

1.7 Fórmulas

Nesta seção, usamos a palavra *fórmula* em amplo sentido: informalmente, uma fórmula aqui é uma expressão formal, possivelmente contendo variáveis. Note que isto inclui fórmulas em lógica de primeira (ou mais alta) ordem, expressões matemáticas em geral, λ-expressões, e trechos de programas.

Definição 1.15 (substituição)**.** Seja φ uma fórmula. Denotamos por $\varphi[y\ /\ x]$ a substituição de x por y em φ. ♦

Exemplo 1.16. Se $\varphi = (x + y)z^2$, então $\varphi[a/x]$ é $(a + y)z^2$. ◄

Exemplo 1.17. Se $\varphi = (p \wedge q) \vee r$, então $\varphi[p/q]$ é $(p \wedge p) \vee r$. ◄

Variáveis em uma fórmula podem ser *livres* ou *ligadas*. Esta é uma propriedade *sintática* de fórmulas, que usaremos com frequência. Não definiremos formalmente estes conceitos, mas tratamos informalmente deles a seguir.

Uma variável em uma fórmula é *livre* quando não é definida na própria fórmula, e portanto faz referência a alguma entidade fora dela.

Uma variável em uma fórmula é ligada quando é definida (ou vinculada) por um operador, como um quantificador universal ou existencial, integral, derivada, limite, somatório, etc – ou quando é argumento de uma função. Dizemos que estes operadores são *vinculantes*. Os operadores a seguir são exemplos de operadores vinculantes:

- $\sum_{n=1}^{k}$ vincula n, mas *não* k.
- $\int_a^b \ldots dx$ vincula x, mas *não* a e b.
- $\lim_{x \to K}$ vincula x, mas *não* K.
- $\exists x$ e $\forall x$ vinculam x.
- $\lambda x \cdot$ vincula x.

Esta noção estende-se naturalmente para trechos de programa: uma variável em um trecho é ligada se foi definida naquele trecho (mesmo que como argumento de uma função); é *livre* se foi definida fora do trecho (por exemplo, foi definida em escopo anterior, ou é global).

É essencial notar que se uma variável x é ligada em uma fórmula φ, e y não ocorre em φ, então podemos trocar φ por $\varphi[y/x]$, trocando x por y, sem que o significado da fórmula mude.

Se x for livre em φ, não podemos trocá-la por outro símbolo, porque x faz referência a uma entidade externa a φ.

Exemplo 1.18. Na fórmula

$$\forall x, y :\ \exists z, w :\ \ x + ay = zw$$

podemos fazer substituições nas variáveis w, x, y, z:

$$[m/w],\ [n/x],\ [o/y],\ [p/z]$$

obtendo a fórmula

$$\forall n, o\ \exists p, m\ n + ao = pm$$

com o *mesmo* significado – portanto estas variáveis estão ligadas nesta fórmula. No entanto, *não podemos substituir a variável a, porque ela faz referência a algo fora da fórmula* – portanto a é variável livre. ◀

Exemplo 1.19. A seguir está uma função em C que verifica se dois números estão próximos o suficiente. Verifica-se, ali, se $|a - b| < \varepsilon$. Como ε deve ser configurável e o programador decidiu que ele pode mudar em tempo de execução, ele foi definido como variável global.

```
bool proximos (double a, double b) {
    if (fabs(a-b) < epsilon)
        return true;
    else
        return false;
}
```

As variáveis `a` e `b` são ligadas, porque foram definidas nesta função como argumentos. Já a variável `epsilon` é livre. Podemos trocar as variáveis ligadas por outras não usadas na função. Por exemplo, usar `x` e `y` ao invés de `a` e `b` sem modificar o significado do programa:

```
bool proximos (double x, double y) {
    if (fabs(x-y) < epsilon)
        return true;
    else
        return false;
}
```

Não podemos, no entanto, mudar o nome `epsilon`, porque ele faz referência a uma entidade externa (uma variável global). ◀

Exemplo 1.20. Na fórmula

$$\exists z :\ z = \lambda x \cdot xyz$$

as variáveis x e z são ligadas, e y é livre. ◀

Exemplo 1.21. Na fórmula

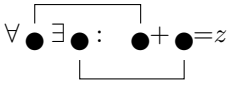
$$\int_0^t e^{tx} dt$$

a variável t é ligada, e x é livre. ◀

Um *diagrama de Stoy* é obtido de uma fórmula trocando as variáveis ligadas por marcas, e ligando cada variável à sua definição. Por exemplo, a fórmula

$$\forall x\ \exists y :\ \ x + y = z$$

tem o diagrama de Stoy a seguir.



1.8 Estado de programas

Estaremos interessados nos valores de variáveis e em como eles são modificados. Damos o nome de "*estado do programa*" ao conjunto de atribuições de valores a variáveis.

Definição 1.22 (estado)**.** O *estado* de um programa é uma função que mapeia variáveis em valores. ♦

Denotamos por $[x \mapsto 2, y \mapsto 3]$ o estado em que a variável x vale 2 e a variável y vale 3.

Usamos a mesma notação para modificações em estados: denotamos por $s[k \ / \ x]$ o estado s, modificando o valor da variável x para k.

Exemplo 1.23. Se $s = [x \mapsto 2, y \mapsto 3]$, então

$$s[10 \ / \ y] = [x \mapsto 2, y \mapsto 10].$$ ◀

Exemplo 1.24. A seguir temos um trecho de programa em *C* (ou Java). Presumimos que antes deste trecho, nenhuma variável tem valor definido, por isso o estado é vazio.

```
x = 5;
y = 2;

/* PRIMEIRO momento */

if x>0 {
  y = 2*x;
}

/* SEGUNDO momento */
```

No *PRIMEIRO* momento, o estado do programa é $[x \to 5, y \to 2]$. No *SEGUNDO* momento, é $[x \to 5, y \to 10]$. ◀

Exemplo 1.25. O estado de um programa é uma função de variáveis em valores. O gráfico do estado $[x \mapsto 4, y \mapsto 10]$ é $\{(x, 4), (y, 10)\}$. ◀

Exemplo 1.26. O trecho de programa "`a = 1;`" pode ser visto como uma função $P : \Sigma \to \Sigma$ que muda o estado de um programa, alterando o valor de a para um – ou seja, *a função estado, que mapeia variáveis em valores, passa a mapear a em um.*

$$\mathrm{gr}(P) = \{(\sigma, \sigma[1 \ / a]), \sigma \in \Sigma\}.$$ ◀

1.9 Árvores de Prova

Definição 1.27 (regra de inferência)**.** Uma *regra de inferência* consiste de um número de hipóteses e uma conclusão. Escrevemos regras de inferência posicionando as hipóteses acima e a conclusão abaixo de uma barra horizontal:

$$\frac{H_1 \quad H_2 \quad \cdots \quad H_n}{C}$$

Por exemplo, a regra *modus ponens* é descrita como

$$\frac{P \quad P \to Q}{Q}$$

Definição 1.28 (sistema de prova)**.** Um sistema de prova é um conjunto de regras de inferência. ♦

Nas regras de inferencia temos *variáveis lógicas*. Por exemplo, na regra *modus ponens*,

$$\frac{P \quad P \to Q}{Q}$$

P e Q são variáveis.

Ao encadearmos regras de inferência, obtemos *árvores de prova*.

Exemplo 1.29. Provaremos que

$$p \to t, p \wedge (q \vee r), s \vdash t \wedge s.$$

A prova pode ser escrita da seguinte maneira:

0. $p \to t$	(premissa)
1. $p \wedge (q \vee r)$	(premissa)
2. s	(premissa)
3. p	(1, eliminação de $\wedge$)
4. t	(0 e 3, *modus ponens*)
5. $t \wedge s$	(2 e 4, introdução de $\wedge$)

A árvore de prova é

$$\dfrac{\dfrac{\dfrac{p \wedge (q \vee r)}{p}}{t} \qquad s}{t \wedge s}$$

◀

1.10 Definições Recursivas e Indução Estrutural

Uma definição para uma variável x é *recursiva* se é da forma

$$x \stackrel{\text{def}}{=} \cdots x \cdots,$$

onde o símbolo $\stackrel{\text{def}}{=}$ significa "é definido como", e $\cdots x \cdots$ é uma expressão com referência à variável x que está sendo definida. Para simplicidade de notação, poderemos usar $=$ ao invés de $\stackrel{\text{def}}{=}$.

Exemplo 1.30. Podemos definir x como

$$x = \frac{2}{x},$$

ou seja, "x é o número que é igual a 2 dividido por ele mesmo". ◀

As definições recursivas que nos interessarão são em *casos* (normalmente casos base e casos recursivos, ou indutivos).

Definição 1.31 (definição recursiva)**.** Uma definição para uma variável x é *recursiva* se é da forma

$$x \stackrel{\text{def}}{=} \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k \mid \beta_1(x) \mid \beta_2(x) \mid \cdots \mid \beta_n(x),$$

onde $\mid$ significa "ou" (x é definido como α_1 *ou* α_2, etc). Os α_i são definições diretas para x, e os β_i são definições recursivas (separamos por conveniência os casos em que x é definida diretamente e recursivamente). ⧫

Exemplo 1.32. O fatorial de um número natural é definido como

$$0! = 1$$
$$n! = n(n-1)!$$

O símbolo fatorial (que estamos definindo) é usado em sua própria definição.

Aqui, α_1 é $0! = 1$, e β_1 é $n! = n(n-1)!$.

Para maior clareza, reescrevemos usando notação λ:

$$F = \lambda n \cdot \begin{cases} 1 & n = 0 \\ n(F\ \ n-1) & n > 0 \end{cases}$$

Entendemos que o símbolo F é o *nome de uma variável, cujo valor é a função fatorial, sendo definida.*

Agora fica evidente que o símbolo F está sendo definido como $\cdots\ F\ \cdots$, e a definição é corretamente classificada como recursiva de acordo com nossa definição. ◀

Exemplo 1.33. É usual encontrar a definição recursiva de sequências numéricas, como no exemplo a seguir.

$$a_1 = 2$$
$$a_n = 2a_{n-1} + 2$$

Aqui a "variável" que está sendo definida é "(a_n)", que é na realidade uma forma compacta de escrever $a : \mathbb{N} \to \mathbb{N}$. Esta definição recursiva define uma sequência, *que é uma função com domínio igual a* $\mathbb{N}$:

$$a = \lambda n \cdot \begin{cases} 2 & n = 1 \\ 2(a\ \ n-1) + 2 & n > 1 \end{cases} \quad ◀$$

Indução estrutural é uma técnica para demonstrar propriedades de estruturas definidas recursivamente.

Teorema 1.34 (indução estrutural)**.** *Seja x definido recursivamente,*

$$x \stackrel{\text{def}}{=} \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k \mid \beta_1(x) \mid \beta_2(x) \mid \cdots \mid \beta_n(x),$$

E seja $P(\cdot)$ uma proposição a respeito de objetos do tipo x. Se:

i) $P(\alpha_i)$ vale para todo i;

ii) $P(x)$ implica $P(\beta_i(x))$ para todo i,

então $P(x)$ vale para todo x.

Exemplo 1.35. Normalmente uma árvore é definida como um grafo conexo sem ciclos. Podemos, no entanto, dar uma definição recursiva:

- Um vértice isolado é uma árvore (a árvore trivial)
- Se T é uma árvore, então também será árvore o grafo obtido de T adicionando um vértice w e ligando w por uma aresta a algum vértice de T com grau estritamente menor que dois.

Provamos agora que toda árvore com e arestas tem exatamente $e + 1$ nós.

- *Base:* a árvore trivial tem um vértice e zero arestas.
- *Hipótese:* T tem e arestas, e $e + 1$ vértices.
- *Passo:* para obter uma nova árvore a partir de T, adicionamos *um* vértice e *uma* aresta, e teremos portanto $e + 1$ arestas e $e + 2$ vértices. ◀

Exemplo 1.36. Provaremos que na linguagem definida pela gramática
$C \to X \mid (C)$
$X \to Xa \mid Xb \mid c$
todas as palavras tem número par de parênteses.

As palavras são geradas por C. Como

$$C \to X \mid (C),$$

verificamos as palavras geradas por X e por (C).

i) Em $X \to Xa \mid Xb \mid c$, verificamos que não há produção de parênteses (temos 0 parênteses, par).

ii) (C) tem um número par de parênteses (2). Como, por hipótese de indução, C gera uma palavra com $2k$ parênteses, então teremos em (C) o total de $2k + 2$ parênteses. ◀

Podemos usar indução estrutural também em árvores de prova e diversas outras estruturas.

Exercícios

Ex. 1 — Complete a estrutura de dados do exemplo 1.2 com expressões booleanas.

Ex. 2 — Represente a função $f(x) = \frac{3g(x-1)}{2}$ usando notação λ.

Ex. 3 — Seja $\sigma = [x \mapsto 10; y \mapsto 3; z \mapsto 1]$. Compute o estado resultante:

$$\text{usando } \sigma' = \sigma[x/y; 12/z],$$
$$\Big((\lambda\sigma \cdot \sigma[z/y])\ \ \sigma'\Big)$$

Ex. 4 — Descreva o gráfico das funções a seguir.

a) $f(X) = \mathcal{P}(X)$, onde o domínio de f é o conjunto de todos os conjuntos finitos[7].

b) $f(x, y) = 3$, onde $x \in \mathbb{N}$, $y \in \mathbb{R}^2$.

c) $f(x) = \lambda y \cdot\ xy$, com $x, y \in \mathbb{R}$.

Ex. 5 — Prove que no triângulo de Sierpinski todo ângulo é de exatamente sessenta graus.

Ex. 6 — Prove que para toda árvore binária T com altura h,

$$2^{h+1} \geq |T| + 1,$$

onde $|T|$ é a quantidade de nós da árvore.

[7] Não há paradoxo aqui!

Capítulo 2

Uma linguagem-exemplo

Precisaremos de uma linguagem extremamente simples para exemplificar os conceitos apresentados. Faremos o mesmo que se faz em muitos livros ao apresentar semântica formal: usamos uma linguagem imperativa minimalista. O leitor reconhecerá esta mesma linguagem em outros livros (IMP em Winskel [Win94] e em Gunter [Gun92]; While em Nielson e Nielson [NN07], e em Stump [Stu14]; SIMP em Fernandez [Fer14]).

A linguagem que usaremos inicialmente será imperativa, e permitirá usar os seguintes cinco comandos.

- atribuição de valores a variáveis, denotada por :=. O comando $x := e$ significa "calcule o valor da expressão e e o armazene na variável x".
- **skip**, que nada faz.
- sequenciamento de comandos, denotado por ponto-e-vírgula. A expressão "$c_1; c_2$" significa "execute c_1, depois execute c_2".
- decisão, onde usaremos os tradicionais **if**, **then** e **else**.
- repetição, onde usaremos **while**.

Também incluímos na linguagem delimitadores para comandos compostos, para que possamos incluir mais de um comando dentro dos braços de **then**, **else** e **while**. *No entanto, estes delimitadores são de interesse apenas para a sintaxe concreta, e não os usaremos na sintaxe abstrata.*

2.1 Limitações

Inicialmente, nossa linguagem só terá variáveis inteiras. As expressões podem ser de dois tipos:

- *Aritméticas*, onde podem ser usadas as operações $+$, $-$ e $*$ (não incluímos divisão porque nossos números são inteiros)
- *Booleanas*, onde pode-se comparar expressões aritméticas com $=$, $\geq$ e $\leq$; e usar operadores lógicos `and`, `or` e `not`.

A linguagem também não permitirá usar comandos de entrada e saída, nem estruturas para definição de blocos, funções ou variáveis locais. Embora com isto ela fique distante de linguagens reais, nos permitirá manter o foco inicialmente em conceitos importantes. Presumiremos que *o programa está em algum estado, com valores já atribuídos às variáveis*, e que nos interessa *verificar o estado do programa após a execução, sem necessariamente retornar ou imprimir valores.*

2.2 Sintaxe

A sintaxe abstrata de nossa linguagem é dada a seguir.

- *Domínios sintáticos:*
  $C \in$ **Cmd**, comandos
  $A \in$ **Aexp**, expressões aritméticas
  $B \in$ **Bexp**, expressões booleanas
  $I \in$ **Id**, identificadores
  $Z \in$ **Int**, números inteiros

- *Gramática abstrata:*

  $\langle C \rangle$ ::= `skip`
  | $\langle I \rangle$ `:=` $\langle A \rangle$
  | $\langle C \rangle$ `;` $\langle C \rangle$
  | `if` $\langle B \rangle$ `then` $\langle C \rangle$ `else` $\langle C \rangle$
  | `while` $\langle B \rangle$ `do` $\langle C \rangle$

  $\langle A \rangle$ ::= Z
  | I
  | $\langle A \rangle$ * $\langle A \rangle$
  | $\langle A \rangle$ + $\langle A \rangle$
  | $\langle A \rangle$ - $\langle A \rangle$

  $\langle B \rangle$ ::= true | false
  | $\langle A \rangle$ `=` $\langle A \rangle$
  | $\langle A \rangle$ `<=` $\langle A \rangle$
  | $\langle A \rangle$ `>=` $\langle A \rangle$
  | $\langle B \rangle$ `and` $\langle B \rangle$
  | $\langle B \rangle$ `or` $\langle B \rangle$
  | `not` $\langle B \rangle$

Note que simplificamos a linguagem, não permitindo o uso de variáveis booleanas e de comparação de igualdade de expressões booleanas (não são válidos os trechos de programa "$x :=$ **false**" ou "`not` (**true** = **false**)").

Após apresentar os conceitos básicos de semântica denotacional, operacional e axiomática, aumentaremos nossa linguagem com procedimentos, passagem de parâmetros, não-determinismo e outras características.

Exercícios

Ex. 7 — Expanda a linguagem dada com procedimentos, dando sua sintaxe concreta e abstrata.

Ex. 8 — Escreva alguns algoritmos simples na nossa linguagem-exemplo:

i) um programa que tome o valor de uma variável n, calcule seu fatorial e o deixe na variável f.

ii) um programa que determine se um número n é primo.

iii) um programa que verifique se os valores das variáveis a, b e c poderiam ser os comprimentos dos lados de um triangulo.

Ex. 9 — Modifique a linguagem, e de suas sintaxes concreta e abstrata, adicionando:

a) procedimentos e funções;

b) declarações de variáveis;

c) declarações de tipos compostos de dados.

Ex. 10 — Implemente um interpretador ou compilador para a linguagem que descrevemos.

Capítulo 3

Semântica Denotacional

Queinnec dá uma breve introdução à semântica denotacional [Que96]; o livro de Lloyd Allison trata exclusivamente deste tópico [All86]. Os livros de Gordon [Gor79] e de Stoy [Sto81], mais antigos, trazem importantes *insights*.

Uma abordagem mais extensa em Teoria de Domínios é dada no livro de Stoltenberg-Hansen e Griffor [SLG94]. O Matemático interessado em Domínios e Categorias de funções parciais poderá consultar também o livro de Fiore [Fio96].

Na Semântica Denotacional de nossa linguagem, olhamos para trechos de programa como *funções que modificam o estado:* cada trecho c, portanto, *denota* uma função $\mathcal{C} : \Sigma \to \Sigma$. Isso apenas para comandos, claro – expressões não modificam estado. A notação que usamos é "$\mathcal{C}[\![c]\!]\sigma = \sigma'$", significando que "o programa (ou trecho) c denota a função que mapeia σ em σ'".

Escrevemos $\mathcal{C}[\![c]\!]\sigma$ ao invés de $\mathcal{C}(c, \sigma)$. Isto porque queremos indicar claramente que o argumento "c" *não* deve ser interpretado: ele é uma palavra de um domínio sintático. Ou seja, $\mathcal{C}[\![010]\!]$ tem como argumento a *palavra* 010, e não o *número* dez. Os símbolos $[\![\ ]\!]$ são demarcadores de *sintaxe*. E como visto no primeiro Capítulo, "010" pode ter significados diferentes, dependendo da linguagem (em C, significa oito; em outras linguagens, usualmente significa dez.

As expressões *usam* o estado do programa (porque dependem dos valores das variáveis), mas não o modificam. Assim, "$\mathcal{E}[\![e]\!]\sigma = n$" significa que a função semântica $\mathcal{E}$ dá o significado semântico da expressão inteira e – que por sua vez, leva do estado σ a um valor inteiro x (o valor da expressão).

3.1 Domínios Sintáticos e Semânticos

Nossa linguagem é bastante simples, com apenas três domínios sintáticos. Como funções semânticas mapeiam sintaxe em significado, precisamos definir um *domínio semântico* para cada domínio sintático.

Os domínios sintáticos **Aexp** e **Bexp** claramente devem ser mapeados em $\mathbb{Z}$ e $\{\mathbf{true}, \mathbf{false}\}$.

Um comando, ou trecho de programa, denota uma função de estados em estados. A função semântica $\mathcal{C}$, portanto, mapeia a linguagem **Cmd** em funções; estas funções transformam estados em estados. portanto a $\mathcal{C}$ deve ser do tipo $\mathcal{C} : \mathbf{Cmd} \to (\Sigma \to \Sigma)$.

Nossa linguagem tem cinco domínios sintáticos: **Cmd**, **Aexp**, **Bexp**, **Id**, **Int**. Definimos agora os domínios semânticos,

$$\begin{aligned} \mathbb{Z} &= \text{números inteiros} \\ \mathbb{B} &= \{\mathbf{true}, \mathbf{false}\} \\ (\Sigma \to \Sigma) &= \text{funções transformando estados,} \end{aligned}$$

e as funções semânticas que construiremos no resto deste Capítulo:

$$\mathcal{E} : \mathbf{Aexp} \to (\Sigma \to \mathbb{Z})$$
$$\mathcal{B} : \mathbf{Bexp} \to (\Sigma \to \mathbb{B})$$
$$\mathcal{C} : \mathbf{Cmd} \to (\Sigma \to \Sigma)$$

Definiremos as funções semânticas em casos. Por exemplo, a função $\mathcal{E} : \mathbf{Aexp} \to \mathbb{Z}$ é definida por um conjunto de equações, dentre elas

$$\mathcal{E}[\![e_1 + e_2]\!]\sigma = \mathcal{E}[\![e_1]\!]\sigma + \mathcal{E}[\![e_2]\!]\sigma$$
$$\mathcal{E}[\![e_1 - e_2]\!]\sigma = \mathcal{E}[\![e_1]\!]\sigma - \mathcal{E}[\![e_2]\!]\sigma$$

A primeira equação, por exemplo, define o caso em que $\mathcal{E}$ é aplicada com o primeiro argumento igual à soma de duas expressões: o significado de $e_1 + e_2$, no estado σ é a soma, usando o estado σ, do significado de e_1 com o significado de e_2.

Há alguns pontos importantes a observar. Primeiro, cada frase da linguagem é representada em uma equação. Além disso, o significado de uma frase (por exemplo, "$e_1 + e_2$") depende somente do significado de suas subfrases (e_1 e e_2). Aparentemente, nestes exemplos, "movemos o operador para fora da função semântica", e o lado direito das equações parece espelhar a sintaxe das frases. Isto nos leva à definição de *equações dirigidas por sintaxe*.

Definição 3.1 (função semântica; equações dirigidas por sintaxe)**.** Uma função que mapeia domínios sintáticos em domínios semânticos é chamada de *função semântica*. Um conjunto de equações definindo uma função semântica é *dirigido por sintaxe* se:

i) há exatamente uma equação para cada produção da gramática, e

ii) cada equação deve dar o significado de uma frase usando *apenas* o significado de suas subfrases. ♦

As restrições impostas por funções semânticas dirigidas por sintaxe junto com as restrições de gramáticas abstratas garantem que a função semântica sendo definida é única.

3.2 Expressões

Começamos com o significado semântico de expressões inteiras e booleanas.

A função semântica $\mathcal{E}$ dá a denotação de expressões inteiras:

$$\mathcal{E} : \mathbf{Aexp} \to (\Sigma \to \mathbb{Z})$$

O significado de números inteiros na base dez é determinado usando o fato de que o sistema que usamos é posicional, e um número com dígitos $d_n d_{n-1} \ldots d_1 d_0$ tem valor $\sum_i 10^i d_i$. Para todo estado σ e número n, $\mathcal{E}[\![n]\!]$ é constante:

$$\mathcal{E}[\![0]\!] = \mathit{0}$$
$$\mathcal{E}[\![1]\!] = \mathit{1} \qquad \mathcal{E}[\![6]\!] = \mathit{6}$$
$$\mathcal{E}[\![2]\!] = \mathit{2} \qquad \mathcal{E}[\![7]\!] = \mathit{7}$$
$$\mathcal{E}[\![3]\!] = \mathit{3} \qquad \mathcal{E}[\![8]\!] = \mathit{8}$$
$$\mathcal{E}[\![4]\!] = \mathit{4} \qquad \mathcal{E}[\![9]\!] = \mathit{9}$$
$$\mathcal{E}[\![5]\!] = \mathit{5} \qquad \mathcal{E}[\![\delta v]\!] = v + 10\mathcal{E}[\![\delta]\!]$$

Verificamos o significado de 241:

$$\begin{aligned}\mathcal{E}[\![241]\!] &= 1 + 10\mathcal{E}[\![24]\!] \\ &= 1 + 10(4 + 10\mathcal{E}[\![2]\!] \\ &= 1 + 10(4 + 10(2)) \\ &= 241.\end{aligned}$$

O significado de expressões aritméticas é dado a seguir.

$$\begin{aligned}\mathcal{E}[\![n]\!]\sigma &= n && (n \text{ é inteiro}) \\ \mathcal{E}[\![v]\!]\sigma &= \sigma(v) && (v \text{ é variável}) \\ \mathcal{E}[\![-e]\!]\sigma &= -\mathcal{E}[\![e]\!]\sigma \\ \mathcal{E}[\![e_1 + e_2]\!]\sigma &= \mathcal{E}[\![e_1]\!]\sigma + \mathcal{E}[\![e_2]\!]\sigma \\ \mathcal{E}[\![e_1 - e_2]\!]\sigma &= \mathcal{E}[\![e_1]\!]\sigma - \mathcal{E}[\![e_2]\!]\sigma \\ \mathcal{E}[\![e_1 * e_2]\!]\sigma &= \mathcal{E}[\![e_1]\!]\sigma * \mathcal{E}[\![e_2]\!]\sigma\end{aligned}$$

Da mesma forma que fizemos com expressões inteiras, construímos a semântica de expressões booleanas. A função semântica $\mathcal{B}$ é

$$\mathcal{B} : \mathbf{Bexp} \to (\Sigma \to \mathbb{B}).$$

Para expressões booleanas, temos dois valores, **true** e **false**,

$$\begin{aligned}\mathcal{B}[\![\mathbf{true}]\!]\sigma &= \mathbf{true} \\ \mathcal{B}[\![\mathbf{false}]\!]\sigma &= \mathbf{false}\end{aligned}$$

regras recursivas para operadores lógicos,

$$\begin{aligned}\mathcal{B}[\![\neg b]\!]\sigma &= \neg\mathcal{B}[\![b]\!]\sigma \\ \mathcal{B}[\![b_1 \vee b_2]\!]\sigma &= \mathcal{B}[\![b_1]\!]\sigma \text{ ou } \mathcal{B}[\![b_2]\!]\sigma \\ \mathcal{B}[\![b_1 \wedge b_2]\!]\sigma &= \mathcal{B}[\![b_1]\!]\sigma \text{ e } \mathcal{B}[\![b_2]\!]\sigma\end{aligned}$$

e para comparações entre expressões aritméticas:

$$\begin{aligned}\mathcal{B}[\![e_1 \geq e_2]\!]\sigma &= \mathbf{if}\ (\mathcal{E}[\![e_1]\!]\sigma \geq \mathcal{E}[\![e_2]\!]\sigma)\ \mathbf{then\ true\ else\ false} \\ \mathcal{B}[\![e_1 \leq e_2]\!]\sigma &= \mathbf{if}\ (\mathcal{E}[\![e_1]\!]\sigma \leq \mathcal{E}[\![e_2]\!]\sigma)\ \mathbf{then\ true\ else\ false} \\ \mathcal{B}[\![e_1 = e_2]\!]\sigma &= \mathbf{if}\ (\mathcal{E}[\![e_1]\!]\sigma = \mathcal{E}[\![e_2]\!]\sigma)\ \mathbf{then\ true\ else\ false}\end{aligned}$$

Não incluímos parênteses porque trabalhamos com sintaxe abstrata.

3.2.1 Números em bases diferentes

Podemos também definir o significado de números em bases diferentes, uma vez que a representação em diferentes bases é sempre feita usando o sistema posicional; variam somente a quantidade de casos e o valor a ser multiplicado na regra indutiva.

Exemplo 3.2. Para números em hexadecimal, definimos a função $\mathcal{X}$. Seja $H = \{0, 1, 2, \ldots, 9, A, B, C, D, E, F\}$. Então

$$\begin{aligned}
\mathcal{X}[\![0]\!] &= 0 \\
\mathcal{X}[\![1]\!] &= 1 \\
\mathcal{X}[\![2]\!] &= 2 \\
\mathcal{X}[\![3]\!] &= 3 \\
\mathcal{X}[\![4]\!] &= 4 \\
\mathcal{X}[\![5]\!] &= 5 \\
\mathcal{X}[\![6]\!] &= 6 \\
\mathcal{X}[\![7]\!] &= 7
\end{aligned}
\qquad
\begin{aligned}
\mathcal{X}[\![8]\!] &= 8 \\
\mathcal{X}[\![9]\!] &= 9 \\
\mathcal{X}[\![A]\!] &= 10 \\
\mathcal{X}[\![B]\!] &= 11 \\
\mathcal{X}[\![C]\!] &= 12 \\
\mathcal{X}[\![D]\!] &= 13 \\
\mathcal{X}[\![E]\!] &= 14 \\
\mathcal{X}[\![F]\!] &= 15 \\
\mathcal{X}[\![\delta v]\!] &= v + 16\mathcal{X}[\![\delta]\!]
\end{aligned}$$

onde $v \in H$, e $\delta \in H^+$. ◀

3.3 Comandos

Construiremos a função semântica $\mathcal{C}$, um comando por vez.

- **skip:** o comando não modifica o estado, logo $\mathcal{C}[\![\textbf{skip}]\!]$ é a função identidade: $\mathcal{C}[\![\textbf{skip}]\!]\ (\sigma) = \sigma$, ou, sem os parênteses,
$$\mathcal{C}[\![\textbf{skip}]\!]\sigma = \sigma$$
- **atribuição:** uma atribuição somente modifica o valor de uma variável; assim, sua semântica é
$$\mathcal{C}[\![x := t]\!]\sigma = \sigma[\mathcal{E}[\![t]\!]\sigma \ / \ x]$$
- **sequência:** executamos dois comandos em sequência, $c_1; c_2$. Se c_1 não para, a função semântica da composição deve resultar em $\perp$, independente de quem seja c_2. Caso contrário, deve ser a função composta $c_1 \circ c_2$.
$$\mathcal{C}[\![c_1; c_2]\!]\sigma = \textbf{if } (\mathcal{C}[\![c_1]\!]\sigma = \perp) \textbf{ then } \perp \textbf{ else } \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)$$
- **if:** a semântica do comando condicional é simples. Dependendo do valor de $\mathcal{B}[\![b]\!]$, c_1 ou c_2 será executado, e o valor da semântica do **if** será, portanto, $\mathcal{C}[\![c_1]\!]$ ou $\mathcal{C}[\![c_2]\!]$.
$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]\sigma = \textbf{if } (\mathcal{B}[\![b]\!]\sigma = \textbf{true}) \textbf{ then } \mathcal{C}[\![c_1]\!]\sigma \textbf{ else } \mathcal{C}[\![c_2]\!]\sigma$$

A função $\mathcal{C}$ é resumida a seguir, exceto por não termos incluído a semântica do comando **while**. Aqui, "c" e "c_i" são comandos; "b" é expressão booleana; e "e" é expressão inteira.

$$\begin{aligned}
\mathcal{C}[\![\textbf{skip}]\!]\sigma &= \sigma \\
\mathcal{C}[\![v := e]\!]\sigma &= \sigma[\mathcal{E}[\![e]\!]\sigma \ / \ v] \\
\mathcal{C}[\![c_1; c_2]\!]\sigma &= \textbf{if } (\mathcal{C}[\![c_1]\!]\sigma = \perp) \textbf{ then } \perp \textbf{ else } \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma) \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]\sigma &= \textbf{if } (\mathcal{B}[\![b]\!]\sigma = \textbf{true}) \textbf{ then } \mathcal{C}[\![c_1]\!]\sigma \textbf{ else } \mathcal{C}[\![c_2]\!]\sigma
\end{aligned}$$

Agora verificamos o problema que encontramos com a semântica de **while**. Tentamos inicialmente a seguinte definição.

$$\begin{aligned}
\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \textbf{if } &(\neg\mathcal{B}[\![b]\!]\sigma) \textbf{ then } \sigma \\
&\textbf{if } (\mathcal{C}[\![c]\!]\sigma = \perp) \textbf{ then } \perp \\
&\textbf{else } \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!](\mathcal{C}[\![c]\!]\sigma)
\end{aligned}$$

O problema com esta tentativa é que a semântica de **while** está sendo definida em termos de si mesma (há um **while** no lado direito da equação).

Como exemplo concreto de um problema que surge desta definição de semântica, observamos o que ocorre quando tentamos determinar o significado semântico de "**while true do skip**".

$$\begin{aligned}\mathcal{C}[\![\textbf{while true do skip}]\!]\sigma &= \mathcal{C}[\![\textbf{while true do skip}]\!](\mathcal{C}[\![\textbf{skip}]\!]\sigma)\\ &= \mathcal{C}[\![\textbf{while true do skip}]\!]\sigma\end{aligned}$$

porque $\mathcal{C}[\![\textbf{skip}]\!]\sigma = \sigma$.

Na verdade, deveríamos chegar a

$$\mathcal{C}[\![\textbf{while true do skip}]\!]\sigma = \bot,$$

para qualquer σ. Não conseguimos porque nossa definição é circular, e ao tentar usá-la, nossa dedução também voltou ao início.

3.4 Definições Recursivas e seus Pontos Fixos

Encontramos um problema para descrever a semântica do comando **while** porque sua definição é naturalmente recursiva: repetições em número parametrizável de vezes são descritas dessa forma[1].

Definição 3.3 (solução para definição recursiva)**.** Uma *solução* para uma definição recursiva de x é um valor que, quando substituído em x, satisfaça a equação da definição.

Ou seja, se a definição é $x \stackrel{\text{def}}{=} \varphi(x)$, então um elemento s é solução se e somente se $s = \varphi(s)$. ♦

Exemplo 3.4. Damos alguns exemplos de definições recursivas de valores numéricos.

- $x = 1 + x$ não tem soluções em $\mathbb{C}$.
- $x = \frac{2}{x}$ tem uma única solução em $\mathbb{C}$: $x = \sqrt{2}$
- $x = 2x$ tem uma única solução sobre $\mathbb{C}$: $x = 0$.
- $x = \frac{4}{x}$ tem duas soluções em $\mathbb{C}$: $x = \pm 2$. ◄

Exemplo 3.5. A quantidade de soluções para uma definição recursiva depende do conjunto universo que usamos.

- $x = \frac{1}{x^3}$ tem
  - uma solução em $\mathbb{N}$: $x = 1$.
  - duas soluções em $\mathbb{Z}$ e $\mathbb{R}$: $x = \pm 1$.
  - quatro soluções em $\mathbb{C}$: $x = \pm 1$, $x = \pm i$.
- $x = \frac{2}{(2x)^3}$ tem
  - nenhuma solução em $\mathbb{N}$ ou $\mathbb{Z}$.
  - duas soluções em $\mathbb{R}$: $x = \pm\frac{1}{\sqrt{2}}$.
  - quatro soluções em $\mathbb{C}$: $x = \pm\frac{1}{\sqrt{2}}$, $x = \pm\frac{i}{\sqrt{2}}$.
- $x = x$ tem tantas soluções quantos elementos houver no conjunto universo. Em $\mathbb{R}$ e $\mathbb{C}$, a definição tem uma quantidade não enumerável de soluções (mas os conjuntos de soluções são diferentes para $\mathbb{R}$ e $\mathbb{C}$). Em $\mathbb{N}$ e $\mathbb{Z}$, tem uma quantidade enumerável de soluções (mas os conjuntos de soluções em $\mathbb{N}$ e $\mathbb{Z}$ são diferentes). Em $\{1, 2, 3\}$, tem três soluções. No conjunto vazio, não tem soluções. ◄

[1] O leitor familiar com Teoria das Funções Recursivas e Computabilidade reconhecerá o operador de ponto fixo, que construiremos nesta seção.

Exemplo 3.6. Retomamos a definição recursiva de sequência dada no exemplo 1.33,

$$a_1 = 2$$
$$a_n = 2a_{n-1} + 2.$$

Esta definição tem uma única solução – a função

$$a_n = 2(2^n - 1),$$

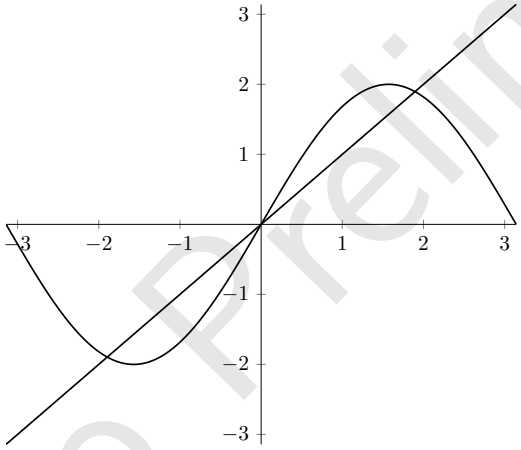
ou, usando explicitamente a notação de função, nossa solução é $\lambda n \cdot 2(2^n - 1)$. ◀

Definição 3.7 (ponto fixo)**.** Um *ponto fixo* de uma função f é um elemento x do domínio de f tal que $f(x) = x$. ♦

Exemplo 3.8. Os números 0 e 1 são os (únicos) pontos fixos da função raiz quadrada em $\mathbb{R}^+$, porque $\sqrt{0} = 0$ e $\sqrt{1} = 1$. ◀

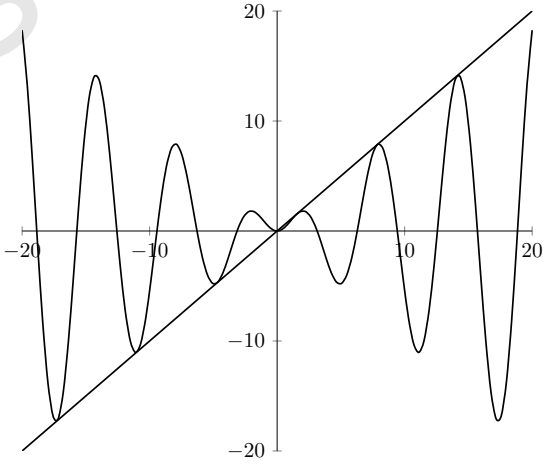
Exemplo 3.9. O zero é o único ponto fixo da função seno, porque $\operatorname{sen}(0) = 0$. ◀

Exemplo 3.10. A função $2\operatorname{sen}(x)$ tem três pontos fixos: 0 e $\pm 1.89549\ldots$. Um ponto fixo da função acontece na interseção dela com a função identidade $f(x) = x$, como vemos no gráfico a seguir.



◀

Exemplo 3.11. A função $x\operatorname{sen}(x)$ tem infinitos pontos fixos: $x = 0$ e $x = \frac{4\pi n + \pi}{2}, n \in \mathbb{Z}$. O gráfico a seguir mostra as funções $f(x) = x$ e $g(x) = x\operatorname{sen}(x)$.



As duas coincidirão nos pontos fixos de $x\,\mathrm{sen}(x)$, Como $\mathrm{sen}(x) \in [-1,1]$, claramente teremos $x = x\,\mathrm{sen}(x)$ exatamente quando $x = 0$, e quando $\mathrm{sen}(x) = +1$. Observe, no entanto, que quando $x < 0$, nos pontos em que $\mathrm{sen}(x) = +1$, temos $x\,\mathrm{sen}(x) < 0$, por isso as interseções no lado esquerdo do gráfico são abaixo do zero. ◀

Definição 3.12 (função geradora de definição recursiva)**.** Associada a toda definição recursiva

$$x \stackrel{\text{def}}{=} \cdots\ x\ \cdots$$

definimos uma *função geradora*

$$\lambda x \cdot \ldots\ x\ \ldots$$ ♦

Fazemos agora uma observação que nos permitirá determinar a semântica de laços **while** e diversas outras construções de repetição: *a solução para uma definição recursiva é o ponto fixo de sua função geradora.*

Exemplo 3.13. Considere a definição recursiva

$$x \stackrel{\text{def}}{=} \frac{1}{x}.$$

A função geradora desta definição é

$$\lambda x \cdot 1/x.$$

Em $\mathbb{R}$, há dois pontos fixos para esta função, ± 1, que são as duas soluções da definição. ◀

O próximo exemplo é de grande importância, porque ilustra uma situação em que a função geradora é um funcional (é to tipo $(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$). Isto acontecerá também com a semântica do **while**, exceto que ao invés de $\mathbb{N}$, usaremos o conjunto de estados do programa.

Exemplo 3.14. No exemplo 1.33 mostramos uma sequência definida recursivamente. Em notação λ, a sequência é

$$\lambda n \cdot \begin{cases} 2 & n = 1 \\ 2(a\ \ n-1) + 2 & n > 1 \end{cases}$$

Como o objeto que estamos definindo é uma função do tipo $\mathbb{N} \to \mathbb{N}$, *a função geradora é uma função do tipo* $(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$.

$$\underbrace{\lambda g}_{\text{dada } g,} \cdot \underbrace{\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 2(g\ \ \ n-1) + 2 \text{ se } n > 1\Big)}_{\text{calculamos a função } \lambda n \cdot \ldots g \ldots}$$

Sabemos que a solução para uma definição recursiva é o ponto fixo de sua função geradora. Verificaremos o que obtemos da função geradora para três funções diferentes, candidatas a ponto fixo da função geradora.

- Se $f(n) = n$, a função geradora nos dará

$$\begin{aligned} &\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 2(f\ \ \ n-1) + 2 \text{ se } n > 1\Big) \\ =&\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 2n - 2 + 2 \text{ se } n > 1\Big) \\ =&\lambda n \cdot 2n, \end{aligned}$$

  que não é ponto fixo da função geradora, porque $f(n)$ não é igual a $\lambda n \cdot 2n$.

- Se $g(n) = 2n$,

$$\begin{aligned} &\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 2(g\ \ \ n-1) + 2 \text{ se } n > 1\Big) \\ =&\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 2(2n-2) + 2 \text{ se } n > 1\Big) \\ =&\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 4n - 2 \text{ se } n > 1\Big) \\ =&\lambda n \cdot 4n - 2, \end{aligned}$$

  que também não é ponto fixo, porque $g(n) \neq \lambda n \cdot 4n - 2$.

- Se $h(n) = 2(2^n - 1)$, a função geradora nos dá

$$\begin{aligned}
&\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 2(h\quad n-1) + 2 \text{ se } n > 1\Big)\\
=&\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 2[2(\underline{2^{n-1} - 1})] + 2 \text{ se } n > 1\Big)\\
=&\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 2[2^n - 2] + 2 \text{ se } n > 1\Big)\\
=&\lambda n \cdot \Big(2 \text{ se } n = 1;\ \ 2(2^n - 1) \text{ se } n > 1\Big)\\
=&\lambda n \cdot 2(2^n - 1),
\end{aligned}$$

  que de fato é o único ponto fixo!

O ponto fixo da função geradora é exatamente a solução da recorrência, que já vimos no exemplo 3.6. ◀

3.5 Ordens Parciais e Domínios

Quando uma definição recursiva define elementos em um conjunto parcialmente ordenado, passa a fazer sentido comparar os pontos fixos da função geradora. Esta seção trata da definição e da prova da existência de *pontos fixos mínimos*.

Definição 3.15 (conjunto parcialmente ordenado)**.** Uma relação $\sqsubseteq$ em um conjunto X é uma *ordem parcial* se é

- reflexiva: $x \sqsubseteq x$.
- transitiva: $x \sqsubseteq y$, $y \sqsubseteq z$ implica em $x \sqsubseteq z$.
- antissimétrica: $x \sqsubseteq y$, $y \sqsubseteq x$ implica em $x = y$

Dizemos que o par $(X, \sqsubseteq)$ é um *conjunto parcialmente ordenado*. ♦

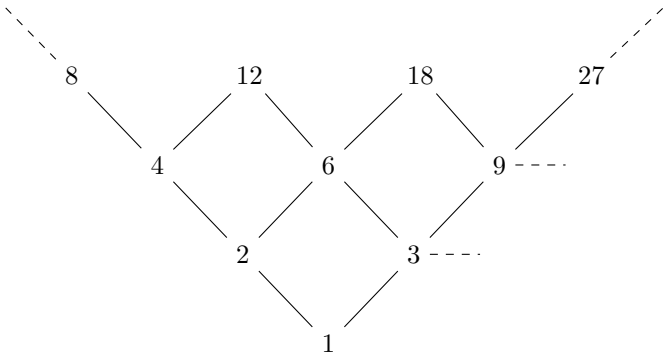
Abusaremos da nomenclatura, e diremos que o conjunto parcialmente ordenado $(X, \sqsubseteq)$ é uma *ordem parcial*.

Exemplo 3.16. Um exemplo simples e evidente de ordem parcial é $(\mathbb{Z}, \leq)$. ◀

Exemplo 3.17. Seja X um conjunto. Então $(X, =)$ é ordem parcial, porque (i) $\forall x \in X, x = x$; (ii) se $x = y$ e $y = z$ então $x = z$. (iii) se $x = y$ e $y = x$, trivialmente temos $x = y$. ◀

Definição 3.18 (diagrama de Hasse)**.** O *diagrama de Hasse* de uma ordem parcial é um grafo onde os vértices são os elementos do conjunto parcialmente ordenado, e as arestas representam a relação de ordem. O diagrama de Hasse é desenhado de forma que se $a \sqsubseteq b$, então a estará abaixo de b. Só são representadas as relações diretas de $\sqsubseteq$ – se $a \sqsubseteq c$ pode ser inferido por transitividade, a aresta (a, c) não é desenhada. ♦

Exemplo 3.19. $(\mathbb{N}^*, |)$, onde $a|b$ significa "a divide b" é conjunto parcialmente ordenado. A figura a seguir mostra apenas parte do diagrama de Hasse desse conjunto; fica claro que o diagrama é infinito, e cresce indefinidamente à direita e para cima.



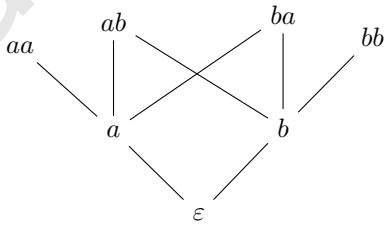
Na base da figura está o número um, que divide qualquer natural; imediatamente acima dele, estão os primos; a seguir os números com apenas dois fatores, e assim por diante (a quantidade de fatores primos do número é a altura dele no diagrama). ◀

Exemplo 3.20. Seja Σ um alfabeto, e Σ^* o conjunto de palavras com os símbolos em Σ. Então $(\Sigma^*, \prec)$, onde $\prec$ significa "é sub-palavra", é ordem parcial, porque (i) toda palavra é sub-palavra de si mesma; (ii) se a é sub-palavra de b, e b é sub-palavra de c, claramente a é sub-palavra de c; (iii) se a é sub-palavra de b e b é sub-palavra de a necessariamente $a = b$.

Para um exemplo concreto usamos o alfabeto $\Sigma = \{a, b\}$. O conjunto de palavras sobre Σ é

$$\begin{aligned}\Sigma^* = \{&\varepsilon, a, b, aa, ab, ba, bb\\ &aaa, aab, aba, baa,\\ &abb, bab, bba, bbb, \ldots\}.\end{aligned}$$

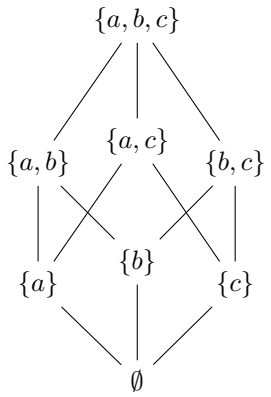
Parte do diagrama de Hasse é mostrada a seguir.



◀

Exemplo 3.21. Seja X um conjunto qualquer. Então $\mathcal{P}(X), \subseteq$ é ordem parcial: (i) todo conjunto é subconjunto de si mesmo; (ii) a relação $\subseteq$ é transitiva; (iii) a relação $\subseteq$ é antissimétrica: se $A \subseteq B$ e $B \subseteq A$, então $A = B$.

Para $X = \{a, b, c\}$, o diagrama de Hasse completo de $(\mathcal{P}(X), \subseteq)$ é mostrado na figura a seguir.



◀

Exemplo 3.22. Considere o conjunto $\mathbb{N} \to \mathbb{R}$ de funções parciais de naturais em reais. Uma possível ordem parcial para este conjunto é

$$a \sqsubseteq b \;\Leftrightarrow\; \text{gr}(a) \subseteq \text{gr}(b).$$

Damos exemplos de gráficos de funções neste conjunto. Se $a_0 = 0, a_n = 2n$, então $\text{gr}(a) = \{(0,0),(1,2),(2,4),\ldots\}$. Se $c_n = 1$, então $\text{gr}(c) = \{(0,1),(1,1),(2,1),\ldots\}$. Como admitimos funções parciais, temos também a função parcial x definida somente em dois pontos, com $\text{gr}(x) = \{(0,0),(1,5)\}$.

Se denotarmos por a^k a função parcial igual a a, mas com o domínio restrito a naturais $\leq k$, temos

- $\text{gr}(a^0) = \{\}$ (a^0 não é definida para nenhum valor)
- $\text{gr}(a^1) = \{(0,0)\}$ (a^1 é definida somente para o zero)
- $\text{gr}(a^2) = \{(0,0),(1,2)\}$ (a^2 é definida para $1, 2$)
- $\text{gr}(a^n) = \{(0,0),(1,2),\ldots,(n,2n)\}$ (a^n é definida para $k \leq n$)

Mas como

$$\begin{aligned}\{\} &\subseteq \{(0,0)\}\\ &\subseteq \{(0,0),(1,2)\}\\ &\subseteq \{(0,0),(1,2),(2,4)\}\\ &\;\;\vdots\\ &\subseteq \{(0,0),(1,2),(2,4),\ldots,(n,2n)\}\\ &\;\;\vdots,\end{aligned}$$

então em nossa ordem parcial

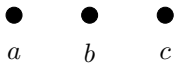
$$a^0 \sqsubseteq a^1 \sqsubseteq a^2 \sqsubseteq \cdots$$

Mais ainda, a^0 é idêntica a f^0 para toda f, e $a^0 = b^0 = \cdots$ precede todas as funções nesta ordem parcial, porque $\emptyset \subseteq X$ para todo conjunto X. ◀

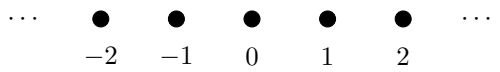
Exemplo 3.23. Seja $\mathcal{G}$ o conjunto de todos os grafos. Então a relação "subgrafo de" induz uma ordem parcial em $\mathcal{G}$, porque (i) todo grafo é subgrafo dele mesmo; (ii) a relação "subgrafo de" é claramente transitiva; (iii) se A é subgrafo de B e B também é subgrafo de A, necessariamente $A = B$. ◀

Definição 3.24 (ordem parcial discreta)**.** Uma ordem parcial $(X, \sqsubseteq)$ é *discreta* se seus elementos são, dois-a-dois, incomparáveis, ou seja, não existem $x, y \in X$, com $x \neq y$ tais que $x \sqsubseteq y$. ♦

Exemplo 3.25. Qualquer conjunto com a relação de igualdade é uma ordem parcial discreta. Um exemplo é $(\{a, b, c\}, =)$, cujo diagrama de Hasse mostramos a seguir.



Outro exemplo é $(\mathbb{Z}, =)$.



Os diagramas de Hasse de ordens parciais discretas não tem arestas exceto pelos loops (que não são mostrados). ◀

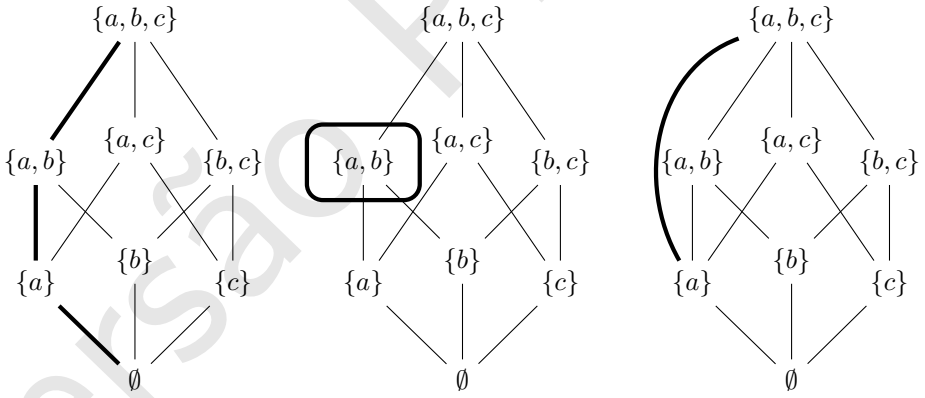
Uma ω-cadeia é, informalmente, um caminho no diagrama de Hasse de uma ordem parcial (incluindo arestas obtidas por transitividade).

Definição 3.26 (ω-cadeia)**.** Seja $(X, \sqsubseteq)$ uma ordem parcial. Uma *ω-cadeia* ou *cadeia* em X é uma sequência não vazia e crescente de elementos de X, $x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq$. ♦

Exemplo 3.27. Seja $S = \{a, b, c\}$. Na ordem parcial $(\mathcal{P}(S), \subseteq)$ há diversas ω-cadeias, dentre elas

- $\emptyset, \{a\}\{a, c\}, \{a, b, c\}$
- $\{a, b\}$
- $\{a\}, \{a, b, c\}$

As ω-cadeias são caminhos de baixo para cima no diagrama de Hasse (na verdade no fecho transitivo dele: os caminhos podem seguir por arestas não mostradas, como $\{a\} \to \{a, b, c\}$).



Na primeira figura, o caminho destacado mostra a ω-cadeia $\emptyset, \{a\}\{a, c\}, \{a, b, c\}$; na segunda, a ω-cadeia $\{a, b\}$; e na terceira, a ω-cadeia $\{a\}, \{a, b, c\}$. ◀

Exemplo 3.28. O conjunto $\mathbb{N}$ é ele mesmo uma ω-cadeia em $(\mathbb{N}, \leq)$. ◀

Exemplo 3.29. Os naturais pares são uma ω-cadeia em $(\mathbb{N}, \leq)$. E também em $(\mathbb{Z}, \leq)$, $(\mathbb{Q}, \leq)$, e em $(\mathbb{R}, \leq)$. ◀

Exemplo 3.30. Damos agora duas relações de ordem parcial para $\mathbb{C}$.

Primeiro definimos $\prec$, que ordena os complexos por norma (e portanto é uma ordem parcial).

Podemos também ordenar os complexos de outra forma: $a + bi \sqsubseteq c + di$ se e somente se $a \leq c$ e $b \leq d$.

Os naturais pares são uma ω-cadeia em $(\mathbb{C}, \prec)$, e também em $(\mathbb{C}, \sqsubseteq)$. ◀

Exemplo 3.31. $(1, 2, 3)$ é uma ω-cadeia em $(\mathbb{N}, \leq)$. ◀

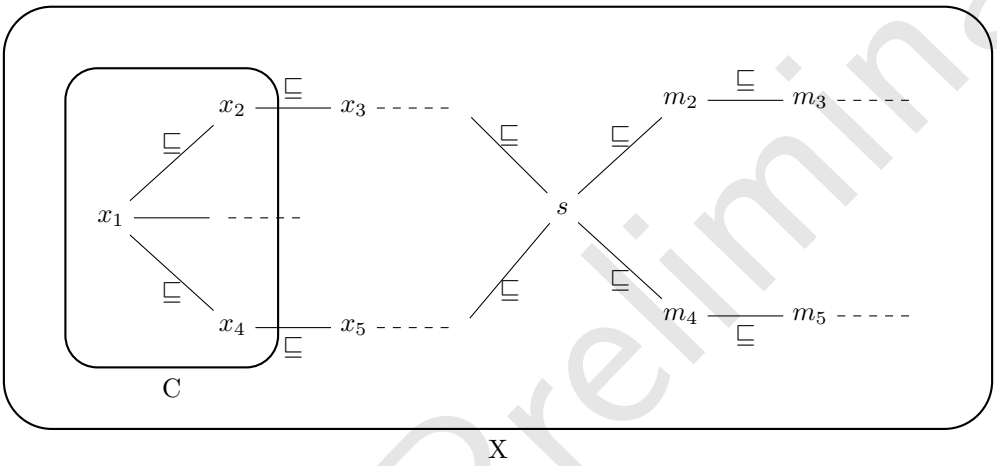
Exemplo 3.32. Uma sequência constante é ω-cadeia em qualquer ordem parcial: por exemplo, em $(\mathbb{Z}, \leq)$, a função $f(x) = 5$ nos dá a sequência $5, 5, 5, \ldots$, que é crescente usando $\leq$. ◀

Definição 3.33 (majorante, supremo)**.** Seja $(X, \sqsubseteq)$ um conjunto parcialmente ordenado, e $S \subseteq X$. Um elemento $x \in X$ é um *majorante*, ou *limitante superior*, de S se para todo $s \in S$, $s \sqsubseteq x$.

Usamos a mesma definição e notação quando S é ω-cadeia, porque toda ω-cadeia é, ela mesma, um conjunto parcialmente ordenado.

O menor dos majorantes de S é o *supremo* de $(S, \subseteq)$. Denotamos o supremo de S por $\sup S$, ou $\sqcup S$. ♦

A figura a seguir ilustra os conceitos de majorante e supremo: s, m_2, m_3, m_4, m_5 são majorantes de $C \subseteq X$. Como s é o menor dos majorantes, então s é supremo de C (escrevemos[2] $s = \sqcup S$).



Observe que x_3 e x_5 *não* são majorantes de C, porque nenhum deles é maior que todos os elementos de C. O majorante em uma ordem parcial não precisa estar "imediatamente acima" do conjunto.

Exemplo 3.34. Em $\mathbb{N}$, a ω-cadeia constante $5, 5, \ldots$ tem como majorantes $5, 6, 7, \ldots$, mas somente 5 é o supremo. ◀

Exemplo 3.35. $\mathbb{N}$ não tem majorante, porque sempre há um natural maior que qualquer um que escolhamos. ◀

Definição 3.36 (pré-dominio)**.** Um *pré-domínio* é um conjunto parcialmente ordenado onde toda ω-cadeia tem supremo. ♦

Exemplo 3.37. A ordem parcial $(\mathbb{Z}, \leq)$ *não* é pré-domínio, porque $\mathbb{Z}$ é ω-cadeia em si mesmo, mas não há supremo em $\mathbb{Z}$ (para todo inteiro que se possa supor ser o supremo, há um maior que pertence a $\mathbb{Z}$). ◀

Exemplo 3.38. As ordens parciais $(\mathbb{N}_\omega, \leq)$, $(\mathbb{Z}_\omega, \leq)$, $(\mathbb{R}_\omega, \leq)$, $(\mathbb{C}_\omega, \prec)$, onde $\prec$ ordena por norma, são pré-domínios. ◀

Exemplo 3.39. Qualquer conjunto com a relação de igualdade é pré-domínio: as únicas ω-cadeias são constantes, porque a relação é $=$, e em cadeias constantes sempre há supremo. ◀

Exemplo 3.40. Considere a ordem parcial $(\mathbb{R}, \leq)$. As ω-cadeias $[0, 1]$ e $[0, 1)$ tem supremo igual a 1; no entanto, somente a primeira tem máximo (também 1). Já a ω-cadeia $[0, +\infty)$ não tem supremo. ◀

[2]Para entender o motivo desta notação, o leitor pode considerar a ordem parcial $(\mathcal{P}(X), \subseteq)$, e verificar que ali o supremo de um subconjunto é dado pela sua união: se $S \subseteq \mathcal{P}(X)$, então o supremo de S é $\cup\{x : x \in S\}$, ou, de forma mais curta, $\cup S$. O mesmo vale para conjuntos de funções ordenadas por seus gráficos (que são conjuntos!)

Exemplo 3.41. $(\mathbb{N}, |)$ é pré-domínio, porque (i) é ordem parcial; (ii) toda ω-cadeia tem supremo. Observe que (*ii*) é verdade porque incluímos o zero: o conjunto tem um menor elemento 1, que divide todos os naturais, e um *maior* elemento zero, a quem todo natural divide! Se há um maior elemento, toda ω-cadeia tem supremo. ◀

Definição 3.42 (domínio)**.** Um *domínio*, ou *domínio de Scott*, é um pré-domínio com um menor elemento. Denotamos o menor elemento de X por $\bot_X$ ou, quando não houver ambiguidade, simplesmente por $\bot$. ♦

Exemplo 3.43. A ordem parcial $(\mathbb{C}, \sqsubseteq)$ dos complexos ordenados de forma que

$$a + bi \sqsubseteq \alpha + \beta i \quad \Rightarrow \quad a < \alpha \text{ e } b < \beta,$$

não é domínio porque, mesmo sendo pré-domínio (toda ω-cadeia tem supremo), não existe menor elemento. ◀

Exemplo 3.44. A ordem parcial $(\mathbb{N}, \leq)$ não é domínio porque, mesmo havendo um menor elemento (o zero) nesta ordem parcial, $\mathbb{N}$ é uma ω-cadeia sem supremo. ◀

Definição 3.45 (elemento ω)**.** Seja X um conjunto numérico ordenado sem um maior elemento, como $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ ou $\mathbb{R}$. Denotamos por X_ω o mesmo conjunto, com um elemento adicional ω, que definimos como sendo maior que todos os outros. Por exemplo,

- $\mathbb{N}_\omega$ é $\mathbb{N} \cup \{\omega\}$, tal que $\omega > n$, para todo $n \in \mathbb{N}$;
- $\mathbb{R}_\omega$ é $\mathbb{R} \cup \{\omega\}$, tal que $\omega > x$, para todo $x \in \mathbb{R}$. ♦

Exemplo 3.46. A ordem parcial $(\mathbb{N}_\omega, \leq)$ é um domínio: (i) é uma ordem parcial; (ii) há um menor elemento 0; (iii) toda ω-cadeia tem supremo (porque quando uma ordem parcial tem máximo, toda ω-cadeia nela tem supremo). ◀

Exemplo 3.47. $(\mathbb{N}, |)$ é domínio. ◀

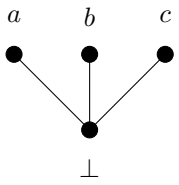
Podemos transformar qualquer pré-domínio em domínio simplesmente adicionando a ele um menor elemento. Chamamos a isso de "elevação de domínio".

Definição 3.48 (elevação de ordem parcial)**.** A *elevação* da ordem parcial $(X, \sqsubseteq)$ é $(X \cup \{\bot\}, \sqsubseteq)$, onde se define que para todo $x \in X$, $\bot \sqsubseteq x$. Denotamos a ordem parcial elevada por $X_\bot$.

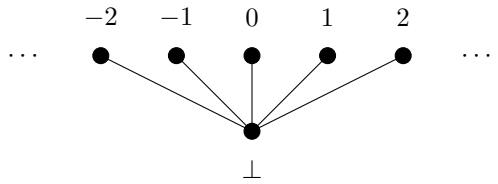
Se a ordem parcial $(X, \sqsubseteq)$ é pré-domínio, dizemos que $X_\bot$ é um *domínio elevado*. ♦

Definição 3.49. $(\mathbb{Z}_\omega, \leq)$ é pré-domímo. A elevação desse pré-domímo é $(\mathbb{Z}_{\omega,\bot}, \leq)$, onde $\bot$ é menor que qualquer inteiro. ♦

Exemplo 3.50. A elevação de ordens parciais discretas (onde a relação de ordem é $=$) resulta em domínios. As ordens parciais do exemplo 3.25, quando elevadas, dão origem a dois domínios. O primeiro é $\{a, b, c, \bot\}$, com a relação ilustrada no diagrama de Hasse a seguir:



O outro domínio é $\mathbb{Z}_\bot$, com a relação no diagrama a seguir:



◀

3.6 Continuidade e Pontos Fixos Mínimos

Damos a seguir definições de monotonicidade e continuidade para funções. Estas ideias são similares às usadas em Cálculo, Análise e Topologia.

Definição 3.51 (função monotônica)**.** Sejam $(A, \sqsubseteq_A)$ e $(B, \sqsubseteq_B)$ dois conjuntos parcialmente ordenados. Uma função $f : A \to B$ é *monotônica* se preserva ordem, ou seja, para todos $a, a' \in A$

$$a \sqsubseteq_A a' \Rightarrow f(a) \sqsubseteq_B f(a'). \qquad \blacklozenge$$

Exemplo 3.52. A função cardinalidade de conjunto, denotada $|\cdot|$, é monotônica em $(\mathcal{P}(X), \subseteq)$, porque

$$A \subseteq B \;\Rightarrow\; |A| \leq |B|. \qquad \blacktriangleleft$$

Os dois próximos exemplos mostram que a mesma função, definida sobre o mesmo conjunto, pode ou não ser monotônica, dependendo da ordem parcial que usamos.

Exemplo 3.53. A função $f(x) = x + 1$ é monotônica em $(\mathbb{N}, \leq)$. $\blacktriangleleft$

Exemplo 3.54. A função $f(x) = x + 1$ *não* é monotônica em $(\mathbb{N}, |)$: $2 \mid 4$, mas $f(2) = 3$ e $f(4) = 5$, e $3 \nmid 5$. $\blacktriangleleft$

Exemplo 3.55. A função $f(x) = x^2$ *não* é monotônica em $(\mathbb{R}, \leq)$, porque $-2 \leq 1$, mas $(-2)^2 \not\leq 1^2$. $\blacktriangleleft$

Teorema 3.56. *Funções monotônicas preservam ω-cadeias.*

Demonstração. Seja $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots$ uma ω-cadeia em um domínio $(X, \sqsubseteq_X)$, e seja $f : X \to Y$ monotônica. Trivialmente, $x_i \sqsubseteq_X x_j$ implica em $f(x_j) \sqsubseteq_Y f(x_k)$, logo a sequência $f(x_i)$ é crescente em $(Y, \sqsubseteq_Y)$. $\square$

Definição 3.57 (função contínua)**.** Sejam $(A, \sqsubseteq_A)$ e $(B, \sqsubseteq_B)$ dois conjuntos parcialmente ordenados. Uma função $f : A \to B$ é *contínua* (ou *Scott-contínua*) se *preserva supremos*, ou seja, para toda ω-cadeia c em A,

$$f(\sqcup c) = \sqcup(f \quad c). \qquad \blacklozenge$$

Exemplo 3.58. Tome a ordem parcial $(\mathbb{Z}_\omega, \leq)$. Defina $f : \mathbb{Z}_\omega \to \{a, b\}$, com $a \sqsubseteq b$:

$$f(x) = \begin{cases} a & \text{se } x \neq \omega \\ b & \text{se } x = \omega. \end{cases}$$

Considere a cadeia $\mathbb{Z}$. Sua imagem sob f induz a ordem parcial $a, a, a, \ldots$ (há uma quantidade infinita de a's na cadeia).

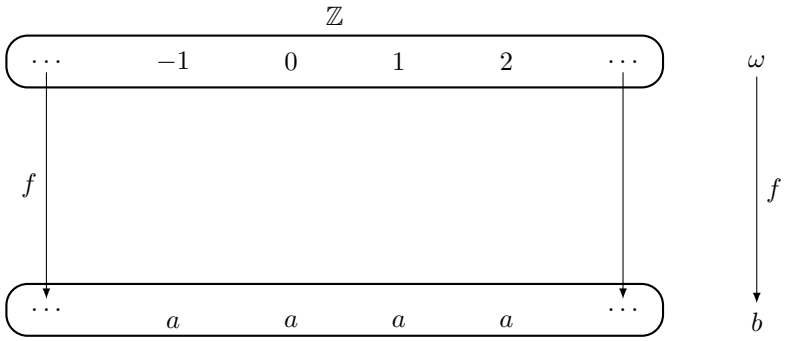
Temos

$$\sqcup\mathbb{Z} = \omega,$$

mas

$$f(\sqcup\mathbb{Z}) = f(\omega) = b$$
$$\sqcup f(\mathbb{Z}) = \sqcup(a, a, a, \ldots) = a.$$

Como $f(\sqcup\mathbb{Z}) \neq \sqcup f(\mathbb{Z})$, a função *não* é contínua. A figura a segur ilustra a situação que descrevemos.



$\sqcup f(\mathbb{Z}) = \sqcup(\ldots, a, a, \ldots) = a$

$f(\sqcup\mathbb{Z}) = b$

◀

A relação entre as noções de continuidade e monotonicidade é dada pelos Teoremas 3.59 e 3.60.

Teorema 3.59. *Funções contínuas são monótonas em qualquer ordem parcial.*

Teorema 3.60. *Em ordens parciais finitas (ou infinitas, mas contendo somente uma quantidade finita de ω-cadeias), toda função monótona é também contínua.*

Definição 3.61 (ponto fixo mínimo)**.** Seja f uma função definida em uma ordem parcial $(X, \sqsubseteq)$. O *ponto fixo mínimo* de f é o menor elemento $x \in X$ tal que $f(x) = x$. Denotamos o ponto fixo mínimo de f por $\operatorname{lfp} f$. ♦

Exemplo 3.62. Tome a ordem parcial $(\mathbb{N}^+, \leq)$, e a função $f(n) = \lceil \sqrt{n} \rceil$. A função f tem pontos fixos 1 e 2. O ponto fixo mínimo de f nessa ordem parcial é 1:

$$\operatorname{lfp} f = 1.$$ ◀

Teorema 3.63. *Pontos fixos mínimos são únicos.*

Chegamos finalmente ao Teorema do Ponto Fixo de Kleene (3.64), que nos permitirá definir a semântica do comando **while**.

Teorema 3.64 (do ponto fixo de Kleene)**.** *Seja f contínua em $(X, \sqsubseteq, \bot)$, e seja $c : \mathbb{N} \to X$ uma sequência tal que $c(n) = f^n(\bot)$. Então c é ω-cadeia, e $\operatorname{lfp}(f) = \sqcup c$.*

Demonstração. Primeiro mostramos que $\sqcup c$ é ponto fixo de f.

Como $\bot$ é o menor elemento de X, $\bot \sqsubseteq f(\bot)$.

Como f é contínua, é também monotônica. Então,

$$\bot \sqsubseteq f(\bot) \;\Rightarrow\; f(\bot) \sqsubseteq f(f(\bot)).$$

Por indução, claramente temos

$$f^n(\bot) \sqsubseteq f^{n+1}(\bot),$$

para todo $n \in \mathbb{N}$, e temos que $f^n(\bot)$ é uma ω-cadeia.

Agora calculamos $f(\sqcup c)$ e verificamos que de fato temos um ponto fixo:

$$\begin{aligned}
f(\sqcup c) &= f(\sqcup\{f^n(\bot)\}) \\
&= \sqcup\{f(f^n(\bot))\} && (f \text{ é contínua}) \\
&= \sqcup\{f^n(\bot) : \ f \geq 1\} \\
&= \sqcup\{f^n(\bot) : \ f \geq 0\} && (\text{incluímos } f^0 = \bot, \text{ não muda o supremo.}) \\
&= \sqcup c.
\end{aligned}$$

Temos portanto $f(\sqcup c) = \sqcup c$, e mostramos que este supremo é ponto fixo.

Agora mostramos que $\sqcup c$ é o *menor* dentre os pontos fixos. Suponha que exista outro ponto fixo menor que $\sqcup c$, que chamamos de x. Então

$$\begin{aligned} \bot &\sqsubseteq x && \text{(por definição de } \bot) \\ f^n(\bot) &\sqsubseteq f^n(x) && (f \text{ é monótona}) \\ f^n(\bot) &\sqsubseteq x && \text{(supomos que } x \text{ é ponto fixo)} \end{aligned}$$

isto mostra que x é majorante de $\{f^n(\bot) : \ n \in \mathbb{N}\}$. Mas $\sqcup c$ é exatamente o supremo desse conjunto, portanto $\sqcup c \sqsubseteq x$, pela definição de supremo. □

3.7 Construindo novos domínios

Podemos construir domínios compostos, usando domínios já existentes. Tratamos nesta seção de algumas operações para composição de domínios: produto finito, união disjunta, e construção de domínios de funções contínuas.

3.7.1 Domínios primitivos

Domínios primitivos são aqueles que definimos como fundamentais. Por exemplo, podemos tomar "caracteres" e "inteiros" como fundamentais, e posteriormente construir domínios compostos como "strings" e "racionais" usando os domínios primitivos. Note que esta não há um conjunto fixo de domínios primitivos; podemos defini-los como quisermos.

É comum definir números naturais, cadeias (strings) de caracteres, valores-verdade (**true** e **false**) como primitivos. Também é usual definir *locais de memória* como tipo primitivo, separando o conceito de variável em "local" e "nome" (dois nomes podem indicar o mesmo local!)

3.7.2 Produto finito de domínios

Teorema 3.65. *Sejam $(A, \sqsubseteq_A)$ e $(B, \sqsubseteq_B)$ domínios, e seja*

$$A \times B = \{(a, b) \ : \ a \in A, b \in B\}.$$

Defina $(a, b) \sqsubseteq (x, y)$ se e somente se $a \sqsubseteq x$ e $b \sqsubseteq y$.

Então $(A \times B, \sqsubseteq)$ é um domínio, com menor elemento $\bot = (\bot_A, \bot_B)$.

Definição 3.66 (produto finito de domínios)**.** O domínio $(A \times B, \sqsubseteq)$ a que se refere o Teorema 3.65 é chamado de *produto finito* dos domínios A e B.

Definimos também as *funções projeção* $\pi_0 : A \times B \to A$ e $\pi_1 : A \times B \to B$, sendo $\pi_0(a, b) = a$ e $\pi_1(a, b) = b$. ♦

Lema 3.67. *Sejam E, D_0, D_1 domínios e seja $f : E \to D_0 \times D_1$ uma função. Então f é contínua se e somente se as duas funções $\pi_i \circ f$ forem contínuas.*

Lema 3.68. *Sejam E, D_0, D_1 domínios e seja $f : D_0 \times D_1 \to E$ uma função. Então f é contínua se e somente se é contínua em cada argumento separadamente.*

Lema 3.69. *Seja $A \times B$ um domínio, e $C \subset A \times B$ um pré-domínio. Então $\pi_i(A)$ é pré-domínio para $i = 0, 1$, e*

$$\bigsqcup A = \Big(\sqcup \pi_0(A), \ \sqcup \pi_1(A) \Big).$$

Teorema 3.70. *Sejam, $A \times B$ um domínio, com funções projeção π_0 e π_1. Então as duas funções projeção são contínuas.*

3.7.3 União disjunta de domínios

Teorema 3.71. *Sejam $(A, \sqsubseteq_A)$ e $(B, \sqsubseteq_B)$ domínios, e seja $A + B$ a união disjunta de A e B, ou seja,*

$$A + B = \{x_A \ : \ x \in A\} \cup \{y_B \ : \ y \in B\},$$

onde os elementos são marcados com rótulos que os identificam como "vindos de A" ou "vindos de B". Pode-se também denoar x_A e y_B por (x, a) e (y, b).

Defina a relação $\sqsubseteq$ da seguinte forma: $x_A \sqsubseteq y_B$ se e somente se $x \sqsubseteq_A y$ e $x \sqsubseteq_B y$.

Então $(A + B, \sqsubseteq)$ é um domínio, se elevado com um novo menor elemento $\bot$.

Definição 3.72 (união disjunta de domínios)**.** O domínio $(A + B, \sqsubseteq)$ a que se refere o Teorema 3.71 é chamado de *união disjunta dos domínios A e B*. ⧫

3.7.4 Domínios de funções

Teorema 3.73. *Sejam $(A, \sqsubseteq_A)$ e $(B, \sqsubseteq_B)$ domínios, e seja*

$$A \to B = \{f \mid f : A \to B \text{ é contínua}\}.$$

Defina a relação $\sqsubseteq$ da seguinte forma: se, para todo $a \in A$, $f(a) \sqsubseteq_B g(a)$, então $f \sqsubseteq g$.

Então $(A \to B, \sqsubseteq)$ é um domínio, com menor elemento igual à função $\bot : A \to B$, tal que

$$\bot(a) = \bot_B,$$

para todo $a \in A$.

Definição 3.74 (domínio de funções contínuas)**.** Quando A e B são domínios, o domínio $A \to B$ a que se refere o Teorema 3.73 é chamado de *domínio de funções contínuas* de A em B. ⧫

3.8 Semântica denotacional do *while*

Agora podemos definir a semântica do comando **while**.

Primeiro definimos o domínio de funções modificadoras de estado do programa. Denotamos por Σ o conjunto de todos os possíveis estados, definimos o domínio elevado de estados

$$\Sigma_\bot = \Sigma \cup \{\bot\}.$$

O domínio de funções semânticas para comandos é $(\Sigma \to \Sigma_\bot, \sqsubseteq)$, onde

$$f \sqsubseteq g \ \Leftrightarrow \ \forall \sigma \in \Sigma, \ f\sigma = \bot \text{ ou } f\sigma = g\sigma,$$

ou seja, a função f *concorda* com a função g em todos os pontos em que é definida, mas *pode estar definida em menos pontos que g.*

A menor função é $\bot_f : \Sigma \to \Sigma_\bot$, que *não é definida em nenhum estado.*

O Teorema 3.75 nos garante que podemos "extrair" de um comando **while** uma sequência de comandos **if**. A demonstração deste Teorema é pedida no exercício 28.

Teorema 3.75. $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \mathcal{C}[\![\textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}]\!]$

Para entender a relação entre a semântica do **while** e pontos fixos de ω-cadeias, definimos como exemplo uma sequência de comandos, começando com w_0, que é o mesmo que **while true do skip** (nunca termina), e em seguida incluindo w_1, que verifica b e executa c uma vez; w_2, que verifica b e executa c duas vezes; e assim por diante.

$$w_0 \stackrel{\text{def}}{=} \textbf{while true do skip}$$

$$w_{i+1} \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } c; w_i \textbf{ else skip}$$

Usando a função geradora

$$F = \lambda g \cdot \lambda \sigma \cdot \begin{cases} \sigma \text{ se } (\neg \mathcal{B}[\![b]\!]) \\ \bot \text{ se } (\mathcal{C}[\![c]\!]\sigma = \bot) \\ g(\mathcal{C}[\![c]\!]\sigma) \text{ caso contrário} \end{cases}$$

vemos que

$$\begin{aligned} \mathcal{C}[\![w_0]\!] &= \bot \\ \mathcal{C}[\![w_1]\!] &= F\bot \\ &\vdots \\ \mathcal{C}[\![w_i]\!] &= F^i\bot \end{aligned}$$

Agora observamos que os comandos w_i e **while** b **do** c são similares: eles só terão significado diferente se o **while** demorar mais que i iterações para terminar o loop (porque nesse caso o **while** termina e w_i não). Detalhamos este argumento nos dois parágrafos a seguir.

Se **while** b **do** c em σ termina o loop após exatamente n iterações,

$$\mathcal{C}[\![w_i]\!]\sigma = \begin{cases} \bot & \text{se } i < n \\ \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma & \text{se } i \geq n. \end{cases}$$

Se **while** b **do** c nunca termina em σ, então para todo i

$$\mathcal{C}[\![w_i]\!]\sigma = \bot = \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma.$$

Mas em ambos os casos, o significado de w_i, para i suficientemente grante, é o mesmo que o do comando **while** b **do** c. Então, para todo estado σ,

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \bigsqcup_{i=0}^{\infty} \mathcal{C}[\![w_i]\!]\sigma.$$

Escrevendo usando o funcional F,

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \bigsqcup_{i=0}^{\infty} \mathcal{C}[\![w_i]\!] = \bigsqcup_{i=0}^{\infty} F^i\bot = \operatorname{lfp} F.$$

O exercício 29 pede a demonstração de que a função geradora F é contínua.

Proposição 3.76. *A função F usada na semântica denotacional do comando* **while** *é contínua.*

Da continuidade de F, vemos que (i) a sequência $F^0\bot, F^1\bot, \ldots, F^n\bot$ é ω-cadeia em $\Sigma \to \Sigma_\bot$; e (ii) o ponto fixo de F é o supremo dessa ω-cadeia.

A semântica do **while** é dada, portanto, pelo ponto fixo mínimo de F:

$$[\![\textbf{while } b \textbf{ do } c]\!] = \operatorname{lfp} F$$

onde $F : (\Sigma \to \Sigma_\bot) \to (\Sigma \to \Sigma_\bot)$ é a função geradora a seguir.

$$F = \lambda g \cdot \lambda \sigma \cdot \begin{cases} \textbf{if } (\neg \mathcal{B}[\![b]\!]\sigma) \textbf{ then } \sigma \\ \textbf{if } (\mathcal{C}[\![c]\!]\sigma = \bot) \textbf{ then } \bot \\ \textbf{else } g(\mathcal{C}[\![c]\!]\sigma) \end{cases}$$

Note que não temos como apresentar uma descrição mais explícita de $\text{lfp}\, F$, porque não sabemos qual é o comando c, nem o estado σ. No entanto, sabemos que $\text{lfp}\, F$ existe e é único, logo nossa descrição da semântica do **while** está correta – e não é mais uma definição circular.

Listamos a seguir a função semântica $\mathcal{C}$ completa para nossa linguagem.

$$\begin{aligned}
\mathcal{C}[\![\textbf{skip}]\!] &= \lambda\sigma \cdot \sigma \\
\mathcal{C}[\![v := e]\!] &= \lambda\sigma \cdot \sigma[\mathcal{E}[\![e]\!]\sigma \,/\, v] \\
\mathcal{C}[\![c_1; c_2]\!] &= \lambda\sigma \cdot \textbf{if } (\mathcal{C}[\![c_1]\!]\sigma = \bot) \textbf{ then } \bot \textbf{ else } \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma) \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!] &= \lambda\sigma \cdot \textbf{if } (\mathcal{B}[\![b]\!]\sigma = \textbf{true}) \textbf{ then } \mathcal{C}[\![c_1]\!]\sigma \textbf{ else } \mathcal{C}[\![c_2]\!]\sigma \\
\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] &= \text{lfp}\, F,
\end{aligned}$$

com

$$F = \lambda g \cdot \lambda\sigma \cdot \begin{cases} \textbf{if } (\neg\mathcal{B}[\![b]\!]\sigma) \textbf{ then } \sigma \\ \textbf{if } (\mathcal{C}[\![c]\!]\sigma = \bot) \textbf{ then } \bot \\ \textbf{else } g(\mathcal{C}[\![c]\!]\sigma) \end{cases}$$

3.9 Entrada e Saída

Tendo finalizado a semântica denotacional de uma primeira pequena linguagem, faremos nela modificações. A primeira, de extrema simplicidade, é adicionar comandos para entrada e saída. Esta modificação é principalmente um exercício inicial, que nos obrigará a reconsiderar nossos domínios semânticos e a noção de estado. Isto se repetirá à medida que modificarmos mais a linguagem: para adicionar mais elementos, tornando-a mais útil, teremos que desmembrar o estado em diferentes componentes e usar mais domínios semânticos.

Até agora temos representado o estado como uma única entidade – uma função $\sigma : \textbf{Id} \to \mathbb{Z}$. Com comandos de leitura e escrita, precisaremos também de *buffers de entrada e saída*.

Os dois comandos de entrada e saída que usaremos são **read** e **write**; a gramática abstrata da linguagem é estendida como segue.

$\langle C \rangle ::=$ read $\langle I \rangle$ | write $\langle E \rangle$

Por exemplo, o programa a seguir calcula o volume de um paralelepípedo, dadas altura, largura e profundidade.

```
read x
read y
read z
write x * y * z
```

Claramente, a extensão para incluir strings e sintaxe mais amigável para **write**, como por exemplo

```
write ("O volume é", x * y * z)
```

é um exercício simples, com o qual não nos ocuparemos.

Além do estado do programa, teremos um *buffer de entrada* e um *buffer de saída*. Modelamos os buffers como sequências de números inteiros.

- estado, $\sigma \in \textbf{Id} \to \mathbb{N}$
- buffer de entrada, $i \in (\mathbb{N})^*$
- buffer de saída $o \in (\mathbb{N})^*$

Definimos novas operações nos dois novos domínios semânticos:

$$\begin{aligned}\text{put} &= \lambda no \cdot o \S n \\ \text{get} &= \lambda i \cdot \langle i \downarrow 1, i \dagger 1 \rangle\,,\end{aligned}$$

onde § significa concatenação; $i \downarrow k$ é o k-ésimo elemento da lista i; e $i \dagger k$ é a sequência i, com os primeiros k elementos removidos.

A função que representa a operação put aceita dois parâmetros: um elemento $n \in \mathbb{Z}$ e um buffer de saída o. O efeito é o de concatenar n em o, e retornar um novo buffer (note que § devolverá uma lista de inteiros – que é do mesmo tipo que o).

A função get faz o contrário: toma o buffer de entrada, separa a cabeça da cauda, e retorna as duas. A cabeça representa o próximo elemento lido; a cauda é o resto dos elementos ainda por ler.

A função wrong é usada para indicar erro.

$$\begin{aligned}\mathcal{C}[\![\textbf{write } e]\!]\sigma io = \; & \textbf{if } \text{writable?}(\mathcal{E}[\![e]\!]) \\ & \textbf{then } \langle \sigma, i, \text{put}(\mathcal{E}[\![e]\!]\sigma, o)\rangle \\ & \textbf{else } \text{wrong } \text{“non-writable object”} \\ \mathcal{C}[\![\textbf{read } v]\!]\sigma io = \; & \textbf{let } (x, i') = \text{get}(i) \textbf{ in} \\ & \langle \sigma[x : \; v], i', o\rangle\end{aligned}$$

Como o significado dos novos comandos modifica os buffers, toda a função semântica $\mathcal{C}$ deve ser reescrita para aceitar os dois novos argumentos (ainda que eles não sejam usados nos outros casos).

$$\begin{aligned}\mathcal{C}[\![\textbf{skip}]\!] &= \lambda\sigma io \cdot \langle \sigma, i, o\rangle \\ \mathcal{C}[\![v := e]\!] &= \lambda\sigma io \cdot \langle \sigma[v : \; \mathcal{E}[\![e]\!]\sigma], i, o\rangle \\ \mathcal{C}[\![c_1; c_2]\!] &= \lambda\sigma io \cdot \textbf{if } (\mathcal{C}[\![c_1]\!]\sigma io = \bot) \textbf{ then } \bot \textbf{ else } \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma io) \\ \mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!] &= \lambda\sigma io \cdot \textbf{if } (\mathcal{B}[\![b]\!]\sigma = \textbf{true}) \textbf{ then } \mathcal{C}[\![c_1]\!]\sigma io; \textbf{ else } \mathcal{C}[\![c_2]\!]\sigma io \\ \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] &= \text{lfp}\, F,\end{aligned}$$

com

$$F = \lambda g \cdot \lambda\sigma io \cdot \begin{cases} \textbf{if } (\neg\mathcal{B}[\![b]\!]\sigma) \textbf{ then } \langle \sigma, i, o\rangle \\ \textbf{if } (\mathcal{C}[\![c]\!]\sigma io = \bot) \textbf{ then } \bot \\ \textbf{else } g(\mathcal{C}[\![c]\!]\sigma io) \end{cases}$$

3.10 Ambientes

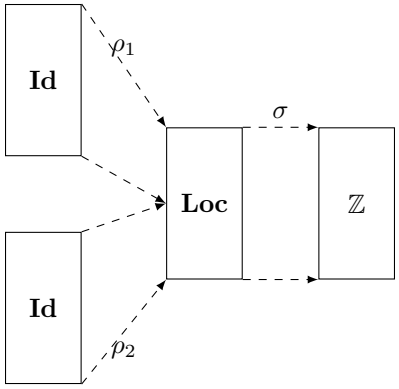
Até agora tratamos o estado do programa como uma função mapeando nomes de variáveis em valores. Para podermos construir a noção de escopo, dividiremos o estado em duas funções:

- Ambiente, que determina um lugar para cada nome de variável;
- Memória (*Store*), que determina um valor para cada lugar

Assim, temos os domínios semânticos:

- $\alpha \in \textbf{Loc}$, locais de armazenamento;
- $v \in \textbf{Id}$, identificadores;
- $\sigma \in \textbf{Sto} : \textbf{Loc} \to \mathbb{Z}$, memória;
- $\rho \in \textbf{Env}_V : \textbf{Id} \to \textbf{Loc}$, ambiente.

A notação $\mathbf{Env}_V$ significa que estes ambientes são os das variáveis (usaremos outro tipo de ambiente para procedimentos, cujo domínio semântico terá o nome $\mathbf{Env}_P$).



Uma vez definido um ambiente, é útil ter uma função que nos dá o valor associado a uma variável nele. Esta função é a composição da memória com o ambiente, que chamaremos de lookup:

$$\text{lookup}_{\rho\sigma} : \mathbf{Id} \to \mathbb{Z} = \sigma \circ \rho$$

Assim, as funções semânticas $\mathcal{E}$ e $\mathcal{B}$ tem seus domínios estendidos:

$$\mathcal{E} : \mathbf{Aexp} \times \mathbf{Env}_V \times \mathbf{Sto} \to \mathbb{Z}$$

Modificamos agora a denotação de variáveis.

$$\mathcal{E}[\![v]\!]\rho\sigma = \text{lookup}_{\rho\sigma}(v)$$

A memória é indexada por números naturais: há posições $0, 1, 2, \ldots$ As posições de memória são alocadas sequencialmente para as variáveis, e o programa em execução sempre deverá saber qual a próxima posição a ser alocada – para isso definimos a função new, que aloca uma nova posição e retorna seu índice.

$$\text{new} : \mathbf{Loc} \to \mathbb{N}$$

Não usaremos os buffers de entrada e saída definidos na seção 3.9, porque só deixaríamos a notação mais carregada, sem nenhum benefício para o desenvolvimento da semântica de variáveis locais, que é nosso foco agora. A função semântica $\mathcal{C}$ também aceitará dois argumentos, e, para que possamos realizar composições, seu resultado será do mesmo tipo que seus argumentos.

$$\mathcal{C} : \mathbf{Cmd} \to (\mathbf{Env}_V \times \mathbf{Sto} \to \mathbf{Env}_V \times \mathbf{Sto})$$

O corpo das definições da função semântica $\mathcal{C}$ não muda; muda apenas a quantidade de parâmetros (agora passamos ρ e σ).

$$\begin{aligned}
\mathcal{C}[\![\mathbf{skip}]\!] &= \lambda\rho\sigma \cdot \langle \rho, \sigma \rangle \\
\mathcal{C}[\![v := e]\!] &= \lambda\rho\sigma \cdot \mathbf{let}\ \text{loc} = \rho(v)\ \mathbf{in}: \\
&\qquad\qquad \langle \rho, \sigma[\text{loc} : \ \mathcal{E}[\![e]\!]\rho\sigma] \rangle \\
\mathcal{C}[\![c_1; c_2]\!] &= \lambda\rho\sigma \cdot \mathbf{if}\ (\mathcal{C}[\![c_1]\!]\rho\sigma = \bot)\ \mathbf{then}\ \bot\ \mathbf{else}\ \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\rho\sigma) \\
\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!] &= \lambda\rho\sigma \cdot \mathbf{if}\ (\mathcal{B}[\![b]\!]\rho\sigma = \mathbf{true})\ \mathbf{then}\ \mathcal{C}[\![c_1]\!]\rho\sigma\ \mathbf{else}\ \mathcal{C}[\![c_2]\!]\rho\sigma \\
\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] &= \text{lfp}\, F,
\end{aligned}$$

com

$$F = \lambda g \cdot \lambda\rho\sigma \cdot \begin{cases} \mathbf{if}\ (\neg\mathcal{B}[\![b]\!]\rho\sigma)\ \mathbf{then}\ \langle \rho, \sigma \rangle \\ \mathbf{if}\ (\mathcal{C}[\![c]\!]\rho\sigma = \bot)\ \mathbf{then}\ \bot \\ \mathbf{else}\ g(\mathcal{C}[\![c]\!]\rho\sigma) \end{cases}$$

3.11 Blocos e escopo

Uma vez que separamos o ambiente da memória, podemos incluir blocos na linguagem, para que possamos facilmente criar escopos, como no trecho de programa a seguir.

```
var x := 5
begin
    var x := 1
    // aqui, x vale 1
end
// aqui, x vale 5
```

A gramática abstrata é modificada, incluindo uma nova produção para comandos em bloco, e uma para declarações de variável.
Domínio sintático: $V \in \mathbf{Decl}_V$

$\langle C \rangle ::=$ begin $\langle V \rangle$ $\langle C \rangle$ end

$\langle V \rangle ::=$ var $\langle I \rangle := \langle E \rangle$; $\langle V \rangle \mid \epsilon$

Note que a regra para $\langle V \rangle$ permite repetição; as declarações podem vir em uma lista.

Cada bloco estende o ambiente com suas variáveis. Após o término do bloco, estas novas variáveis são retiradas do ambiente, e o ambiente anterior volta a ser usado. Isto nos dá uma nova função semântica $\mathcal{V}$, que estende o ambiente ρ com o nome da nova variável, e aloca para ela uma posição nova na memória (σ).

$$\mathcal{V} : \mathbf{Decl}_V \to \mathbf{Env}_V \times \mathbf{Sto} \to \mathbf{Env}_V \times \mathbf{Sto}$$

Na especificação da regra para declaração de variáveis, a sintaxe **var** $v := e; d$ significa uma declaração (**var** $v := e$) seguida de outras declarações (d).

$$\begin{aligned}
\mathcal{C}[\![\textbf{begin } d\ c \textbf{ end}]\!] &= \lambda\rho\sigma \cdot \textbf{let } \rho_2, \sigma_2 = \mathcal{V}[\![d]\!]\rho\sigma \textbf{ in :} \\
&\qquad \textbf{let } \rho_3, \sigma_3 = \mathcal{C}[\![c]\!]\rho_2, \sigma_2 \textbf{ in :} \\
&\qquad\quad \langle \rho, \sigma_3 \rangle \\
\mathcal{V}[\![\textbf{var } v := e; d]\!] &= \lambda\rho\sigma \cdot \textbf{let } \mathrm{loc} = \mathrm{new} \textbf{ in :} \\
&\qquad \langle \rho[v :\ \mathrm{loc}],\ \sigma[\mathrm{loc} :\ \mathcal{E}[\![e]\!]\rho\sigma] \rangle \\
\mathcal{V}[\![\epsilon]\!] &= \lambda\rho\sigma \cdot \langle \rho, \sigma \rangle
\end{aligned}$$

3.12 Procedimentos

Blocos só nos serão realmente úteis se os usarmos para definir procedimentos. Estendemos a linguagem com sintaxe para definir e chamar procedimentos, por enquanto sem a passagem de parâmetros.

```
proc fat:
begin
    var i := n
    var pr := 1
    while i>0 do
    begin
        i := i + 1
        pr := pr * i
    end
end
```

Adicionamos à gramática abstrata a definição e chamada de procedimentos. Note que estamos estendendo

dois domínios sintáticos: o dos comandos, com `call`, e o das declarações, com `proc`.
Domínio sintático: $P \in \mathbf{Decl}_P$

$\langle C \rangle ::= \mathbf{call}\ \langle I \rangle$

$\langle P \rangle ::= \mathbf{proc}\ \langle I \rangle : \langle C \rangle\ ;\ \langle P \rangle | \ \epsilon$

Um procedimento é um pequeno programa, que transforma um estado da memória em outro. O domínio semântico dos procedimentos é, portanto,

$$\mu \in \mathbf{Env}_P : \mathbf{Id} \to (\mathbf{Sto} \times \mathbf{Sto})$$

O domínio semântico $\mathbf{Env}_P$ é o dos procedimentos, que é igual ao dos programas. A função semântica $\mathcal{C}$ é estendida com $\mathcal{C}[\![\mathbf{call}\ p]\!]\mu\rho\sigma = \mu(p)$. Isto porque em $\mu(p)$ armazenamos justamente o significado semântico do procedimento com nome p – assim, a semântica de **call** é simples.

Detalhamos agora a função semântica $\mathcal{P}$, que dá o significado das *declarações* de procedimentos. Esta função calcula $f = \mathcal{C}[\![p]\!]$, para todo procedimento p na declaração, e devolve um novo ambiente μ, com este novo procedimento.

$$\mathcal{P} : \mathbf{Decl}_P \to (\mathbf{Env}_P \times \mathbf{Env}_V) \to \mathbf{Env}_P$$

$$\begin{aligned}
\mathcal{P}[\![\mathbf{proc}\ p : c; d]\!] &= \lambda\mu\rho \cdot \mathbf{let}\ f = \mathcal{C}[\![c]\!]\mu\rho\ \mathbf{in} : \\
&\qquad \mathcal{P}[\![d]\!](\mu[p : f])\rho \\
\mathcal{P}[\![\epsilon]\!] &= \lambda\mu\rho \cdot \rho \\
\mathcal{C}[\![\mathbf{call}\ p]\!] &= \lambda\mu\rho\sigma \cdot \mu(p)
\end{aligned}$$

3.13 Passagem de parâmetros

Antes de apresentar a semântica de diferentes formas de passagem de parâmetros, discutiremos informalmente algumas delas.

- **avaliação estrita:** cada parâmetro é avaliado completamente antes de se passar o controle ao procedimento;
  - **por valor:** o valor da expressão é calculado, e o resultado é atribuído ao parâmetro formal. É a mais conhecida forma de passagem de parâmetro.
  - **por referência:** só é possível quando o argumento é uma variável; o local de memória da variável é compartilhado entre a variável usada como argumento e o parâmetro formal (equivale a parâmetros `var` em Pascal, e é simulado com passagem de ponteiros em C.
  - **por compartilhamento:** semelhante à passagem por referência, exceto que atribuições feitas a parâmetros não são visíveis para ao método que chamou; já *mutações* nestes parâmetros, são.
- **avaliação não-estrita:** o controle é passado ao procedimento, e cada parâmetro é computado à medida que é necessário.
  - **por nome:** o parâmetro é calculado cada vez que necessário, usando o *ambiente* que existia na chamada do procedimento, mas com a *memória* do momento em que é usado.
  - **por texto:** o texto da expressão passada como parâmetro é substituído no corpo do procedimento.

Note que a avaliação estrita corresponde à computação de funções estritas.

Há outras formas de passagem de parâmetros que não discutimos aqui; alguns exemplos de semântica de passagem de parâmetros são suficientes para a compreensão de outros mecanismos.

3.13.1 Declaração

Não fixamos a quantidade de argumentos que um procedimento pode ter, por isso o ambiente de procedimentos passa agora a ser uma tupla. O i-ésimo elemento é o ambiente μ_i, que mapeia nomes e argumentos de procedimentos com i argumentos em funções transformadoras de estado:

$$\mu = (\mu_0, \mu_1, \mu_2, \ldots, \mu_k),$$

$$\begin{aligned}
\mu_0 &: \mathbf{Id} \to (\mathbf{Sto} \times \mathbf{Sto}) \\
\mu_1 &: \mathbf{Id} \times \mathbb{Z} \to (\mathbf{Sto} \times \mathbf{Sto}) \\
\mu_2 &: \mathbf{Id} \times \mathbb{Z} \times \mathbb{Z} \to (\mathbf{Sto} \times \mathbf{Sto}) \\
&\vdots
\end{aligned}$$

A função semântica para declaração de procedimentos será

$$\mathcal{P} : \mathbf{Decl}_P \times \mathbf{Env}_P \times \mathbf{Env}_V \to \mathbf{Env}_P \times \mathbf{Env}_V,$$

que detalharemos nas próximas seções.

3.13.2 Por valor

Cada declaração modifica o ambiente de procedimentos, incluindo mais um, e o retorna.

Para realizar chamadas de procedimento por valor, o comando **call** avalia cada expressão passada como parâmetro, e depois usa os valores resultantes como argumentos.

$$\begin{aligned}
\mathcal{P}[\![\mathbf{proc}\ p(v_1, v_2, \ldots, v_n) : c; d]\!] &= \lambda\mu\rho \cdot \big(\mathbf{let}\ f = \lambda v_1 v_2 \ldots v_n \cdot \mathcal{C}[\![c]\!]\mu\rho\ \mathbf{in}: \\
&\qquad \mathcal{P}[\![d]\!](\mu[p :\ f])\rho\big) \\
\mathcal{P}[\![\epsilon]\!] &= \lambda\mu\rho \cdot \langle\mu, \rho\rangle \\
\mathcal{C}[\![\mathbf{call}\ p(e_1, e_2, \ldots, e_n)]\!]\mu\rho\sigma &= \big(\mathbf{let}\ q = \mu(p)\ \ \mathbf{in}: \\
&\qquad \mathbf{let}\ v_1 = \mathcal{E}[\![e_1]\!]\mu\rho\sigma;\ \ldots;\ v_n = \mathcal{E}[\![e_n]\!]\mu\rho\sigma\ \mathbf{in}: \\
&\qquad q(v_1, \ldots, v_n)\big)
\end{aligned}$$

Poderíamos também ter escrito

$$\begin{aligned}
\mathcal{P}[\![\mathbf{proc}\ p(v_1, v_2, \ldots, v_n) : c; d]\!] &= \lambda\mu\rho \cdot \big(\mathbf{let}\ f = \Lambda v_1 v_2 \ldots v_n \cdot \mathcal{C}[\![c]\!]\mu\rho\ \mathbf{in}: \\
&\qquad \mathcal{P}[\![d]\!](\mu[p :\ f])\rho\big) \\
\mathcal{P}[\![\epsilon]\!] &= \lambda\mu\rho \cdot \langle\mu, \rho\rangle \\
\mathcal{C}[\![\mathbf{call}\ p(e_1, e_2, \ldots, e_n)]\!]\mu\rho\sigma &= \big(\mathbf{let}\ q = \mu(p)\ \ \mathbf{in}: \\
&\qquad q(\mathcal{E}[\![e_1]\!]\mu\rho\sigma, \ldots \mathcal{E}[\![e_n]\!]\mu\rho\sigma)\big)
\end{aligned}$$

onde Λ é o equivalente de λ, mas forçando avaliação estrita dos argumentos (ou seja, Λ denota uma função estrita, que resulta em $\perp$ sempre que algum de seus argumentos é $\perp$).

3.13.3 Por nome

Para passagem por nome **call** criará, para cada argumento uma função que recebe uma memória e retorna o valor do argumento.

$$\begin{aligned}
\mathcal{P}[\![\mathbf{proc}\ p(v_1, v_2, \ldots, v_n) : c; d]\!] &= \lambda\mu\rho \cdot \big(\mathbf{let}\ f = \lambda v_1 v_2 \ldots v_n \cdot \mathcal{C}[\![c]\!]\mu\rho\ \mathbf{in}: \\
&\qquad \mathcal{P}[\![d]\!](\mu[p :\ f])\rho\big) \\
\mathcal{P}[\![\epsilon]\!] &= \lambda\mu\rho \cdot \langle\mu, \rho\rangle \\
\mathcal{C}[\![\mathbf{call}\ p(e_1, e_2, \ldots, e_n)]\!]\mu\rho\sigma &= \big(\mathbf{let}\ q = \mu(p)\ \ \mathbf{in}: \\
&\qquad q(\mathcal{E}[\![e_1]\!]\mu\rho, \ldots \mathcal{E}[\![e_n]\!]\mu\rho)\big)
\end{aligned}$$

Note que usamos λ na definição do procedimento, e não Λ. Além disso, na chamada do procedimento, usamos $\mathcal{E}[\![e_1]\!]\mu\rho$, sem passar a memória σ.

Exercícios

Ex. 11 — (Aquecimento) Construa uma função semântica para números usando representação romana antiga.
Os valores de cada numeral romano são

I	1	C	100
V	5	D	500
X	10	M	1000
L	50		

Os números são construídos usando as seguintes regras:

- A repetição até três vezes de um numeral representa soma: II=2, III=3, Não se pode repetir V, L e D.
- Concatenar númerais em ordem decrescente representa adição: XVII = 10+5+2=17.
- Números escritos em ordem crescente representam subtração: XL = 50=10 = 40. *Somente* são permitidas as seguintes representações desse tipo: IV, IX, XL, XC, CD, CM.
- Uma barra acima do numeral o multiplica por mil: $\overline{\text{L}}$=50 000.

Ex. 12 — (Aquecimento) Prove que:

a) de fato não é necessário repetir V, L e D em números romanos (ou seja, que podemos representar as mesmas quantidades, repetindo ou não esses numerais).

b) há como representar a mesma quantidade de duas maneiras, e que poderíamos proibir a barra em alguns dos símbolos (quais?)

Ex. 13 — (Aquecimento) Proponha uma ordem parcial para distribuições de probabilidade sobre um espaço amostral. Sua ordem parcial é pré-domínio? Domínio?

Ex. 14 — (Aquecimento) Expanda a linguagem dada como exemplo com expressões envolvendo condicionais:
`if bool then expr1 else expr2`
é uma expressão *inteira*, significando "se bool for verdade, epr1, senão, expr2".

a) Exiba sintaxe concreta e abstrata para a nova versão da atribuição.

b) Mostre a semântica desse comando.

Ex. 15 — (Aquecimento) Usando a semântica da nossa linguagem, determine o significado do trecho de código: `a := b; b:= a`.

Ex. 16 — (Aquecimento) Seja Σ um alfabeto. Determine se (Σ, pref), onde pref é a relação "é prefixo de", é conjunto parcialmente ordenado.

Ex. 17 — (Aquecimento) Determine uma classe de funções que seja monotônica em $(\mathbb{N}, |)$.

Ex. 18 — Determine três ordens parciais que façam de qualquer conjunto de intervalos reais um pré-domínio. Identifique quando também for um domínio, *sem a necessidade de elevação.*

Ex. 19 — Modifique a semântica da nossa linguagem para usar inteiros como booleanos, como na linguagem C.

Ex. 20 — Seja $\mathcal{V}(K)$ o conjunto de todos os espaços vetoriais de dimensão finita sobre um corpo K. Por exemplo, $\mathcal{V}(\mathbb{R}) = \{0\} \cup \mathbb{R} \cup \mathbb{R}^2 \cup \mathbb{R}^3 \cup \ldots$

a) Mostre que a relação "subespaço de" induz uma ordem parcial em $\mathcal{V}$. (Pode considerar que, havendo isomorfismo com subespaço, é subespaço)

b) $\mathcal{V}$ é pré-domínio?

c) $\mathcal{V}$ é um domínio?

Ex. 21 — Prove que em qualquer ordem parcial, o menor elemento é único.

Ex. 22 — O conjunto de funções $\mathbb{N} \to \mathbb{R}^+$, com a ordem $\mathcal{O}$ (usada em análise de algoritmos) é ordem parcial? É pré-domínio? Domínio?

Ex. 23 — Modifique a semântica da linguagem desenvolvida neste Capítulo para incluir um valor de erro, "**err**", em expressões. Sempre que uma expressão resultar em erro, ele deve ser propagado para outras expressões (uma conta envolvendo **err** sempre resulta em **err**). Por exemplo, $x/0$ resulta em **err**. No programa

```
y := x/0
z := 0 + (y-y)
```

o estado final é $[y \mapsto \textbf{err}\ ;\ z \mapsto \textbf{err}]$.

Ex. 24 — Defina um comando **repeat**. O comando **repeat** c **until** b é semelhante ao **while**, mas que só verifica a condição de parada ao final do loop – e o loop para quando b é *verdadeiro*.

```
repeat
    ...
until x = 0
```

Inclua-o na semântica denotacional de nossa linguagem.

Ex. 25 — Considere a Lógica de Primeira Ordem sobre expressões inteiras. Queremos poder usar predicados onde os objetos são afirmações sobre inteiros, incluíndo números, variáveis, comparações $(\leq, =, \geq)$, operações aritméticas. Por exemplo:

$$\forall x, \forall, y, \exists z,\ \ yz = x.$$

a) Exiba sintaxe concreta e abstrata para essa Lógica. (Apesar de não ser uma linguagem de programação, podemos definir sintaxe para ela; livros de Lógica básica normalmente definem a sintaxe da Lógica de Primeira Ordem).

b) Mostre a semântica dessa Lógica. Os domínios semânticos não são funções transformadoras de estado, porque não temos atribuições. Temos expressões lógicas e predicados, que podem ser verdadeiros ou falsos, e expressões inteiras.

Ex. 26 — Expanda a linguagem dada como exemplo para incluir dupla avaliação:

```
a,b := expr1, expr2
```

a) Exiba sintaxe concreta e abstrata para a nova versão da atribuição.

b) Mostre que há *ao menos duas* possíveis semânticas para esse comando, e que elas *não* são equivalentes.

Ex. 27 — Quando expandimos a linguagem comcomandos de entrada e saída, construímos dois *comandos*, `read` $\langle I \rangle$ e `write` $\langle E \rangle$. Faça o mesmo, mas sem incluir um comando para leitura – a leiura deve ser feita como parte de uma expressão, como nos exemplos a seguir:

```
x := 2 + read
y := read
if read < 0  then ...
```

Ex. 28 — Demonstre o teorema 3.75.

Ex. 29 — Demonstre a proposição 3.76.

Ex. 30 — (Adaptado de Schmidt) Inventamos um comando **entangle**, que obedece a seguinte propriedade:

$$\mathcal{C}[\![\textbf{entangle } b \textbf{ in } c]\!] = \mathcal{C}[\![\textbf{if } b \textbf{ then } (c; \textbf{entangle } b \textbf{ in } c; c) \textbf{ else } c]\!]$$

a) Descreva informalmente o significado do comando **entangle** b **in** c.

b) Defina a semântica denotacional para **entangle**, e prove que ela tem a propriedade acima.

c) Que dificuldade você teria para implementar **entangle** em um compilador?

Ex. 31 — Mude a semântica da linguagem para que não seja possível usar variáveis não declaradas.

Ex. 32 — Adicione ponteiros à linguagem, dando sua semântica denotacional.

Ex. 33 — Adicione funções (procedimentos com valor de retorno) à linguagem.

Ex. 34 — Nossa linguagem permite usar o mesmo identificador para nomear uma variável e um procedimento.

i) Aproveite este fato para introduzir a maneira de Pascal de retornar valor de uma função. Ao invés de "`return k`", em Pascal fazemos "`funcao := k`". Por exemplo, no programa a seguir

```
function f(x:real): real
begin
    f := x * 2;
end
```

a função retorna `x * 2`, porque este valor é atribuído à variável que tem o mesmo nome da função.

ii) Modifique a semântica para que variáveis e procedimentos não possam compartilhar nomes (isto é, usem o mesmo espaço de nomes).

Ex. 35 — Mude a linguagem para que tenhamos procedimentos de primeira classe (ou seja, que possam ser armazenados em variáveis e passados como parâmetros).

Ex. 36 — Modifique a semântica da declaração de procedimentos para que seja possível definir procedimentos recursivos.

Ex. 37 — Modifique a semântica para suportar:

a) passagem de parâmetros por referência.

b) passagem de procedimentos por referência.

c) passagem de operadores por referência (você precisará adicionar definições de operadores).

Capítulo 4

Semântica Denotacional: com continuações

Há dispositivos de controle em linguagens de programação que escapam da construção que usamos até agora, que somente permite execução sequencial de comandos. Alguns exemplos são o tratamento de exceções; desvios incondicionais (`goto`) e co-rotinas. O mecanismo que usamos para a modelagem destes é o das *continuações*.

A formulação que usamos nos Capítulos anteriores é chamada de *semântica denotacional direta*, e a que usaremos neste capítulo é a *semântica denotacional com continuações* (ou "no estilo de continuações"[1]).

Uma *continuação* representa, em qualquer momento da execução de um comando, o "resto da computação" (da transformação de estados) que deve acontecer após a execução deste comando.

Suponha que queiramos modificar a primeira versão de nossa função semântica para comandos, $\mathcal{C}$. Esta função era do tipo

$$\mathcal{C} : (\mathbf{Cmd} \times \mathbf{Sto}) \to \mathbf{Sto},$$

A versão modificada desta função semântica, com continuações, é

$$\mathcal{K} : (\mathbf{Cmd} \times \mathbf{Cont} \times \mathbf{Sto}) \to \mathbf{Sto}.$$

Dado um comando, uma continuação, e um estado, um novo estado é produzido. Se $c \in \mathbf{Cmd}$, $\kappa \in \mathbf{Cont}$, $\sigma, \sigma' \in \mathbf{Sto}$, então teremos

$$\mathcal{K}\Big(\llbracket c \rrbracket, \underbrace{\kappa}_{\mathbf{Sto}\times\mathbf{Sto}}, \sigma\Big) \longrightarrow \sigma'.$$

Para a linguagem que definimos inicialmente, a semântica com continuações é dada a seguir.

$$\begin{aligned}
\mathcal{K}\llbracket \mathbf{skip} \rrbracket \kappa\sigma &= \kappa\sigma \\
\mathcal{K}\llbracket v := e \rrbracket \kappa\sigma &= \kappa\sigma[v : \ \mathcal{E}\llbracket e \rrbracket \sigma] \\
\mathcal{K}\llbracket c_1; c_2 \rrbracket \kappa\sigma &= \mathcal{K}\llbracket c_1 \rrbracket (\mathcal{K}\llbracket c_2 \rrbracket \kappa)\sigma \\
\mathcal{K}\llbracket \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rrbracket \kappa\sigma &= \mathbf{if}\ (\mathcal{B}\llbracket b \rrbracket \sigma = \mathbf{true})\ \mathbf{then}\ \mathcal{C}\llbracket c_1 \rrbracket \kappa\sigma\ \mathbf{else}\ \mathcal{C}\llbracket c_2 \rrbracket \kappa\sigma \\
\mathcal{K}\llbracket \mathbf{while}\ b\ \mathbf{do}\ c \rrbracket \kappa &= \mathrm{lfp}\, F,
\end{aligned}$$

com

$$F = \lambda g \cdot \lambda w\sigma \cdot \begin{cases} \mathbf{if}\ (\mathcal{B}\llbracket b \rrbracket \sigma)\ \mathbf{then}\ g(\mathcal{K}\llbracket c \rrbracket w\sigma) \\ \mathbf{else}\ \kappa\sigma \end{cases}$$

[1] Em Inglês, "direct style denotational semantics" e "continuation style denotational semantics".

4.1 Término com `abort`

Muitas linguagens oferecem um comando que permite interromper completamente a execução do programa. Aqui daremos a este comando o nome[2] de **abort**.

Nesta seção expomos a semântica denotacional simplificada do comando **abort**. A sintaxe abstrata do comando é trivial:

$\langle C \rangle ::=$ **abort**

Podemos definir que o comando **abort** termine deixando a memória da forma como estava. A única diferença, então, entre **abort** e **skip**, é que **abort** *não aplica a continuação que lhe foi passada,* e não havendo mais o "resto da computação", o programa termina.

$$\mathcal{K}[\![\textbf{abort}]\!]\kappa\sigma = \sigma$$

Ou, se usarmos ambientes, $\mathcal{K}[\![\textbf{abort}]\!]\kappa\rho\sigma = \rho\sigma$. Na verdade, podemos resumir de maneira mais geral a ação de **abort** descrevendo que o significado de **abort**, fixada qualquer continuação, é a função identidade para os outros argumentos:

$$\mathcal{K}[\![\textbf{abort}]\!]\kappa = \text{id}$$

4.2 Tratamento de exceções

O mecanismo usual (com variações) de tratamento de erros em linguagens de programação modernas é o de *tratamento de exceções*.

Situações excepcionais durante a execução do programa requerem que o controle seja transferido para uma seção de tratamento de erros. Um exemplo de como isto pode ser implementado em uma linguagem é o mecanismo **try** .. **catch**, existente em Java, Ruby e Python[3]. O código a seguir ilustra este mecanismo.

```
try
    c_1
    throw z
    c_2
catch x:
    c_3 (x)
```

Esta construção começa com a execução do comando `c_1`. Quando uma exceção ocorre, `throw z` envia o controle para a região do `catch`, e `c_3` é executado.

A gramática abstrata é estendida com mais uma única produção para acomodar comandos protegidos por **try** .. **catch**.

$\langle C \rangle ::=$ **try** $\langle C \rangle$ **catch** $\langle I \rangle : \langle C \rangle$

Para construírmos o significado semântico do conjunto **try** .. **throw** .. **catch**, criamos um *ambiente de exceções* – o domínio semântico $\mathbf{Env}_E$, que mapeia exceções em continuações. Para simplificar a exposição, identificamos cada exceção com seu nome e temos

$$\mathbf{Env}_E : \mathbf{Id} \to \mathbf{Cont}.$$

Em **try** c_1 **catch** $x : c_2$, devemos primeiro atualizar o ambiente de exceções, mapeando x no significado de c_2 (que é uma continuação), e depois usar este ambiente aumentado para o significado da execução de c_1.

[2] Em Erlang, Java[a] e Pascal, `halt`; em C, `abort`; em Ruby, `exit!`; em Perl, `die`; em Python e Lua, `exit`; em Ada, `Abort_Task`; em Forth, `bye`; em Fortran, `stop`; em Haskell, `exitFailure`; em Javascript, `quit`.

[3] Em Python usa-se `try..except`, e em Ruby usa-se `rescue`, sem o `try`, que fica implícito no início do bloco onde há um `raise`. Estas diferenças só são relevante em termos de sintaxe concreta.

[a] Em Java, `exit` sai após "limpar" o ambiente; `Runtime.getRuntime().halt()` sai imediatamente.

O comando **throw** x simplesmente pesquisa no ambiente corrente de exceções por x, que estará mapeado em uma continuação; descarta a continuação anterior e usa esta. Com isto o controle muda para o trecho de tratamento de exceção dentro do **catch**.

$$\mathcal{K} : \mathbf{Cmd} \to (\mathbf{Env}_E \times \mathbf{Cont} \times \mathbf{Sto}) \to \mathbf{Sto}$$

$$\begin{aligned}\mathcal{K}[\![\mathbf{try}\ c_1\ \mathbf{catch}\ x :\ c_2]\!]\tau\kappa\sigma &= \big(\mathbf{let}\ \tau' = \tau[x :\ \mathcal{K}[\![c_2]\!]\tau\kappa]\ \ \mathbf{in}: \\ &\qquad \mathcal{K}[\![c_1]\!]\tau'\kappa\sigma\big) \\ \mathcal{K}[\![\mathbf{throw}\ x]\!]\tau\kappa\sigma &= \tau(x)(\sigma)\end{aligned}$$

Note que na primeira linha da definição de $\mathcal{K}[\![\mathbf{try}\ c_1\ \mathbf{catch}\ x :\ c_2]\!]$, a expressão "$\mathcal{K}[\![c_2]\!]\tau\kappa$" é uma continuação (é do tipo $\mathbf{Sto} \to \mathbf{Sto}$).

Novamente apresentamos a função semântica dos comandos de nossa linguagem, desta vez com tratamento de exceções e **abort**.

$$\begin{aligned}\mathcal{K}[\![\mathbf{skip}]\!]\tau\kappa\sigma &= \kappa\sigma \\ \mathcal{K}[\![v := e]\!]\tau\kappa\sigma &= \kappa\sigma[v :\ \mathcal{E}[\![e]\!]\sigma] \\ \mathcal{K}[\![c_1; c_2]\!]\tau\kappa\sigma &= \mathbf{if}\ \mathcal{K}[\![c_1]\!]\tau\big(\mathcal{K}[\![c_2]\!]\tau\kappa\big)\sigma \\ \mathcal{K}[\![\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]\tau\kappa\sigma &= \mathbf{if}\ (\mathcal{B}[\![b]\!]\sigma = \mathbf{true})\ \mathbf{then}\ \mathcal{C}[\![c_1]\!]\tau\kappa\sigma\ \mathbf{else}\ \mathcal{C}[\![c_2]\!]\tau\kappa\sigma \\ \mathcal{K}[\![\mathbf{abort}]\!]\tau\kappa\sigma &= \sigma \\ \mathcal{K}[\![\mathbf{try}\ c_1\ \mathbf{catch}\ x :\ c_2]\!]\tau\kappa\sigma &= \big(\mathbf{let}\ \tau' = \tau[x :\ \mathcal{K}[\![c_2]\!]\tau\kappa\sigma]\ \ \mathbf{in}: \\ &\qquad \mathcal{K}[\![c_1]\!]\tau'\kappa\sigma\big) \\ \mathcal{K}[\![\mathbf{throw}\ x]\!]\tau\kappa\sigma &= \tau(x)(\sigma) \\ \mathcal{K}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\tau\kappa &= \mathrm{lfp}\, F,\end{aligned}$$

com

$$F = \lambda g \cdot \lambda\tau w\sigma \cdot \begin{cases} \mathbf{if}\ (\mathcal{B}[\![b]\!]\sigma)\ \mathbf{then}\ g(\mathcal{K}[\![c]\!]\tau w\sigma) \\ \mathbf{else}\ \kappa\sigma \end{cases}$$

Exercícios

Ex. 38 — Use continuações para modelar o infame comando **goto**. Um comando poderá ter um *rótulo*, e o comando **goto** x transfere o controle para o comando com o rótulo x. O trecho de programa a seguir ilustra o uso do `goto`.

```
x := 0
goto saida
x := 10
label saida: y := x
```

Depois da execução, o estado mapeará tanto x como y em zero, porque a linha `x := 10` não será executada. A gramática abstrata terá mais duas produções:

$\langle C \rangle$ ::=**label** I : $\langle C \rangle$
| **goto** I

Ex. 39 — Defina a semântica do **abort** sem usar continuações.

Ex. 40 — Em Common Lisp as exceções são reiniciáveis – é possível retornar do tratamento de uma exceção. Já em Ada este comportamento é proibido. Mostre como modelar, com semântica denotacional e continuações, exceções continuáveis como as de Common Lisp.

Ex. 41 — Modifique o mecanismo de tratamento de exceções para **throw** possa aceitar, além do tipo de exceção, parâmetros formais:

```
try
...
throw some_exception(x,y,z)
...
catch some_exception(a,b,c):
...
// use a, b, c
```

Ex. 42 — Dê a semântica denotacional do comando `yield`, que pode ser usado para implementar co-rotinas.

Ex. 43 — A linguagem INTERCAL tem o comando **comefrom**, com efeito oposto ao do **goto**:

- **label** L declara que daqui o controle pode ir para outro ponto do programa.
- **comefrom** L é uma declaração, que diz que o controle deve vir de L para este ponto.

Note que pode haver vários **comefrom** para cada rótulo, e a semântica neste caso pode ser definida de maneiras interessantes (não-deterministicamente; criando threads de execução; por rodízio; etc). Construa uma semântica denotacional para o **comefrom**.

Capítulo 5

Semântica Operacional

A semântica operacional de uma linguagem descreve *como* programas são executados. Esta descrição é feita por um mapeamento de cada trecho do programa em uma máquina abstrata. Inicialmente, nos interessa que nossa máquina virtual tem variáveis, e que podemos modificar o conteúdo delas.

Um *estado* de um programa é uma função de variáveis em $\mathbb{N}$. Uma *configuração* é o estado atual, junto com o programa que ainda resta para ser executado.

Definição 5.1 (configuração)**.** Uma *configuração* de um programa é um par $\langle c, \sigma\rangle$, onde c é um programa e σ é um estado. ♦

Regras de semântica natural (ou de passo largo) determinam a configuração *final* de um programa, e são portanto da forma

$$\langle c, \sigma\rangle \Downarrow \sigma'.$$

Regras de semântica estrutural (ou de passo pequeno) determinam uma nova configuração, *não necessariamente final*, para o programa, e portanto podem ser de uma das duas formas a seguir.

$$\langle c, \sigma\rangle \to \langle c', \sigma'\rangle$$
$$\langle c, \sigma\rangle \to \sigma'$$

Usamos a notação de *regras de reescrita*: uma regra de semântica operacional é da forma

$$\frac{\text{premissas}}{\text{conclusão}}$$

Por exemplo,

$$\frac{\sigma \vdash e_1 \Downarrow n_1 \quad \sigma \vdash e_2 \Downarrow n_2}{\sigma \vdash e_1 + e_2 \Downarrow n_1 + n_2}$$

$\sigma \vdash e_1 \Downarrow n_1$ significa que "na presença de σ, a expressão e_1 leva ao número n_1".

Nas seções a seguir, definimos a semântica operacional para comandos de nossa linguagem.

5.1 Semântica Natural ("de passo largo")

Começamos pelas duas regras mais simples, que teremos como dois axiomas. Primeiro, se o programa a ser executado é somente **skip** no estado σ, então o estado não será alterado. Da mesma forma, se o programa a ser executado é uma atribuição de valor a variável, o efeito é somente o de modificar o estado.

$$\texttt{nul} \quad \langle \textbf{skip}, \sigma\rangle \Downarrow \sigma$$
$$\texttt{atr} \quad \langle v := e, \sigma\rangle \Downarrow \sigma[v : \mathcal{E}[\![e]\!]\sigma]$$

Usamos uma única regra para sequenciamento, que determina que se c_1 leva de σ a σ', e c_2 leva de σ' a σ'', então a composição dos dois leva de σ a σ''.

$$\texttt{seq} \quad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma', \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma''}$$

As regras para o comando **if** são também bastante simples. Temos uma para quando a condição é verdadeira *no ambiente atual* e outra para quando a condição é falsa.

$$\texttt{ifT} \quad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \Downarrow \sigma'} \qquad (\mathcal{B}[\![b]\!]\sigma = \textbf{true})$$

$$\texttt{ifF} \quad \frac{\langle c_2, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \Downarrow \sigma'} \qquad (\mathcal{B}[\![b]\!]\sigma = \textbf{false})$$

Para o **while** temos também duas regras. Quando $\mathcal{B}[\![b]\!]\sigma$ é falso, simplesmente não alteramos o estado. Quando é verdadeiro, verificamos o efeito do comando c sobre o estado σ e dizemos que a semântica é a mesma que se executarmos c uma vez, e depois executarmos novamente o **while**

$$\texttt{enqT} \quad \frac{\langle c, \sigma\rangle \Downarrow \sigma', \quad \langle \textbf{while } b \textbf{ do } c, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma''} \qquad (\mathcal{B}[\![b]\!]\sigma = \textbf{true})$$

$$\texttt{enqF} \quad \langle \textbf{while } \ b \ \textbf{ do } \ c, \sigma\rangle \Downarrow \sigma \qquad (\mathcal{B}[\![b]\!]\sigma = \textbf{false})$$

A semântica natural para nossa linguagem é resumida a seguir.

$$\texttt{nul} \quad \langle \textbf{skip}, \sigma\rangle \Downarrow \sigma$$

$$\texttt{atr} \quad \langle v := e, \sigma\rangle \Downarrow \sigma[v : \ \mathcal{E}[\![e]\!]\sigma]$$

$$\texttt{seq} \quad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma', \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma''}$$

$$\texttt{ifT} \quad \frac{\sigma \vdash b \Downarrow \textbf{true} \quad \langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \Downarrow \sigma'}$$

$$\texttt{ifF} \quad \frac{\sigma \vdash b \Downarrow \textbf{false} \quad \langle c_2, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \Downarrow \sigma'}$$

$$\texttt{enqT} \quad \frac{\sigma \vdash b \Downarrow \textbf{true} \quad \langle c, \sigma\rangle \Downarrow \sigma', \quad \langle \textbf{while } b \textbf{ do } c, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma''}$$

$$\texttt{enqF} \quad \frac{\sigma \vdash b \Downarrow \textbf{false}}{\langle \textbf{while } \ b \ \textbf{ do } \ c, \sigma\rangle \Downarrow \sigma}$$

Note que podemos definir uma função semântica semelhante à que definimos com semântica denotacional.

Definição 5.2 (função semântica natural)**.** Definimos a função semântica natural como

$$\mathcal{C}_N[\![c]\!]\sigma = \sigma' \ \Leftrightarrow \ \langle c, \sigma\rangle \Downarrow \sigma'. \qquad \blacklozenge$$

5.2 Semântica Estrutural ("de passo curto")

A diferença da semântica natural para a estrutural é que na estrutural queremos poder representar execução parcial do programa; queremos regras que não nos deem somente estados finais, mas também *configurações subsequentes*. Por isso precisamos modificar todas as regras, exceto a de **skip** e a de atribuições.

Para o sequenciamento $(c_1; \ldots)$, temos uma regra para a situação em que c_1 produz um estado final, e outra para a situação em que resulta em uma nova configuração.

$$\texttt{seq}^1 \quad \frac{\langle c_1, \sigma\rangle \to \langle c_2, \sigma'\rangle}{\langle c_1; c_3, \sigma\rangle \to \langle c_2; c_3, \sigma'\rangle}$$

$$\texttt{seq}^2 \quad \frac{\langle c_1, \sigma\rangle \to \sigma'}{\langle c_1; c_2, \sigma\rangle \to \langle c_2, \sigma'\rangle}$$

Novamente temos duas regras para **if**. Note que o comando **if** b **then** c_1 **else** c_2 em um estado σ não produzirá um estado final; ele simplesmente seleciona entre $\langle c_1, \sigma\rangle$ e $\langle c_2, \sigma\rangle$.

$$\texttt{ifT} \quad \frac{\sigma \vdash b \to \textbf{true}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \langle c_1, \sigma\rangle}$$

$$\texttt{ifF} \quad \frac{\sigma \vdash b \to \textbf{false}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \langle c_2, \sigma\rangle}$$

A regra para o comando **while** apenas afirma que podemos extrair de **while** b **do** c um comando **if**. Esta regra não é dirigida a sintaxe: a semântica operacional não define funções, e não é composicional.

$$\texttt{enq} \quad \langle \textbf{while } \ b \ \textbf{ do } \ c, \sigma\rangle \to \langle \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}, \sigma\rangle$$

A seguir temos a semântica estrutural completa de nossa linguagem.

$$\texttt{nul} \quad \langle \textbf{skip}, \sigma\rangle \to \sigma$$

$$\texttt{atr} \quad \langle v := e, \sigma\rangle \to \sigma[v : \ \mathcal{E}[\![e]\!]\sigma]$$

$$\texttt{seq}^1 \quad \frac{\langle c_1, \sigma\rangle \to \langle c_2, \sigma'\rangle}{\langle c_1; c_3, \sigma\rangle \to \langle c_2; c_3, \sigma'\rangle}$$

$$\texttt{seq}^2 \quad \frac{\langle c_1, \sigma\rangle \to \sigma'}{\langle c_1; c_2, \sigma\rangle \to \langle c_2, \sigma'\rangle}$$

$$\texttt{ifT} \quad \frac{\sigma \vdash b \to \textbf{true}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \langle c_1, \sigma\rangle}$$

$$\texttt{ifF} \quad \frac{\sigma \vdash b \to \textbf{false}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \langle c_2, \sigma\rangle}$$

$$\texttt{enq} \quad \langle \textbf{while } \ b \ \textbf{ do } \ c, \sigma\rangle \to \langle \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}, \sigma\rangle$$

Compare as regras de sequenciamento com a usada na semântica natural, onde *toda* expressão de execução do programa sempre leva a um estado:

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma''}$$

Neste exemplo, as três expressões $\ldots \Downarrow \ldots$ levam a estados (σ, σ'' e σ'').

Na semântica estrutural, a regra $\texttt{seq}^1$ presume que $\langle c_1, \sigma\rangle$ leva a uma *configuração* $\langle c_2, \sigma'\rangle$; já a regra $\texttt{seq}^2$ presumo que $\langle c_1, \sigma\rangle$ leva a um *estado final* σ'. Os dois casos são tratados.

Os comandos **if**, por outro lado, *sempre* levam a uma configuração, porque não são finais: sempre resta executar um dos dois "braços" do **if**.

O comando **while** também sempre leva a uma configuração, e ela é exatamente a de um **if**, contendo um **while**.

Definimos para a semântica estrutural, também, uma função semântica.

Definição 5.3 (função semântica estrutural)**.** Definimos a função semântica estrutural como

$$\mathcal{C}_E[\![c]\!]\sigma = \sigma' \quad \Leftrightarrow \quad \langle c, \sigma\rangle \to \sigma'. \qquad \blacklozenge$$

5.3 Expressões

A construção de expressões aritméticas e booleanas é semelhante em estrutura àquela feita com semântica denotacional.

Proposição 5.4. *Existem semânticas naturais $\Downarrow_E, \Downarrow_B$ e estruturais $\to_E, \to_B$ para expressões aritméticas equivalentes a $\mathcal{E}$, ou seja, se e é expressão aritmética e b expressão booleana, então.*

$$\begin{array}{lcccl} \langle e, \sigma\rangle \Downarrow_E v & \Leftrightarrow & \langle e, \sigma\rangle \to_E v & \Leftrightarrow & \mathcal{E}[\![e]\!]\sigma = v \\ \langle b, \sigma\rangle \Downarrow_B w & \Leftrightarrow & \langle b, \sigma\rangle \to_B w & \Leftrightarrow & \mathcal{B}[\![b]\!]\sigma = w \end{array}$$

5.4 Término com `abort`

Uma das maneiras de modelar o término da execução com **abort** é aumentar o conjunto de estados:

$$\hat{\Sigma} = \Sigma \cup \Big(\{\mathbf{err}\} \times \Sigma\Big).$$

Assim, se

$$\begin{array}{l} [], \\ [x:\ 1, y:\ 10], \\ [x:\ 0, y:\ 0, z:\ 1] \end{array}$$

são estados, também serão estados

$$\begin{array}{l} \big(\mathbf{err}, []\big), \\ \big(\mathbf{err}, [x:\ 1, y:\ 10]\big), \\ \big(\mathbf{err}, [x:\ 0, y:\ 0, z:\ 1]\big). \end{array}$$

O comando **abort** deve fazer o programa parar imediatamente no estado em que estiver. A semântica estrutural é modificada com as duas novas regras a seguir.

$$\texttt{seq}^3 \quad \frac{\langle c_1, (x, \sigma)\rangle \to (\mathbf{err}, \sigma')}{\langle c_1; c_2, \sigma\rangle \to (\mathbf{err}, \sigma')}$$

$$\texttt{abo} \quad \langle \mathbf{abort}, \sigma\rangle \to (\mathbf{err}, \sigma)$$

A nova regra para sequenciamento é necessária para terminar imediatamente a execução após o **abort**,

evitando que o segundo comando da composição seja considerado.

$$\texttt{abo} \quad \langle \textbf{abort}, \sigma \rangle \to (\textbf{err}, \sigma)$$

$$\texttt{nul} \quad \langle \textbf{skip}, \sigma \rangle \to \sigma$$

$$\texttt{atr} \quad \langle v := e, \sigma \rangle \to \sigma[v : \ \mathcal{E}[\![e]\!]\sigma]$$

$$\texttt{seq}^1 \quad \frac{\langle c_1, \sigma \rangle \to \langle c_2, \sigma' \rangle}{\langle c_1; c_3, \sigma \rangle \to \langle c_2; c_3, \sigma' \rangle}$$

$$\texttt{seq}^2 \quad \frac{\langle c_1, \sigma \rangle \to \sigma'}{\langle c_1; c_2, \sigma \rangle \to \langle c_2, \sigma' \rangle}$$

$$\texttt{seq}^3 \quad \frac{\langle c_1, (x, \sigma) \rangle \to (\textbf{err}, \sigma')}{\langle c_1; c_2, \sigma \rangle \to (\textbf{err}, \sigma')}$$

$$\texttt{ifT} \quad \frac{\sigma \vdash b \to \textbf{true}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \to \langle c_1, \sigma \rangle}$$

$$\texttt{ifF} \quad \frac{\sigma \vdash b \to \textbf{false}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \to \langle c_2, \sigma \rangle}$$

$$\texttt{enq} \quad \langle \textbf{while } \ b \ \textbf{ do } \ c, \sigma \rangle \to \langle \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}, \sigma \rangle$$

5.5 Blocos e variáveis locais

Voltamos agora a considerar declarações de variáveis para blocos.

$\langle C \rangle ::=$ begin $\langle V \rangle \ \langle C \rangle$ end

$\langle V \rangle ::=$ **var** $\langle I \rangle := \langle E \rangle ; \ \langle V \rangle \mid \epsilon$

Para semântica natural, a regra a seguir permite criar um ambiente com variáveis locais. A regra determina que o ambiente σ é modificado, adicionando a variável n com o valor da expressão e, resultando no ambiente σ'.

$$\texttt{var}^1 \quad \frac{\sigma \vdash e \Downarrow v}{\langle \textbf{var } n := e, \sigma \rangle \Downarrow \sigma[n : \ v]}$$

$$\texttt{var}^2 \quad \frac{\begin{array}{c}\sigma \vdash v_1 \Downarrow_V \sigma' \\ \sigma' \vdash v_2 \Downarrow_V \sigma''\end{array}}{\langle v_1; v_2, \sigma \rangle \Downarrow_V \sigma''}$$

Para *usar* este ambiente, temos a seguinte regra.

$$\texttt{blo} \quad \frac{\begin{array}{c}\sigma \vdash v \Downarrow \sigma' \\ \langle c, \sigma' \rangle \Downarrow \sigma''\end{array}}{\langle \textbf{begin } v \ c \textbf{ end}, \sigma \rangle \Downarrow \sigma''}$$

Informalmente, a regra diz que se a declaração v transforma o estado σ em σ', então podemos executar c em σ'. A definição de variáveis locais com semântica estrutural é mais difícil, e todas as regras teriam que ser modificadas para que o estado atual seja "passado adiante", e modificado quando há declaração de novas variáveis.

5.6 Procedimentos

Relembramos a gramática que definimos para declarar e chamar procedimentos.

$\langle C \rangle ::=$ **call** $\langle I \rangle$

$\langle P \rangle ::=$ **proc** $\langle I \rangle : \langle C \rangle \ ; \langle P \rangle | \ \epsilon$

Usaremos um ambiente de procedimentos, $\mathbf{Env}_P$, que mapeia identificadores em definições de procedimento.

$$\mathbf{Env}_P : \mathbf{Id} \to \mathbf{Proc}$$

Note que não definimos **Proc** como função modificadora de estados, porque na semântica operacional não tratamos trechos de programas como funções. Os elementos de **Proc** podem ser vistos como trechos de programa (literalmente).

$$\texttt{call}^1 \quad \frac{\begin{array}{c}\langle e_i, \sigma \rangle \to_E v_i \\ p(x_1, \ldots, x_k) = c \ \in \ \mathbf{Env}_P \\ \langle c, \sigma[x_1 : v_1, \ \ldots, \ x_k : v_k] \rangle \to \sigma' \end{array}}{\mathbf{Env}_P \vdash p(e_1, \ldots, e_k) \to \sigma'}$$

$$\texttt{call}^2 \quad \frac{\begin{array}{c}\langle e_i, \sigma \rangle \to_E v_i \\ p(x_1, \ldots, x_k) = c \ \in \ \mathbf{Env}_P \\ \langle c, \sigma[x_1 : v_1, \ \ldots, \ x_k : v_k] \rangle \to \langle c', \sigma' \rangle \end{array}}{\mathbf{Env}_P \vdash p(e_1, \ldots, e_k) \to \langle c', \sigma' \rangle}$$

A regra $\texttt{call}^1$ representa chamadas de procedimento que terminam em estado final, e a regra $\texttt{call}^2$ para chamadas que resultam em nova configuração.

Os parâmetros foram passados por valor (foram avaliados antes de serem inseridos como parâmetros). para passagem por nome, a modificação é simples:

$$\texttt{call}^1 \quad \frac{\begin{array}{c} p(x_1, \ldots, x_k) = c \ \in \ \mathbf{Env}_P \\ \langle c, \sigma[x_1 : e_1, \ \ldots, \ x_k : e_k] \rangle \to \sigma' \end{array}}{\mathbf{Env}_P \vdash p(e_1, \ldots, e_k) \to \sigma'}$$

$$\texttt{call}^2 \quad \frac{\begin{array}{c} p(x_1, \ldots, x_k) = c \ \in \ \mathbf{Env}_P \\ \langle c, \sigma[x_1 : e_1, \ \ldots, \ x_k : e_k] \rangle \to \langle c', \sigma' \rangle \end{array}}{\mathbf{Env}_P \vdash p(e_1, \ldots, e_k) \to \langle c', \sigma' \rangle}$$

A definição de procedimentos com semântica natural é mais complexa, e não será abordada.

5.7 Execução não determinística

O comando **either** (c_1, c_2) decide não-deterministicamente executar um dos dois comandos, c_1 ou c_2. Sua gramática abstrata é dada a seguir.

$\langle C \rangle ::=$ **either** $(\langle C \rangle, \langle C \rangle)$

Modelamos facilmente este comando, bastando que a construção seja semelhante ao **if**, exceto que não usamos uma condição booleana para decidir que regra usar – ao construir a árvore de derivação, *qualquer uma* das regras pode ser usada.

Para semântica natural, as regras são dadas a seguir.

$$\texttt{nd}^1 \quad \frac{\langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \textbf{either } (c_1, c_2), \sigma \rangle \Downarrow \sigma'}$$

$$\texttt{nd}^2 \quad \frac{\langle c_2, \sigma \rangle \Downarrow \sigma'}{\langle \textbf{either } (c_1, c_2), \sigma \rangle \Downarrow \sigma'}$$

Em seguida temos as regras pra semântica estrutural.

$$\texttt{nd}^1 \quad \langle \textbf{either } (c_1, c_2), \sigma \rangle \to (c_1, \sigma')$$

$$\texttt{nd}^2 \quad \langle \textbf{either } (c_1, c_2), \sigma \rangle \to (c_2, \sigma')$$

5.8 Execução paralela

Modelamos agora a execução paralela de comandos. Não tratamos dos problemas decorrente do compartilhamento de memória; apenas modelamos o comando **par**, que executa dois outros comandos em paralelo[1].

Estendemos a gramática abstrata, incluindo o comando **par**:

$\langle C \rangle ::= \textbf{par}\ (\langle C \rangle, \langle C \rangle)$

A semântica estrutural é modificada, adicionando regras para este comando. Como esta forma de descrição semântica representa o efeito de cada passo do programa, é fácil descrever o efeito de **par** (c_1, c_2): basta que um passo de um dos dois comandos seja executado.

$$\texttt{par}^1 \quad \frac{\langle c_1, \sigma\rangle \to \langle c_3, \sigma'\rangle}{\langle \textbf{par}\ (c_1, c_2), \sigma\rangle \to \langle \textbf{par}\ (c_3, c_2), \sigma'\rangle}$$

$$\texttt{par}^2 \quad \frac{\langle c_2, \sigma\rangle \to \langle c_3, \sigma'\rangle}{\langle \textbf{par}\ (c_1, c_2), \sigma\rangle \to \langle \textbf{par}\ (c_1, c_3), \sigma'\rangle}$$

$$\texttt{par}^3 \quad \frac{\langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{par}\ (c_1, c_2), \sigma\rangle \to \langle c_2, \sigma'\rangle}$$

$$\texttt{par}^4 \quad \frac{\langle c_2, \sigma\rangle \to \sigma'}{\langle \textbf{par}\ (c_1, c_2), \sigma\rangle \to \langle c_1, \sigma'\rangle}$$

As duas primeiras regras tratam o caso em que o passo executado ainda não é final, resultando em um par (programa, estado). Neste caso, a execução de **par** resulta em um estado diferente, mas ainda com um outro comando **par** para ser executado. As outras duas regras tratam do caso em que o comando executado termina; neste caso, a execução do comando **par** resulta em novo estado, e a configuração tem por executar o comando que não foi executado dentro do **par**.

A descrição da semântica de paralelismo foi fácil porque usamos semântica estrutural. Usando semântica natural, não teríamos como descrever a intercalação de passos de programa.

Exercícios

Ex. 44 — Prove que as duas semânticas operacionais (natural e estrutural) que definimos para nossa linguagem são determinísticas: se $\langle c, \sigma\rangle \to \langle c_1, \sigma_1\rangle$ e $\langle c, \sigma\rangle \to \langle c_2, \sigma_2\rangle$, então $\langle c_1, \sigma_1\rangle = \langle c_2, \sigma_2\rangle$.

Ex. 45 — Na seção 5.5 expusemos duas regras: uma para criar um ambiente com declarações de variáveis, e uma para usá-las. Mostre que as duas podem ser fundidas em uma só.

Ex. 46 — Defina semântica estrutural para variáveis locais, como sugerido no final da seção 5.5.

Ex. 47 — Implemente uma forma de proteção para que blocos não sejam intercalados durante a execução paralela:

$\langle C \rangle ::= \textbf{nopar}\ \langle C \rangle$

Ex. 48 — Modifique a semântica de chamada de procedimentos para

a) Permitir chamadas recursivas;

b) Utilizar escopo dinâmico para variáveis (mas estático para procedimentos);

c) Utilizar escopo dinâmico para variáveis e procedimentos;

d) Permitir ao programador escolher entre escopo estático ou dinâmico ao declarar variáveis e procedimentos.

[1] A notação **par** é usada na linguagem *Occam*.

Capítulo 6

Corretude de Implementação

Neste Capítulo mostramos como a semântica operacional de uma linguagem pode ser usada para demonstrar a corretude de sua implementação.

6.1 Uma CPU hipotética

A arquitetura que consideraremos tem uma memória e uma pilha. Operações aritméticas e lógicas acontecem exclusivamente na pilha, e a única maneira de usar a memória é trazendo valores da memória para o topo da pilha ou levando do topo da pilha para a memória.

- Booleanos são representados como zero (**false**) e diferente de zero (*true*).
- Toda operação lógica ou aritmética lê operandos da pilha, e deixa seu resultado na pilha.
- Comandos que usam a pilha para tomar decisões desempilham o valor que ali estava.
- Há correspondência um-para-um entre rótulos e comandos `JZ`.

`STO`	*var*	move topo para *end*
`PUSH`	*var*	copia (empilha) *end* para topo
`PUSH`	*#val*	empilha valor *#val*
`POP`		desempilha (descarta) topo
`ADD`		operação: soma
`SUB`		operação: subtração
`EQ`		comparação: $a = b$?
`LE`		comparação: $a \leq b$?
`AND`		operação: e
`OR`		operação: ou
`NOT`		operação: negação
`JZ`	L	salto para L se topo= 0; desempilha
`NOP`		nada faz

Nossos programas terão rótulos na memória para que o comando `JZ` possa usar. Durante a execução, interpretamos um rótulo ".L" como uma instrução `NOP`.

Uma configuração de nossa máquina é uma tupla (c, ξ, σ), onde c é um programa; ξ é a pilha; e σ é a memória. Para simplificar a notação, representaremos a memória como função de nomes em valores, sem preocupação com endereços fisicos. Por exemplo, se a pilha é $[3 : 2 : 3]$ e o estado é $[x : 4; y : 5]$, então a execução do trecho a seguir,

```
.L1
      ...
      PUSH x
        EQ
        JZ L1
      PUSH y
       ...
```

na linha `PUSH x`, pode ser representado como segue:

$$\big(\texttt{PUSH}\ x : \ \texttt{EQ} : \ \texttt{JZ}\ L_1, [3:2:3], [x:4; y:5]\big).$$

6.2 Semântica para o *assembly* da arquitetura

Definimos uma semântica operacional para a arquitetura de nossa máquina. A derivação de configurações (ou seja, um passo de execução da máquina abstrata) é denotada $k_1 \gg k_2$.

$$(\texttt{STO}\ v : c, k : \xi, \sigma) \gg (c : \xi, \sigma[v : k])$$
$$(\texttt{PUSH}\ v, \xi, \sigma) \gg (c : \sigma(v) : \xi, \sigma)$$
$$(\texttt{PUSH}\ \#k, \xi, \sigma) \gg (c : k : \xi, \sigma)$$
$$(\texttt{POP} : c, x : \xi, \sigma) \gg (c, \xi, \sigma)$$
$$(\texttt{ADD} : c, x : y : \xi, \sigma) \gg (c, (x + y) : \xi, \sigma)$$
$$(\texttt{SUB} : c, x : y : \xi, \sigma) \gg (c, (x - y) : \xi, \sigma)$$
$$(\texttt{EQ} : c, x : y : \xi, \sigma) \gg (c, (x?y) : \xi, \sigma)$$
$$(\texttt{AND} : c, x : y : \xi, \sigma) \gg (c, (x \wedge y) : \xi, \sigma)$$
$$(\texttt{OR} : c, x : y : \xi, \sigma) \gg (c, (x \vee y) : \xi, \sigma)$$
$$(\texttt{NOT} : c, x : \xi, \sigma) \gg (c, (\neg x) : \xi, \sigma)$$
$$(\texttt{JZ}\ L : c_1 : c_2 : \cdots : .L : c, k : \xi, \sigma) \gg (c, \xi, \sigma) \qquad (k = 0)$$
$$(\texttt{JZ}\ L : c_1 : c_2 : \cdots : .L : c, k : \xi, \sigma) \gg (c_1 : c_2 : \cdots : c, \xi, \sigma) \qquad (k \neq 0)$$
$$\frac{(c_1 : c_2 : ..., \xi, \sigma) \gg (c', k : \xi', \sigma')}{(.L : c_1 : c_2 : \cdots : \texttt{JZ}\ L : c, \xi, \sigma) \gg (c_1 : c_2 : \cdots : c)} \qquad (k \neq 0)$$
$$\frac{(c_1 : c_2 : ..., \xi, \sigma) \gg (c', k : \xi', \sigma')}{(.L : c_1 : c_2 : \cdots : \texttt{JZ}\ L : c, \xi, \sigma) \gg (c_1 : c_2 : \cdots : .L : c_1 : c_2 : \cdots : \texttt{JZ}\ L : c, \xi, \sigma)} \qquad (k = 0)$$
$$(\texttt{NOP}, \xi, \sigma) \gg (\epsilon, \xi, \sigma)$$

Definiremos agora uma função semântica

$$\mathcal{M} : \mathbf{Asm} \to (\mathbf{Sta} \times \mathbf{Sto})_\perp \to (\mathbf{Sta} \times \mathbf{Sto})_\perp$$

$$\mathcal{M}[\![c]\!]\sigma = \begin{cases} \sigma' & \text{se } (c, \cdot, \sigma) \gg (\cdot, \xi, \sigma') \\ \perp & \text{se } (c, \cdot, \sigma) \text{ diverge} \end{cases}$$

6.3 Tradução da linguagem para o *assembly*

Definimos funções de tradução para expressões aritméticas, booleanas e para código.

$$\mathcal{TE} : \mathbf{Aexp} \to \mathbf{Asm}$$
$$\mathcal{TB} : \mathbf{Bexp} \to \mathbf{Asm}$$
$$\mathcal{TC} : \mathbf{Cmd} \to \mathbf{Asm}$$

A função $\mathcal{TE}$ é simples: para deixar uma constante k no topo da pilha, basta executar `PUSH` $\#k$. Para deixar uma *variável* v, há a instrução `PUSH` v. Para realizar uma operação (soma ou subtração) em uma das expressões, executamos o código para cada uma – depois disso os resultados das duas estarão no topo da pilha, e podemos usar `ADD` ou `SUB` para efetuar a operação.

$$\begin{aligned}
\mathcal{TE}[\![k]\!] &= \texttt{PUSH}\ \#k \\
\mathcal{TE}[\![v]\!] &= \texttt{PUSH}\ v \\
\mathcal{TE}[\![e_1 + e_2]\!] &= \mathcal{TE}[\![e_1]\!] : \mathcal{TE}[\![e_2]\!] : \texttt{ADD} \\
\mathcal{TE}[\![e_1 - e_2]\!] &= \mathcal{TE}[\![e_1]\!] : \mathcal{TE}[\![e_2]\!] : \texttt{SUB}
\end{aligned}$$

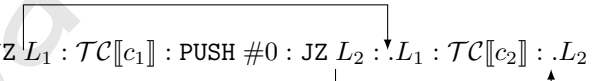
A função $\mathcal{TB}$ é semelhante; usamos `PUSH` $\#0$ e `PUSH` $\#1$ para as constantes **false** e **true**. As operações lógicas são realizadas de maneira semelhante à função $\mathcal{TE}$.

$$\begin{aligned}
\mathcal{TB}[\![\textbf{true}]\!] &= \texttt{PUSH}\ \#1 \\
\mathcal{TB}[\![\textbf{false}]\!] &= \texttt{PUSH}\ \#0 \\
\mathcal{TB}[\![b_1\ \textbf{and}\ b_2]\!] &= \mathcal{TB}[\![b_1]\!] : \mathcal{TB}[\![b_2]\!] : \texttt{AND} \\
\mathcal{TB}[\![b_1\ \textbf{or}\ b_2]\!] &= \mathcal{TB}[\![b_1]\!] : \mathcal{TB}[\![b_2]\!] : \texttt{OR} \\
\mathcal{TB}[\![\textbf{not}\ b]\!] &= \mathcal{TB}[\![b]\!] : \texttt{NOT} \\
\mathcal{TB}[\![e_1 = e_2]\!] &= \mathcal{TE}[\![e_1]\!] : \mathcal{TE}[\![e_2]\!] : \texttt{EQ} \\
\mathcal{TB}[\![e_1 \leq e_2]\!] &= \mathcal{TE}[\![e_1]\!] : \mathcal{TE}[\![e_2]\!] : \texttt{LE} \\
\mathcal{TB}[\![e_1 \geq e_2]\!] &= \mathcal{TB}[\![e_1 = e_2]\!] : \mathcal{TB}[\![e_1 \leq e_2]\!] : \texttt{NOT} : \texttt{OR}
\end{aligned}$$

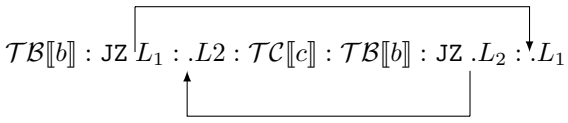
A função $\mathcal{TC}$ não traduz comandos em sequências de instruções.

$$\begin{aligned}
\mathcal{TC}[\![\textbf{skip}]\!] &= \texttt{NOP} \\
\mathcal{TC}[\![x := e]\!] &= \mathcal{TE}[\![e]\!] : \texttt{STO}\ x \\
\mathcal{TC}[\![c_1; c_2]\!] &= \mathcal{TC}[\![c_1]\!] : \mathcal{TC}[\![c_2]\!] \\
\mathcal{TC}[\![\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2]\!] &= \mathcal{TB}[\![b]\!] : \texttt{JZ}\ L_1 : \mathcal{TC}[\![c_1]\!] : \texttt{PUSH}\ \#0 : \texttt{JZ}\ L_2 : .L_1 : \mathcal{TC}[\![c_2]\!] : .L_2 \\
\mathcal{TC}[\![\textbf{while}\ b\ \textbf{do}\ c]\!] &= \mathcal{TB}[\![b]\!] : \texttt{JZ}\ L_1 : .L2 : \mathcal{TC}[\![c]\!] : \mathcal{TB}[\![b]\!] : \texttt{JZ}\ .L_2 : .L_1
\end{aligned}$$

O diagrama a seguir ilustra a tradução de **if** b **then** c_1 **else** c_2 para a linguagem da máquina. Note que como não temos desvios incondicionais, usamos "`PUSH` $\#0$: `JZ` L_2" para desviar para L_2.



O próximo diagrama ilustra a tradução de **while** b **do** c:



6.4 Corretude

Definimos uma função semântica para a tradução da linguagem de alto nível para o *assembly*.

$$\mathcal{C}_M = M \circ \mathcal{TC}$$

Queremos mostrar que $\mathcal{C}_M$ é equivalente a $\mathcal{C}_E$ (a função semântica implícita na semântica estrutural da linguagem – definição 5.3).

Definição 6.1 (relação de bissimulação)**.** A *relação de bissimulação*, $\approx$, entre configurações da linguagem de alto nível e a linguagem assembly é dada por

$$\begin{aligned} \langle c, \sigma\rangle &\approx (\mathcal{TC}[\![c]\!], \cdot, \sigma) \\ \sigma &\approx (\cdot, \cdot, \sigma). \end{aligned}$$

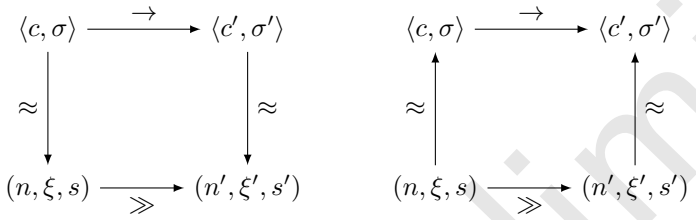
♦

A seguir provamos que a tradução $\mathcal{TC}$ está correta, usando a técnica da bissimulação.

Teorema 6.2 (corretude de tradução por bissimulação)**.** *A tradução $\mathcal{TC}$ está correta, ou seja,*

i) se $\langle c, \sigma\rangle \approx (n, \cdot, s)$ e $\langle c, \sigma\rangle \to^* \langle c', \sigma'\rangle$, então existe $(n', \cdot, s')$ tal que $(n, \cdot, s) \gg (n', \cdot, s')$ e $\langle c', \sigma'\rangle \approx (n', \cdot, s')$ – diagrama (a);

ii) se $\langle c, \sigma\rangle \approx (n, \cdot, s)$ e $(n, \cdot, s) \gg (n', \cdot, s')$, então existe $(n', \cdot, s')$ tal que $\langle c, \sigma\rangle \to^* \langle c', \sigma'\rangle$ e $\langle c', \sigma'\rangle \approx (n', \cdot, s')$ – diagrama (b).



(a) (b)

Exercícios

Ex. 49 — Embora não seja semântica denotacional, a semântica operacional da função de tradução de expressões aritméticas é *composicional*. Já a função de tradução de expressões booleanas, $\mathcal{TB}$, não é. Eplique porque, e mostre como torná-la composicional.

Ex. 50 — Inclua os comandos **repeat** .. **until** e **abort** na linguagem, faça a tradução deles a demonstração de corretude da tradução.

Ex. 51 — Modifique a máquina virtual para que use um contador de programa.

Capítulo 7

Semântica Axiomática

A *semântica axiomática* de um programa nos permitirá expressar e provar propriedades da forma

$$\{P\}c\{Q\},$$

onde c é um trecho de programa e P e Q são asserções a respeito do estado. As idéias básicas da semântica axiomática foram desenvolvidas inicialmente por Floyd [Flo67] e Hoare [Hoa69].

Usaremos neste Capítulo um pouco do Cálculo de Predicados. Introduções ao assunto são dadas nos livros de Hedman [Hed04], de Enderton [End01], de Mendelson [Men15] e de Leary [Lea00].

7.1 A linguagem das asserções

Ë necessário determinar com mais rigor o que são as asserções: qual sua sintaxe e sua semântica. Com isto também determinaremos sobre o que elas podem versar.

As asserções em **Asser** são predicados, onde usamos tanto variáveis do programa como metavariáveis. A sintaxe é dada a seguir. Relembramos que os símbolos B e I denotam expressões booleanas e identificadores em nossa sintaxe abstrata.

$\langle asser \rangle$ ::= **true** | **false**
| $\langle B \rangle$
| $\forall$ I $\langle B \rangle$
| $\exists$ I $\langle B \rangle$

Exemplo 7.1. As seguir temos algumas possíveis asserções, gramaticamente corretas.

$$n <= k$$
$$3 <= 4$$
$$\forall n \;\; n + k = p - n$$
$$\exists k \;\; \forall j \;\; k * j = k$$
$$-n <= 0 \;\wedge\; n = k + 1$$
$$n <= k \;\Rightarrow p <= q + 1$$

◀

Definição 7.2 (asserção de corretude parcial)**.** Uma *asserção de corretude parcial* é

$$\{P\}c\{Q\},$$

onde $P, Q \in$ **Asser** e $c \in$ **Cmd**. ♦

Quando uma asserção P é válida em um estado σ, denotamos

$$\sigma \models P.$$

O estado indefinido satisfaz qualquer asserção, ou seja,

$$\bot \models P, \quad \forall P$$

7.2 Lógica de Hoare

O sistema de prova proposto por Hoare para a linguagem imperativa minimalista é mostrado a seguir. Cada regra de inferência permite provar a corretude de um dos comandos isoladamente.

$$\texttt{skip} \quad \{P\}\ \textbf{skip}\ \{P\}$$

$$\texttt{atr} \quad \{P[x \to A[k]]\}x := k\{P\}$$

$$\texttt{seq} \quad \frac{\{P\}c_1\{Q\},\ \ \{Q\}c_2\{R\}}{\{P\}c_1; c_2\{R\}}$$

$$\texttt{if} \quad \frac{\{B[b] \wedge P\}c_1\{Q\},\ \ \{\neg B[b] \wedge P\}c_2\{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\{Q\}}$$

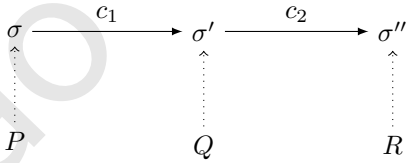
$$\texttt{enq} \quad \frac{\{B[b] \wedge P\}c\{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ c\{\neg B[b] \wedge P\}}$$

$$\texttt{cons} \quad \frac{\{P'\}c\{Q'\}}{\{P\}c\{Q\}} \qquad (\text{se } P \Rightarrow P',\ \ Q' \Rightarrow Q)$$

A regra `skip` simplesmente determina que se P vale em um determinado estado, continua valendo após a execução de `skip` – o que é evidente.

A regra `atr` declara que se P vale em um estado σ, substituindo x por $A[k]$, então P valerá no estado que resulta depois da execução de $x := k$.

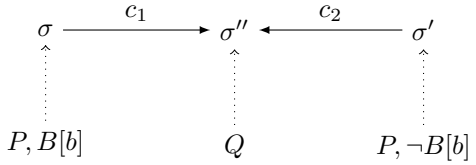
A regra `seq` é também bastante simples: se um trecho c_1 leva do estado σ a σ', e outro comando c_2 leva do estado σ' em σ'', e valem $P(\sigma)$, $Q(\sigma')$, $R(\sigma'')$, então para o programa composto $c_1; c_2$ vale $\{P\}\ c_1; c_2\ \{R\}$.



A regra `if` expressa que se

- c_1 leva de σ em σ'';
- c_2 leva de σ' em σ'';
- valem $P(\sigma)$, $P(\sigma')$, $Q(\sigma'')$;
- $B[b]$ vale em σ, mas é falso em σ',

então vale $\{P\}$ **if** b **then** c_1 **else** c_2 $\{P\}$.



A regra `enq` determina que se o comando c não invalida a proposição P, então "**while** b **do** c" também preserva a propriedade P. Mais ainda, se $B[b]$ vale antes do laço, $\neg B[b]$ valerá depois.

A regra `cons` expressa o efeito da relação de consequência lógica entre proposições. Sua justificativa é pedida no exercício 52.

7.2.1 Inferência

As inferências que podemos realizar neste sistema de prova sào asserções de corretude a respeito de trechos de programas.

Definição 7.3 (inferência de asserção)**.** Se α e β são asserções, e $\alpha \Rightarrow \cdots \Rightarrow \beta$, dizemos que é possível *inferir β a partir de α*, e denotamos

$$\alpha \vdash_H \beta.$$

♦

Exemplo 7.4. Considere o seguinte trecho de programa.

```
i := 1
p := x
while n ≠ i do
    p := p*x
    i := i + 1
```

Provaremos que, se $n \geq 1$ antes de começar este trecho, então depois de sua execução teremos $n = i$ e $p = x^n$.

A asserção que traduz o que queremos provar é esta:

$$\{n \geq 1\} i := 1;\ p := x;\ \textbf{while}\ n \neq i\ \textbf{do}\ \big(p := px;\ i := i + 1\big)\{p = x^n\}$$

Primeiro, identificamos a invariante de laço que queremos. A invariante, sendo uma afirmação sobre as variáveis do programa, depende do estado, portanto denotamos $I(\sigma)$:

$$I(\sigma) = \big(1 \leq \sigma(i) \leq \sigma(n) \wedge \sigma(p) = \sigma(x)^{\sigma(i)}\big)$$

É inconveniente, no entanto, usar o símbolo σ tantas vezes na expressão, e também desnecessário, já que é o único estado sendo tratado. Simplificando a notaçào, e deixando σ implícito, temos

$$I = \big(1 \leq i \leq n \wedge p = x^i\big)$$

Os passos da demonstração são mostrados a seguir.

$$\begin{aligned}
&(1) \vdash_H \{I[i : i+1]\} i := i + 1 \{I\} && (\texttt{atr})\\
&(2) \vdash_H \{I[i : i+1][p : px]\} p := p * x \{I[i : i+1]\} && (\texttt{atr})\\
&(3) \vdash_H \{I[i : i+1][p : px]\} p := p * x; i := i + 1 \{I\} && ((2), (1),\ \texttt{seq})\\
&(4) (i \neq n \wedge I) \Rightarrow (I[i : i+1])[p : px]\\
&(5) \vdash_H \{i \neq n \wedge I\} p := p * x; i := i + 1 \{I\} && ((3), (4),\ \texttt{cons})\\
&(6) \vdash_H \{I\}\ \textbf{while}\ i \neq n\ \textbf{do}\ \big(p := p * x; i := i + 1\big)\{i = n \wedge I\} && ((5),\ \texttt{while})\\
&(7) \vdash_H \{I[i : 1]\} i := 1 \{I\} && (\texttt{atr})\\
&(8) \vdash_H \{I[i : 1][p : x]\} i := 1 \{I[i : 1]\} && (\texttt{atr})\\
&(9) \vdash_H \{I[i : 1][p : x]\} i := 1; p := x \{I\} && ((8), (7)\ \texttt{seq})\\
&(10) \vdash_H \{n \geq 1\} i := 1; p := x; \textbf{while}\ i \neq n\ \textbf{do}\ \Big(p := p * x; i := i + 1\Big)\{i = n \wedge I\} && ((9), (6)\ \texttt{seq})
\end{aligned}$$

◄

Definição 7.5 (árvore de inferência)**.** Podemos encadear as triplas $\{P\}c\{Q\}$ usadas em uma demonstração de corretude; teremos como resultado uma *árvore de inferência*. ♦

Exemplo 7.6. Mostramos a árvore de prova do exemplo 7.4. Como a árvore é muito grande, mostramos duas subárvores separadamente, e as unimos com duas arestas.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_H \{I[i:1][p:x]\}i:=1;p:=x\{I[i:1]\}}}{\vdash_H \{I[i:1]\}i:=1\{I\}}\,\text{seq}}{\vdash_H \{I[i:1][p:x]\}i:=1;p:=x\{I\}}}{\vdash_H \{I[i:i+1][p:px]\}p:=p*x\{I[i:i+1]\}}}{\vdash_H \{I[i:i+1]\}i:=i+1\{I\}}\,\text{seq}}{\vdash_H \{I[i:i+1][p:px]\}p:=p*x;i:=i+1\{I\}}\,\text{cons}}{\vdash_H \{i\neq n\wedge I\}p:=p*x;i:=i+1\{I\}}\,\text{enq}}{\vdash_H \{I\}\ \textbf{while}\ i\neq n\ \textbf{do}\ \big(p:=p*x;i:=i+1\big)\{i=n\wedge I\}}$$

$$\dfrac{\dfrac{}{\vdash_H \{I[i:1][p:x]\}i:=1;p:=x\{I\}}\qquad \vdash_H \{I\}\ \textbf{while}\ i\neq n\ \textbf{do}\ \big(p:=p*x;i:=i+1\big)\{i=n\wedge I\}}{\vdash_H \{n\geq 1\}i:=1;p:=x;\textbf{while}\ i\neq n\ \textbf{do}\ \Big(p:=p*x;i:=i+1\Big)\{i=n\wedge I\}}\,\text{seq}$$

As folhas da árvore são evidentemente axiomas – nesta árvore, os axiomas são todos regras to tipo `atr`. Observe que na regra `cons` usamos o fato $(i\neq n\wedge I)\Rightarrow(i[i:i+1])[p:px]$. ◀

7.3 Equivalência de programas

Definição 7.7. Dois trechos de programa c_1 e c_2 são equivalentes de acordo com a semântica axiomática definida pela Lógica de Hoare se e somente se para todas as proposições P, Q,

$$\Big(\vdash \{P\}c_1\{Q\}\Big) \Leftrightarrow \Big(\vdash \{P\}c_2\{Q\}\Big).$$ ♦

Exemplo 7.8. ◀

7.4 Consistência e completude

Demonstramos agora que a Lógica de Hoare, apresentada na seção 7.2, é consistente, ou seja, que todas as asserções que podemos derivar com as regras dadas são válidas.

Nossa definição de validade, no entanto, será relativa: conceitualmente, uma asserção é válida se expressa um efeito que sempre será observado em uma semântica – aqui, escolhemos a semântica natural.

Definição 7.9 (validade de asserção de corretude)**.** Seja $\{P\}c\{Q\}$ uma asserção. Se, para todo estado σ,

- $P(\sigma)$ vale;
- $\langle c,\sigma\rangle \to \sigma'$;
- $Q(\sigma')$ vale;

então $\{P\}c\{Q\}$ é *válida*, e denotamos $\models \{P\}c\{Q\}$. ♦

Dizemos que um sistema de prova é *consistente* se as inferências que fazemos usando suas regras levam a fórmulas válidas – ou seja, se é possível deduzir β a partir de α ($\alpha \vdash \beta$), então sempre que α for verdade, β também será ($\alpha \models \beta$). Analogamente, o sistema de prova é *completo* se toda fórmula válida pode ser deduzida – ou seja, se a validade de α implica na de β ($\alpha \models \beta$), então é possível inferir β com as regras do sistema, a partir de α ($\alpha \vdash \beta$).

Definição 7.10 (consistência da Lógica de Hoare)**.** Um sistema de prova com asserções é *consistente* se, para toda asserção de corretude parcial $\{P\}c\{Q\}$, $\vdash \{P\}c\{Q\}$ implica em $\models \{P\}c\{Q\}$. ♦

Definição 7.11 (completude da Lógica de Hoare)**.** Um sistema de prova com asserções é *completo* se, para toda asserção de corretude parcial $\{P\}c\{Q\}$, $\models \{P\}c\{Q\}$ implica em $\vdash \{P\}c\{Q\}$. ♦

7.4.1 Consistência

Damos uma demonstração parcial da consistência da Lógica de Hoare para nossa linguagem. O exercício 55 pede a elaboração do restante da demonstração.

Teorema 7.12 (consistência da lógica de Hoare)**.** *A Lógica de Hoare para nossa linguagem é consistente.*

Demonstração. A demonstração é por indução estrutural.

A base de indução se dá com as regras `skip` e `atr`, que são axiomas da Lógica. Faremos aqui a base e um dos passos; os outros são pedidos no exercício 55

- `skip`: nada há a provar, já que trivialmente, se uma proposição P era verdade antes do **skip**, e este não modifica o estado, a proposição continuará sendo verdade depois.
- `atr`: a pré-condição é $P[x : \mathcal{E}[\![k]\!]]$, logo a proposição P deve valer com x valendo $\mathcal{E}[\![k]\!]$. Como o comando executado apenas modifica x, tornando-o igual a $\mathcal{E}[\![k]\!]$, então P continua sendo válida.

Há agora quatro passos a demonstrar: `seq`, `if`, `enq`, `cons`.

- `if`: Presumimos que

$$\begin{aligned} (i) \quad & \models \{\mathcal{B}[\![b]\!] \wedge P\} c_1 \{Q\}, \\ (ii) \quad & \models \{\neg\mathcal{B}[\![b]\!] \wedge P\} c_2 \{Q\}. \end{aligned}$$

  Agora consideramos dois estados, σ e σ', para os quais P vale, e tais que

$$\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \sigma'.$$

  Quando $\mathcal{B}[\![b]\!] = \textbf{true}$, teremos $\mathcal{B}[\![b]\!] \wedge P$ (i), e

$$\langle c_1, \sigma\rangle \to \sigma',$$

  e portanto, como presumimos (i), temos $Q(\sigma') = \textbf{true}$.

  Para $\mathcal{B}[\![b]\!] = \textbf{false}$, o raciocínio é simétrico. □

7.4.2 Completude (abordagem extensional)

Usaremos aqui a mesma abordagem de Nielson e Nielson [NN07]: embora seja impossível demonstrar a completude da Lógica de Hoare usando a linguagem de asserções que propusemos, a demonstração é possível se mudarmos a forma de descrever as asserções. Nesta seção, trataremos as asserções como proposições (predicados sem argumentos), e desta forma poderemos demonstrar a completude do sistema de provas.

Definição 7.13 (pré-condição liberal mais fraca)**.** Dados um comando $c \in \textbf{Cmd}$ e uma asserção Q, definimos a *pré-condiçào liberal mais fraca* para c e Q como a asserção, denotada $\text{wlp}(c, Q)$, tal que

$$\text{wlp}(c, Q) = \textbf{true}$$

se e somete se para todo σ' tal que

$$\langle c, \sigma\rangle \to \sigma',$$

temos $Q(\sigma') = \textbf{true}$. ♦

Lema 7.14. *Para todo comando $c \in$ **Cmd** e toda asserção Q,*

i) $\models \{\text{wlp}(c, Q)\}c\{Q\}$

ii) se $\models \{P\}c\{Q\}$*, então* $P \Rightarrow \text{wlp}(c, Q)$

Lema 7.15. *Se, para todo $c \in$ **Cmd** e Q, tivermos*

$$\vdash_H \{\text{wlp}(c, Q)\}c\{Q\}, \tag{7.1}$$

então a Lógica de Hoare é completa no sentido extensional.

Demonstração. Suponha que $\models \{P\}c\{Q\}$. O Lema 7.14 nos garante então que

$$P \Rightarrow \text{wlp}(c, Q). \tag{7.2}$$

Assim, com 7.1 e 7.2 podemos usar a regra `cons` para obter

$$\vdash_H \{P\}c\{Q\}. \qquad \square$$

Teorema 7.16. *A Lógica de Hoare que apresentamos é completa no sentido extensional.*

Demonstração. Por indução estrutural **(esta demonstração está incompleta)** $\square$

7.4.3 Incompletude (abordagem intensional)

7.5 Corretude total

Até agora tratamos de asserções de corretude parcial, porque admitimos que trechos de programa podem entrar em *loop* e nunca parar. Nesta seção discutiremos asserções de *corretude total*, quando damos também a garantia de parada para o programa.

Denotamos por

$$[P]\ c\ [Q]$$

a asserção equivalente a "$\{P\}c\{Q\}$", com a garantia adicional de que o trecho c sempre para (ou seja, não resulta em estado indefinido).

Exercícios

Ex. 52 — Justifique a regra `cons` da Lógica de Hoare.

Ex. 53 — O máximo divisor comum entre dois inteiros, $\text{mdc}(a, b)$ é definido como segue.

$$\text{mdc}(a, 0) = a$$
$$\text{mdc}(a, b) = \text{mdc}\Big(b, a \pmod b\Big)$$

Escreva um programa **while** que calcule o mdc de dois números, e prove que ele está correto de acordo com a definição de mdc.

Ex. 54 — Proponha pré e pós-condições parao seguinte programa, e demonstre sua corretude usando a Lógica de Hoarre.

```
q := 0
r:= N
while r >= D do
    r := r -D
    q := q + 1
```

Ex. 55 — Prove os casos faltando do Teorema 7.12.

Capítulo 8

λ-Cálculo

Assim como máquinas de Turing, o λ-Cálculo foi proposto por Alonzo Church como uma maneira de formalizar a noção de "computação". Para a teoria de Linguagens de Programação, a importância maior do λ-Cálculo está principalmente no fato de muitas das características de linguagens serem mais facilmente expressadas nele do que em linguagens reais.

O λ-Cálculo é um formalismo para manipulação simbólica de funções. Embora seja simples descrever funções que operam sobre elementos de conjuntos, a notação usual não é adequada para expressar funções que não apenas usam o resultado de outras via composição de funções, mas funções aceitam outras como argumento (funções *de alta ordem*). Por exemplo, podemos escrever a função que soma 3, $f(x) = x + 3$, sem maiores problemas. Mas a função que recebe um número n e uma função g e soma n com $g(n + 1)$ seria $f(n, g) = n + g(n + 1)$, mas a notação começa a ficar confusa (olhando para $g(n + 1)$, podemos imaginar que g é um número multiplicado por $n + 1$). O λ-cálculo permite expressar estas funções de maneira natural.

Uma curta introdução ao λ-Cálculo é dada no livro de Chris Hankin [Han04]; O livro de J. Hindley e J. Seldin [HS08] é uma introdução mais extensa; O livro de Henk Barendregt [Bar12] contém uma exposição exaustiva do λ-Cálculo e de suas muitas variantes.

8.1 Sintaxe

A sintaxe abstrata do λ-Cálculo é dada a seguir.

$\langle termo \rangle ::= \langle var \rangle \mid \langle termo \rangle \ \langle termo \rangle \mid \lambda \ \langle var \rangle \ \cdot \langle termo \rangle$

Denotamos por Λ o conjunto de todos os termos do λ-Cálculo.

A sintaxe concreta do λ-Cálculo exige parênteses ao redor de termos compostos (os que não são variáveis isoladas).

Exemplo 8.1. São λ-termos x, xy, $(\lambda x \cdot y)$, $((xy)z)$ e $(x(yz))$. ◀

A gramática dada descreve as possíveis formas de λ-termos, que podem ser combinadas:

- **Átomo:** uma variável isolada. Por exemplo, x;
- **Abstração:** $(\lambda x \cdot m)$. Uma *abstração* é a descrição de uma transformação sintática em λ-termos. $(\lambda x \cdot m)$ significa "troque x por m";
- **Aplicação:** (mn). Uma aplicação descreve onde abstrações devem ser usadas.

No λ-Cálculo normalmente os parênteses são usados somente quando necessário. As regras a seguir são usadas para simplificar a notação.

- Aplicações associam à esquerda: $wxyz$ é o mesmo que $(wx)yz$, $((wx)y)z$ e $(((wx)y)z)$, que são *diferentes* de $w(xy)z$, $wx(yz)$ e $(w(x(yz))$;

- O corpo de uma abstração estende-se à direita tando quanto possível: $\lambda x \cdot wxyz$ é o mesmo que $(\lambda x \cdot wxyz)$.
- Abreviamos sequências de abstrações, omitindo os λs desnecessários: o termo $\lambda x \cdot \lambda y \cdot \lambda z \cdot xxyyzz$ é abreviado $\lambda xyz \cdot xxyyzz$.

8.1.1 Variáveis livres e ligadas

Uma variável em um λ-termo pode estar livre ou ligada. Por exemplo, em $(\lambda x \cdot x + 2)(4 * y)$, a variável x está ligada (porque é usada na abstração) e y está livre. Se denotarmos por $VLG(l)$ o conjunto de variáveis ligadas de um λ-termo l, então VLG pode ser definido indutivamente:

$$\begin{aligned} VLG(x) &= \emptyset \\ VLG(\lambda x \cdot M) &= VLG(M) \cup \{x\} \\ VLG(MN) &= VLG(M) \cup VLG(N) \end{aligned}$$

Podemos então denotar o conjunto das variáveis livres de um λ-termo por $VLV(M)$. Note que $VLV(M)$ não é o complemento de VLG(M), uma vez que a mesma variável pode ser tanto livre como ligada em um λ-termo – por exemplo, $(\lambda x \cdot 2x)(abx)$.

$$\begin{aligned} VLV(x) &= \{x\} \\ VLV(\lambda x \cdot M) &= VLV(M) \setminus \{x\} \\ VLV(MN) &= VLV(M) \cup VLV(N) \end{aligned}$$

8.1.2 α-equivalência

Os termos do λ-Cálculo são usualmente agrupados em classes de equivalência: sabemos que, por exemplo, $\lambda x \cdot xz$ é o mesmo que $\lambda y \cdot yz$.

Definição 8.2 (α-equivalência)**.** Dois termos t_1 e t_2 são α-equivalentes se um pode ser obtido do outro pela substituição de variáveis não-livres. Denotamos $t_1 \equiv_\alpha t_2$. ♦

Note que α-equivalência é uma relação de equivalência, e que consequentemente particiona os termos.

Exemplo 8.3. Damos exemplos de alguns termos α-equivalentes.

$$\begin{aligned} &(\lambda x \cdot x) \equiv_\alpha (\lambda y \cdot y) \equiv_\alpha (\lambda z \cdot z) \\ &(\lambda x \cdot xy) \equiv_\alpha (\lambda z \cdot zy) \\ &(\lambda x \cdot xy)x \equiv_\alpha (\lambda z \cdot zy)x && (x \text{ fora dos parênteses é livre!}) \\ &(\lambda x \cdot y)x \equiv_\alpha (\lambda z \cdot y)x && (x \text{ fora dos parênteses é livre!}) \end{aligned}$$

◄

8.2 Semântica, β-redução

Denotamos $M \rightarrowtail N$ quando pudermos transformar, sintaticamente, M em N. Uma semântica operacional para o λ-Cálculo é dada aseguir.

Tomamos como base a idéia de aplicação de uma função a um argumento: se temos $(\lambda x \cdot m)\ n$, então

- $(\lambda x \cdot m)$ é uma função, e
- n é um argumento.

A aplicação da função no argumento n deve então ser dada pela regra

$$(\lambda x \cdot m)\ n \rightarrowtail m[n/x].$$

Esta regra é chamada de β-redução.

Definição 8.4 (β-redex)**.** Um β-redex em um termo é um subtermo da forma $(\lambda x \cdot m)n$. ⧫

A seguir temos uma semântica operacional para o λ-Cálculo – uma semântica estrutural simples, *onde as configurações são termos do λ-Cálculo.* Além da regra de β-redução, há outras regras que tratam dos casos em que a β-redução pode ser usada em parte de um termo.

$$\beta \quad (\lambda x \cdot m)n \rightarrowtail m[n/x] \qquad \lambda \quad \frac{m \rightarrowtail m'}{\lambda x \cdot m \rightarrowtail \lambda x \cdot m'}$$

$$\texttt{apl}^1 \quad \frac{t \rightarrowtail t'}{(tm) \rightarrowtail (t'm)} \qquad \texttt{apl}^2 \quad \frac{m \rightarrowtail m'}{(tm) \rightarrowtail (tm')}$$

Dizemos que a regra β-redução, aplicada uma vez em um β-redex de um termo é uma *contração*. Uma sequência de aplicações é uma *redução*.

8.3 Formas normais, e computação com λ-Cálculo

Podemos aplicar β-reduções repetidamente em um termo somente enquando ele contiver β-redexes. Quando não houver mais, chegamos ao final da sequência de redução.

Definição 8.5 (Forma Normal)**.** Um λ-termo está em sua *forma normal* se não pode ser reduzido, ou seja, se não contém β-redexes. Dizemos que M é a *forma normal de N* se $N \rightarrowtail^* M$ e M é forma normal. ⧫

Exemplo 8.6. A seguir temos uma redução. Em cada linha, o β-redex que será utilizado é sublinhado.

$$\begin{aligned}(\lambda x \cdot xyx)(\lambda x \cdot xy)\underline{((\lambda x \cdot xx)z)} &\rightarrowtail \underline{(\lambda x \cdot xyx)(\lambda x \cdot xy)}(zz)\\ &\rightarrowtail \underline{(\lambda x \cdot xy)y}(\lambda x \cdot xy)(zz)\\ &\rightarrowtail yy\underline{(\lambda x \cdot xy)(zz)}\\ &\rightarrowtail yy(zz)y\end{aligned}$$

Como $yyzzy$ não tem β-redexes como subtermos, ele é a forma normal do termo inicial (e dos outros intermediários, evidentemente!) ◀

Encontrar a forma normal de um λ-termo é equivalente a determinar o valor de uma função recursiva parcial ou a verificar o resultado de computação de uma máquina de Turing.

8.3.1 Termos sem forma normal

Há termos que não tem forma normal. A seguir temos um exemplo: tentaremos reduzir o termo $\Omega = (\lambda x \cdot xx)(\lambda x \cdot xx)$

$$\begin{aligned}(\lambda x \cdot xx)(\lambda x \cdot xx) &\rightarrowtail (\lambda x \cdot xx)(\lambda x \cdot xx)\\ &\rightarrowtail (\lambda x \cdot xx)(\lambda x \cdot xx)\\ &\rightarrowtail (\lambda x \cdot xx)(\lambda x \cdot xx)\\ &\vdots\end{aligned}$$

Dizemos que este termo *diverge*, e denotamos $\Omega \uparrow$.

A divergência no λ-Cálculo é semelhante a programas que não param em linguagens imperativas: se um programa p equivale a um λ-termo e, então

$$e \uparrow \iff [\![p]\!] = \bot.$$

8.3.2 Não-determinismo

Pode haver mais de uma possibilidade de redução para o mesmo termo, como ilustrado a seguir.

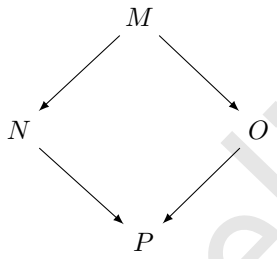
$$(\lambda x \cdot xy)\underline{((\lambda w \cdot w)z)} \rightarrowtail (\lambda x \cdot xy)z$$
$$\underline{(\lambda x \cdot xy)((\lambda w \cdot w)z)} \rightarrowtail (\lambda w \cdot w)zy$$

A regra aplicada nos dois casos foi a mesma, mas a parte do termo em que aplicamos a regra é diferente.

8.3.3 Confluência

O *Teorema de Church-Rosser* afirma que se é possível reduzir um λ-termo a duas diferentes formas, então estas duas podem ser reduzidas a uma terceira. Esta propriedade do λ-cálculo é chamada de *confluência*.

Teorema 8.7 (Church-Rosser)**.** *Seja M um λ-termo. Se M é redutível a N e M também é redutível a $O \neq N$, então existe P tal que $N \rightarrowtail^* P$ e $O \rightarrowtail^* P$.*

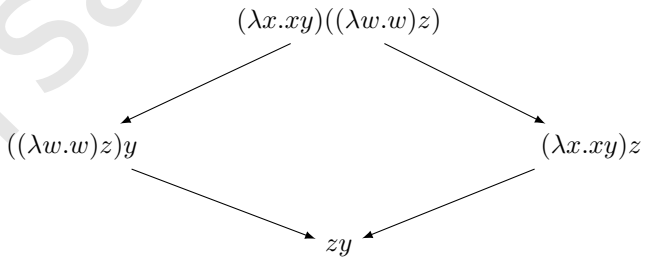


Como corolário do teorema de Church-Rosser temos a unicidade de formas normais.

Corolário 8.8. *Se um λ-termo M tem forma normal, ela é única modulo $\equiv_\alpha$, ou seja, se M tem formas normais P e Q, então $P \equiv_\alpha Q$. então $N \equiv_\alpha O$.*

Exemplo 8.9. Abaixo temos duas reduções, seguidas por um diagrama que ilustra a propriedade da confluência.

$$(\lambda x.xy)\underline{((\lambda w.w)z)} \rightarrowtail (\lambda x.xy)z \rightarrowtail zy$$
$$\underline{(\lambda x.xy)((\lambda w.w)z)} \rightarrowtail ((\lambda w.w)z)y \rightarrowtail zy$$



◀

8.4 Ordem de avaliação

Um λ-termo pode ter vários β-redexes, e podemos portanto aplicar β-reduções em diferentes ordens. Destacamos aqui duas ordens relevantes – a ordem *aplicativa* e a ordem *normal*.

Definição 8.10 (Ordem aplicativa)**.** Uma redução se dá na *ordem aplicativa* se o β-redex contraído em cada passo é, dentre os mais internos, o mais à esquerda. ♦

Exemplo 8.11. Reduziremos o λ-termo $(\lambda xy \cdot yxy)((\lambda x \cdot zx)w)w$. Primeiro reescrevemos o termo sem abreviar os λs:

$$(\lambda x \cdot (\lambda y \cdot yxy))\ (\underline{(\lambda x \cdot zx)w})w$$

Agora começamos aredução.

$$\begin{aligned}(\lambda x \cdot (\lambda y \cdot yxy))\ (\underline{(\lambda x \cdot zx)w})w &\rightarrowtail \underline{(\lambda x \cdot (\lambda y \cdot yxy))\ (zw)}w \\ &\rightarrowtail \underline{(\lambda y \cdot y(zw)y)}w \\ &\rightarrowtail w(zw)w.\end{aligned}$$

A forma normal deste termo é, portanto, $w(zw)w$. ◀

Definição 8.12 (Ordem normal)**.** Uma redução se dá na *ordem normal* se o β-redex contraído em cada passo é, dentre os mais externos, o mais à esquerda. ♦

Exemplo 8.13. Reduziremos o mesmo termo do exemplo 8.11, usando ordem normal.

$$\begin{aligned}\underline{(\lambda x \cdot (\lambda y \cdot yxy))\ ((\lambda x \cdot zx)w)}w &\rightarrowtail \underline{(\lambda y \cdot y\ ((\lambda x \cdot zx)w)\ y)w} \\ &\rightarrowtail w\ \underline{((\lambda x \cdot zx)w)}\ w) \\ &\rightarrowtail w(zw)w.\end{aligned}$$

Chegamos (como esperávamos) ao mesmo termo ao qual chegamos no exemplo 8.11. ◀

As duas ordens de aplicação podem exigir mais ou menos passos; mas usando a ordem normal, temos a garantia de que, se existe uma forma normal para o termo, ela será encontrada. Isto não vale para a ordem aplicativa.

Teorema 8.14 (da padronização)**.** *Se alguma sequência de reduções começando com um termo t termina, então a sequência de reduções em ordem normal começando com t termina.*

Exemplo 8.15. Tentaremos reduzir o mesmo termo, $(\lambda xy \cdot x)xw((\lambda x \cdot xx)(\lambda x \cdot xx))$, usando as duas ordens diferentes. Como a abstração à esquerda tem dois parâmetros, primeiro a traduzimos para o λ-Cálculo puro:

$$(\lambda xy \cdot x)z((\lambda x \cdot xx)(\lambda x \cdot xx)) = (\lambda x \cdot (\lambda y \cdot x))z((\lambda x \cdot xx)(\lambda x \cdot xx))$$

Agora executamos reduções usando a ordem normal:

$$\begin{aligned}\underline{(\lambda x \cdot (\lambda y \cdot x))}z((\lambda x \cdot xx)(\lambda x \cdot xx)) &\rightarrowtail \underline{(\lambda y \cdot z)((\lambda x \cdot xx)(\lambda x \cdot xx))} \\ &\rightarrowtail z\end{aligned}$$

Facilmente chegamos a uma forma normal, z.

A seguir temos uma tentativa de reduzir o mesmo termo, usando ordem aplicativa.

$$\begin{aligned}(\lambda x \cdot (\lambda y \cdot x))z\underline{((\lambda x \cdot xx)(\lambda x \cdot xx))} &\rightarrowtail (\lambda x \cdot (\lambda y \cdot x))z\underline{((\lambda x \cdot xx)(\lambda x \cdot xx))} \\ &\rightarrowtail (\lambda x \cdot (\lambda y \cdot x))z\underline{((\lambda x \cdot xx)(\lambda x \cdot xx))} \\ &\vdots\end{aligned}$$

Note que, ao tentarmos avaliar o segundo argumento, $\Omega = (\lambda x \cdot xx)(\lambda x \cdot xx)$, seguimos realizando a mesma redução indefinidamente. ◀

8.5 Combinadores

Usar o λ-Cálculo fica mais fácil quando definimos λ-termos que realizam operações interessantes – e damos então nomes a esses termos.

Definição 8.16 (combinador)**.** Um *combinador* é um λ-termo sem variáveis livres. ♦

Exemplo 8.17. Os termos $(\lambda x \cdot x)$, $(\lambda x \cdot xx)$, $(\lambda xy \cdot yy)$ são combinadores.
Já x, $(\lambda x \cdot xy)$, (xy) e $(\lambda x \cdot z)$ não são combinadores, porque todos tem variáveis livres. ◀

A seguir mencionamos alguns combinadores relevantes.
O combinador mais simples que existe é o combinador identidade, $id = \lambda x \cdot x$.
Já encontramos antes o combinador $\Omega = (\lambda x \cdot xx)(\lambda x \cdot xx)$, que não tem forma normal.
Podemos definir dois combinadores projeção:

$$L = \lambda xy \cdot x$$
$$R = \lambda xy \cdot y$$

Também é interessante o combinador *self*, que transforma x em (xx):

$$\text{self} = \lambda x \cdot (xx).$$

Note que $\Omega = (\text{self self})$.

O combinador *flip* toma uma função binária, dois argumentos x e y, e aplica a função aos argumentos, *na ordem trocada*:

$$\text{flip} = \lambda f \cdot \lambda x \cdot \lambda y \cdot (f\ yx)$$

O combinador *apply* toma uma função, seus argumentos, e aplica a função a eles.

$$\text{apply} = \lambda f x_1 x_2 \ldots x_n \cdot (f\ x_1 x_2 \ldots x_n).$$

Podemos usar *apply* para realizar *aplicação parcial*: se $g = (\lambda xyz \cdot M)$ é uma função de três argumentos, então $(\text{apply}\ gw)$ é uma função de dois argumentos:

$$\begin{aligned}(\text{apply}\, gw) &= (\lambda fx \cdot (fx))(\lambda xyzM)w \\ &= (\lambda xyzM)w \\ &= \lambda yz \cdot M[w/x].\end{aligned}$$

8.6 Iteração e o combinador Y

Como no λ-cálculo há apenas uma operação (β-redução), pode parecer que ele não é Turing-equivalente por não permitir a aplicação de uma determinada computação um número de vezes. No entanto, é possível escrever λ-termos equivalentes a laços *for* e *while*.

Há duas funções que precisaremos para construir iteradores em λ-cálculo: uma função que testa se seu argumento é zero e outra que calcula o sucessor de um número. Não mostraremos aqui a construção delas.

$$\text{zero?}\ \ n\ \ x\ \ y = \begin{cases} x & \text{se } n = 0 \\ y & \text{se } n \neq 0. \end{cases}$$

A definição acima mostra que a função zero recebe n, x e y, e depois retorna x ou y, dependendo do valor de n.

A iteração em λ-cálculo não é simples como em máquinas de Turing ou em funções recursivas. Usando as funções *zero?* e *pred* (que retorna o predecessor de um número natural), poderíamos construir um iterador que funciona como laço *for*:

$$\begin{aligned} I = (\lambda\ n\ f\ x \cdot \text{zero?}\ n \quad & x \\ & (I \quad (\text{pred}\ n)\ f\ (f\ x))). \end{aligned}$$

Ou seja: I toma três argumentos, n, f e x. Se $n = 0$, retorna x; senão, retorna I aplicado a $n-1, f, i$ e x.

Mas estaríamos usando I em sua própria definição, e se tentarmos reduzir um λ-termo construído com I, notaremos que as reduções apenas aumentam o tamanho do termo!

Seja S a função geradora do iterador que queremos – ou seja, a função que recebe uma função M e aplica o raciocínio já descrito para I em M:

$$S = (\lambda\ M \cdot (\lambda\ n\ f\ x \cdot \text{zero?}\ n \quad x \quad (M \quad (\text{pred}\ n)\ f\ (f\ x))))$$

Esta definição não é circular. Agora precisamos de um termo que possamos passar para S, de forma que S nos retorne exatamente o iterador que queremos. Ou seja, queremos I tal que

$$I = SI,$$

porque assim, já tendo definido S, poderemos usar SI como iterador. Um operador desta forma é chamado de ponto fixo para S:

Definição 8.18 (Ponto fixo)**.** Um λ-termo F é um ponto fixo para um termo M se $F = MF$ ⧫

É sempre possível encontrar o ponto fixo para qualquer termo.

Definição 8.19 (Combinador de ponto fixo)**.** Um combinador de ponto fixo é um λ-termo que transforma algum outro λ-termo M em um ponto fixo. ⧫

Teorema 8.20 (do ponto fixo)**.** *Para todo $f \in \Lambda$, existe um $x \in \Lambda$ tal que $fx = x$.*

Demonstração. Seja $w = \lambda x \cdot f(xx)$, com $x = ww$. Então

$$x = ww = (\lambda x \cdot f(xx))w = f(ww) = fx. \qquad \square$$

O combinador de ponto fixo usado na demonstração é chamado de *combinador Y*:

$$Y = \lambda\ f\ \cdot (\lambda\ x\ \cdot\ f\ (x\ x))\ (\lambda\ x\ \cdot\ f\ (x\ x))$$

Este combinador foi descoberto por Haskell Curry[1].

Podemos agora escrever um λ-termo que funciona como um laço do tipo "enquanto" para encontrar o menor valor para o qual uma função vale zero[2]:

$$Z = \lambda\ f\ n\ \cdot\ \text{zero?}\ (f\ n)\ n\ (Z\ f\ (\text{suc}\ n)).$$

O termo acima define a função Z como aquela que implementa: "se $f(n) = 0$ retorne n, senão tente Z novamente em $n+1$". Agora usamos este λ-termo para construir outro, que

$$L = \lambda\ M\ \cdot\ (\lambda\ f\ n\ \cdot\ \text{zero?}\ (f\ n)\ n\ (M\ f\ (\text{suc}\ n))).$$

O operador de minimização então é:

$$Z = YL.$$

8.7 Programação em λ-Cálculo

Mostramos brevemente nesta seção como usar o λ-Cálculo para realizar algumas operações lógicas e aritméticas. Isto mostra como este sistema é equivalente a Máquinas de Turing e linguagens de programação. Para isto codificaremos números e booleanos em λ-termos, usando a chamada *codificação de Church*.

[1] Há outros combinadores de ponto fixo, inclusive um descoberto por Alan Turing: $Y = (\lambda\ x\ y\ \cdot\ (x\ x\ y))\ (\lambda\ x\ y\ \cdot\ (x\ x\ y))$.

[2] Semelhante ao operador μ para funções recursivas parciais

8.7.1 Números naturais

Os axiomas de Peano para a aritmética são dados a seguir. Quando não houver quantificação, deve-se entender $\forall a, b, c \in \mathbb{N}$. Denotamos por $s(n)$ o sucessor de n.

1. O zero é um número natural.
2. (reflexividade de $=$) $a = a$.
3. (simetria de $=$) $a = b \implies b = a$.
4. (transitividade de $=$) se $a = b$ e $b = c$ então $a = c$.
5. (fechamento) se $a \in \mathbb{N}$ e $a = b$, então $b \in \mathbb{N}$.
6. $s(a) \in \mathbb{N}$.
7. $a = b$ se e somente se $s(a) = s(b)$.
8. não há $n \in \mathbb{N}$ tal que $s(n) = 0$.
9. (indução finita) Para todo conjunto X de naturais, se $0 \in X$, e se $n \in X$ implica em $s(n) \in X$, então $X = \mathbb{N}$.

John von Neumann mostrou que é possível descrever estes axiomas usando a teoria de conjuntos de Zermelo-Fraenkel. A construção indutiva é

$$\begin{aligned} 0 &= \emptyset, \\ s(x) &= x \cup \{x\}. \end{aligned}$$

Concretamente, temos

$$\begin{aligned} 0 &= \emptyset \\ 1 &= s(0) = \emptyset \cup \{\ \emptyset\ \} = \{\ \emptyset\ \} \\ 2 &= s(1) = \{\ \emptyset\ \} \cup \{\ \{\emptyset\}\ \} = \{\ \emptyset, \{\emptyset\}\ \} \\ 3 &= s(2) = \{\ \emptyset, \{\emptyset\}\ \} \cup \{\ \{\emptyset, \{\emptyset\}\}\ \} = \{\ \emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\ \} \\ &\vdots \end{aligned}$$

Alonzo Church observou que é possível fazer algo semelhante usando funções no λ-Cálculo. Todo número na codificação de Church é da forma

$$\lambda f \cdot \lambda x \cdot \ \ldots,$$

O número de vezes que f é aplicada sobre x é o número representado. Assim, o zero é exatamente x (sem que f seja aplicada):

$$0 = \lambda f \cdot \lambda x \cdot x.$$

Para representar o número n, aplicamos a função f n vezes:

$$\begin{aligned} n &= \lambda f \cdot \lambda x \cdot \overbrace{(f(f \cdots (fx) \cdots))}^{n \text{ aplicações de } f} \\ &= \lambda f \cdot \lambda x \cdot (f^n\ x). \end{aligned}$$

No entanto, esta definição não nos serve, porque é circular (n é usado no lado direito em sua própria definição). Há, no entanto, uma codificação da função sucessor que nos será útil:

$$s = \lambda n \cdot \lambda f \cdot \lambda x \cdot s(n\ f\ x)$$

A seguir verificamos que o combinador s, quando aplicado em zero, resulta em um

$$\begin{aligned} s(0) &= \lambda n \cdot \lambda f \cdot \lambda x \cdot f(n\ f\ x)0 \\ &= \underline{\lambda n \cdot \lambda f \cdot \lambda x \cdot f(n\ f\ x)(\lambda f \cdot \lambda x \cdot x)} \\ &\rightarrowtail \lambda f \cdot \lambda x \cdot f(\underline{(\lambda f \cdot \lambda x \cdot x)\ f\ x}) \\ &\rightarrowtail \lambda f \cdot \lambda x \cdot (fx) \\ &= 1. \end{aligned}$$

Podemos dar o nome de z ao combinador zero e s ao combinador sucessor.

$$\begin{aligned} z &= \lambda s \cdot (\lambda x \cdot x) \\ s &= \lambda n \cdot \lambda f \cdot \lambda x \cdot f(n\ f\ x) \end{aligned}$$

A função z sempre retorna a identidade $(\lambda x \cdot x)$. A função s recebe um número n e retorna uma função que itera s n vezes sobre um argumento. Assim, temos

$$\begin{aligned} z &= \lambda f \cdot id \\ s &= \lambda n \cdot \lambda f \cdot \lambda x \cdot f^n(x) \end{aligned}$$

Isto nos dá:

$$\begin{aligned} 0 &:= z \\ 1 &:= sz \\ 2 &:= ssz \\ &\vdots \\ n &:= (s)^n z \end{aligned}$$

A soma e multiplicação são codificadas da seguinte forma:

$$\begin{aligned} m + n &:= \lambda mnfx \cdot mf(nfx) \\ m \times n &:= \lambda mnf \cdot m(nf)x \end{aligned}$$

Exemplo 8.21. Neste exemplo realizamos a soma $1 + 2$ usando a codificação de Church. As reduções são feitas na ordem normal.

$$\begin{aligned} 1 + 2 &= (sz) + (ssz) \\ &= (\lambda fx \cdot (fx)) + (\lambda fx \cdot (f(fx))) \\ &= \underline{(\lambda mnfx \cdot mf(nfx))}\ \overbrace{(\lambda fx \cdot (fx))}^{1}\ \overbrace{(\lambda fx \cdot (f(fx)))}^{2} \\ &\rightarrowtail \underline{(\lambda nfx \cdot (\lambda fx \cdot (fx))f(nfx))\ (\lambda fx \cdot (f(fx)))} \\ &\rightarrowtail (\lambda fx \cdot \underline{(\lambda fx \cdot (fx))f}((\lambda fx \cdot (f(fx)))fx)) \\ &\rightarrowtail (\lambda fx \cdot f(\underline{(\lambda fx \cdot (f(fx)))fx})) \\ &\rightarrowtail (\lambda fx \cdot f(f(fx))) \\ &= 3. \end{aligned}$$

◀

Assim como definimos um operador s que produz o sucessor de um número na codificação de Church, podemos definir um combinador p, que produz seu predecessor.

$$p = \lambda n \cdot \lambda f \lambda x \cdot n(\lambda g \cdot \lambda h \cdot h(gf))(\lambda w \cdot x)(\lambda w \cdot w)$$

A subtração portanto é

$$\text{sub}(m,n) := \lambda m \cdot \lambda n \cdot npm.$$

Finalmente, o operador *zero?* determina se um número é zero:

$$\text{zero?} := \lambda n \cdot n(\lambda x \cdot F)T.$$

8.7.2 Booleanos

Uma das maneiras de representar booleanos é mostrada a seguir.

$$\textbf{true} := \lambda xy \cdot x$$
$$\textbf{false} := \lambda xy \cdot y$$

As operações lógicas *"e"* e *"ou"* podem ser codificadas como os seguintes termos:

$$b_1 \wedge b_2 := (\lambda ab \cdot bab) b_1 b_2$$
$$\neg b := (\lambda a \cdot a\, \textbf{false}\, \textbf{true}) b$$

A validade desta codificação pode ser facilmente verificada construindo as tabelas-verdade para *"e"* e *"ou"*.

8.7.3 Controle

Finalmente, podemos codificar algo semelhante ao comando **if** de maneira surpreendentemente simples:

$$\textbf{if}\ b\ \textbf{then}\ t_1\ \textbf{else}\ t_2 := (\lambda x \cdot x)\ b\ t_1\ t_2.$$

A seguir mostramos a redução desta codificação quando $b = \textbf{true}$. Para $b = \textbf{false}$, o raciocínio é semelhante.

$$\begin{aligned}\textbf{if true then}\ t_1\ \textbf{else}\ t_2 &:= (\lambda x \cdot x)\textbf{true}\, t_1 t_2 \\ &:= (\lambda x \cdot x)(\lambda xy.x) t_1 t_2 \\ &\rightarrowtail\ (\lambda x \cdot x) t_1 \\ &\rightarrowtail\ t_1\end{aligned}$$

8.8 Problemas indecidíveis

Sendo o λ-cálculo equivalente em poder computacional às máquinas de Turing e às funções recursivas parciais, é evidente que os problemas que são indecidíveis usando estes outros formalismos também são indecidíveis no λ-cálculo. Alguns deles são:

- A e B tem a mesma forma normal? (Equivalente a decidir se duas máquinas de Turing darão a mesma resposta se alimentadas com a mesma entrada)
- A tem forma normal? (Equivalente ao problema da parada)

Notas

Nos anos 20, David Hilbert propôs que se buscasse uma formalização da Matemática que se baseasse em um conjunto finito de axiomas e que fosse completo e consistente. Esta proposta é conhecida como o "programa de Hilbert", que incluía a decidibilidade da Matemática: a identificação de procedimentos efetivos para decidir se proposições a respeito da Matemática são verdadeiras ou falsas.

Em 1931, Kurt Gödel mostrou que nenhuma teoria que represente a aritmética pode provar sua própria consistência. Anos mais tarde surgiu a formalização da noção de "procedimento efetivo" e vários matemáticos

mostraram, cada um usando uma formalização diferente do conceito de algoritmo (mas todos equivalentes), que há questões – algumas muito simples – que não podem ser respondidas por qualquer algoritmo. Apesar destes resultados negativos, a formalização do conceito de algoritmo levou ao desenvolvimento da Teoria da Computação. Um desdes formalismos é o λ-cálculo. Apesar de máquinas de Turing serem o padrão para a exposição da teoria de Computabilidade, o λ-Cálculo é mais útil na descrição teórica de linguagens de programação.

Exercícios

Ex. 56 — Prove que se $M \rightarrowtail^* N$, então $VLV(N) \subseteq VLV(M)$.

Ex. 57 — Descreva uma semântica *natural* para o λ-Cálculo, e prove que ela é equivalente à semântica estrutural dada neste Capítulo.

Ex. 58 — Ache a forma normal dos termos (é mais fácil se você der nomes a subtermos):

- $(\lambda e \cdot eue)(abc)$
- $(((\lambda x \cdot (\lambda y \cdot axa))q)r)$
- $(\lambda a \cdot (\lambda x \cdot xaux))(\lambda y \cdot yy)$

Ex. 59 — Apresente um termo que aumenta de tamanho sempre que uma β-redução é aplicada nele.

Ex. 60 — Prove o equivalente da indecidibilidade para o λ-cálculo, sem usar equivalência com outros formalismos.

Ex. 61 — Descreva uma gramática que gere infinitos λ-termos sem forma normal. Prove que *toda* palavra gerada por sua gramática é um λ-termo sem forma normal.

Ex. 62 — Mostre que o λ-cálculo pode gerar linguagens livres de contexto.

Ex. 63 — Codificamos o *"e"* lógico como $b_1 \wedge b_2 := \lambda ab \cdot bab$. Encontre outra codificação.

Ex. 64 — Mostre como codificar os operadores lógicos "ou" ($\vee$) e "ou exclusivo" ($\oplus$) no λ-Cálculo.

Ex. 65 — Mostre como codificaar a operação de exponenciação na codificação de Church.

Ex. 66 — Tente escrever o algoritmo de Euclides usando o λ-Cálculo com a codificação de Church.

Ex. 67 — Usamos YG para encontrar o ponto fixo de G (Y é o combinador Y). Agora, tente encontrar *duas* funções para as quais Y é ponto fixo (determine duas funções F_1 e F_2 tais que $F_i Y = Y$).

Capítulo 9

Tipos

No λ-Cálculo original, não havia a preocupação com que valores cada variável poderia assumir, porque como já mencionamos, o λ-Cálculo é um mecanismo sintático. No entanto, é comum tanto em Matemática como em linguagens de programação que sejam especificados os tipos dos objetos com que se trabalha. O λ-Cálculo simplesmente tipado, chamado de $\lambda^{\rightarrow}$, é uma versão do λ-Cálculo visto no Capítulo anterior onde termos tem tipos.

9.1 Tipos: conceito e sintaxe

No λ-Cálculo tipado, atribuímos tipos a variáveis, e como consequência, determinamos tipos de termos (porque são todos construídos indutivamente a partir de variáveis). Denotamos por $a : \tau$ a afirmação de que a é do tipo τ.

Como conceitualmente só temos variáveis e funções, supomos que há um conjunto finito de tipos primitivos, e definimos tipos de funções sobre estes conjuntos:

i Todo tipo atômico é um tipo;

ii Se a e b são tipos, $(a \rightarrow b)$ é um tipo, e é chamado de *tipo função*. Então $(a \rightarrow b)$ é o tipo de todas as funções $f : a \rightarrow b$.

Por exemplo, a seguir temos alguns tipos.

- $\mathbb{N}$ (tipo atômico)
- $(\mathbb{N} \rightarrow \mathbb{N})$
- $(\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N}))$
- $((\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N})$
- $((\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N}))$

Adotamos a convenção de associar tipos *à direita* (e isto está de acordo com a associação de aplicação de função à esquerda).

A gramática de $\lambda^{\rightarrow}$ é:

$\langle M \rangle ::= \langle V \rangle \mid \lambda \langle I \rangle{:}\langle T \rangle \mid \langle M \rangle \langle M \rangle$

$\langle T \rangle ::= \langle P \rangle \mid \langle T \rangle \rightarrow \langle T \rangle$

Temos portanto variáveis, sem declaração de tipos; abstrações, onde declaramos o tipo do argumento; e

aplicações, onde não declaramos tipo.

A semântica do $\lambda^{\rightarrow}$-Cálculo consiste das quatro regras a seguir. A única diferença explícita está na regra de β-redução, que só é aplicável quando o argumento é do tipo especificado pela abstração.

$$\beta \quad (\lambda x : \sigma \cdot m)n \rightarrowtail m[n/x] \qquad \lambda \quad \frac{m \rightarrowtail m'}{\lambda x \cdot m \rightarrowtail \lambda x \cdot m'}$$

$$\texttt{apl}^1 \quad \frac{t \rightarrowtail t'}{(tm) \rightarrowtail (t'm)} \qquad \texttt{apl}^2 \quad \frac{m \rightarrowtail m'}{(tm) \rightarrowtail (tm')}$$

As três últimas regras somente tratam de redução em diferentes contextos. Formalizamos, portanto, a noção de contexto.

As regras para ordem aplicativa são dadas a seguir. Definimos contextos de avaliação como

$\langle C \rangle$ ::= [] | C M | vC

onde v é um *valor*.

$$\beta \quad (\lambda x : \sigma \cdot M)v \rightarrowtail M[v/x] \qquad \texttt{cont} \quad \frac{M \rightarrowtail N}{C[M] \rightarrowtail C[N]}$$

Para ordem normal,

$\langle C \rangle$ ::= [] | C M

$$\beta \quad (\lambda x : \sigma \cdot M)N \rightarrowtail M[N/x] \qquad \texttt{cont} \quad \frac{M \rightarrowtail N}{C[M] \rightarrowtail C[N]}$$

A seguir tratamos de contextos de tipos.

Definição 9.1 (contexto de tipos)**.** Um *contexto de tipos* (ou *ambiente de tipos*) Γ é uma função parcial de variáveis em tipos. Denotamos por $\Gamma(x)$ o tipo da variável x no contexto Γ, e por $\mathrm{dom}(\Gamma)$ o conjunto de variáveis com tipos definidos por Γ. O contexto vazio é denotado por $\emptyset$. ⧫

Um contexto de tipos somente determina tipos de variáveis. Definimos também regras para *julgamento de tipos*, que nos permitem determinar os tipos de termos em geral.

$$\texttt{var} \quad \Gamma \vdash x : \Gamma(x) \qquad \lambda \quad \frac{\Gamma, x : \tau_1, \quad \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \cdot M : \tau_1 \rightarrow \tau_2} \qquad \texttt{apl} \quad \frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma : MN : \tau_2}$$

Definição 9.2 (termo bem-tipado)**.** Um termo é bem-tipado se admite um tipo no ambiente vazio. Em outras palavras,

- Um λ-termo $x : \sigma$ tem tipo σ e é bem-tipado
- Se o tipo de x é σ e o de M é τ, então $\lambda\, x : \sigma \cdot M$ é bem-tipado e tem tipo $\sigma \rightarrow \tau$.
- Se o tipo de M é $\sigma \rightarrow \tau$ e o de N é σ então MN é bem-tipado e tem tipo τ. ⧫

As regras de tipagem permitem *definir* claramente os tipos dos λ-termos; no entanto, queremos poder *inferir* os tipos de subtermos a partir do tipo de um termo. Isto é possível usando o Lema da inversão de regras de tipagem, que essencialmente enuncia que estas regras podem ser invertidas.

Lema 9.3 (inversão de regras de tipagem)**.** *Se $\Gamma \vdash M : \tau$, então*

- *se M é variável, existe $x \in \mathrm{dom}(\Gamma)$, tal que $\Gamma(x) = \tau$;*
- *se M é aplicação $(M_1 M_2)$, então $\Gamma \vdash M_1 : \tau_2 \rightarrow \tau$, e $\Gamma \vdash M_2 : \tau_2$, para algum τ_2;*
- *se M é abstração $\lambda x : \tau_0 \cdot M_1$, então τ é da forma $\tau_0 \rightarrow \tau_1$, e $\Gamma . x : \tau_0 \vdash M_1 : \tau_1$.*

9.2 Tipagem intrínseca e extrínseca (Church × Curry)

No estilo de Church, cada termo tem um tipo *intrínseco*, e não há termos como $\lambda x \cdot x$. Ao invés disso, temos para um termo para cada possível atribuição de tipos: $\lambda x : \sigma \cdot \sigma$, por exemplo.

Já no estilo de Curry, "$\lambda x \cdot x$" denota infinitas funções, com tipos diferentes.

9.3 Remoção de tipos

Definição 9.4 (remoção de tipos)**.** A *remoção de tipo* de termos é definida como segue.

$$\begin{aligned} \lceil x \rceil &= x \\ \lceil \lambda x : \tau : M \rceil &= \lambda x \cdot \lceil M \rceil \\ \lceil M\, N \rceil &= \lceil M \rceil \, \lceil N \rceil \end{aligned}$$ ♦

Uma semântica para $\lambda^{\rightarrow}$ é *não-tipada* (ou "*removedora de tipos*[1]" se toda derivação que ela define é isomorfa a uma derivação no λ-Cálculo. Os dois Lemas a seguir formalizam esta idéia.

Lema 9.5 (simulação direta)**.** *Se $M \rightarrowtail N$ então $\lceil M \rceil \rightarrowtail \lceil N \rceil$.*

Lema 9.6 (simulação inversa)**.** *Se $\lceil M \rceil \rightarrowtail w$ então existe M' tal que $M \rightarrowtail M'$ e $\lceil M' \rceil = w$.*

9.4 Consistência

A consistência do sistema de tipos determina que termos bem-tipados devem divergir ou convergir para um valor. Os dois Teoremas a seguir, juntos, implicam na consistência do sistema de tipos que apresentamos.

Teorema 9.7 (redução de sujeito)**.** *A redução preserva tipos: se $M \rightarrowtail N$, então para todo tipo σ tal que $\emptyset \vdash M : \sigma$, também temos $\emptyset \vdash N : \sigma$.*

Teorema 9.8 (progresso)**.** *Um termo fechado e bem-tipado sempre é redutível ou é um valor: se $\emptyset \vdash M : \sigma$, então ou M é valor ou existe N tal que $M \rightarrowtail N$.*

Demonstração. A sintaxe nos permite escrever variáveis, aplicações e abstrações.

i) variáveis não são termos fechados, e não precisam ser consideradas;

ii) abstrações são, por definição, valores, e portanto vale o enunciado;

iii) aplicações: presumimos, como no enunciado, que $\emptyset \vdash M : \sigma$. Como M é abstração, escrevemos $M = M_1 M_2$. Pelo Lema da inversão das regras de tipagem existem dipos σ_1 e σ_2 tais que $\emptyset \vdash M_1 : \sigma_2 \rightarrow \sigma_1$ e $\emptyset \vdash M_2 : \sigma_2$. Pela hipótese de indução, M_1 é redutível ou é um valor. Se M_1 é redutível, então M também é, porque $[]M_2$ é contexto de avaliação. Se M_1 é valor, e é do tipo $\sigma_2 \rightarrow \sigma_1$, então $M_1 M_2$ é β-redex, e o termo M pode ser reduzido. □

Embora tenhamos demonstrado a consistência do λ-Cálculo tipado, o mesmo não acontece com linguagens de programação reais.

9.5 Novos tipos

Até agora usamos apenas tipos primitivos genéricos. É interessante, no entanto, considerar alguns tipos primitivos epsecíficos, como unidade e booleanos, e tipos compostos.

[1] "Type-erasing" em Inglês.

9.5.1 Unidade

O tipo unitário é usado quando uma expressão não tem valor possível. Por exemplo, a expressão

```
x := if false then 10
```

não tem valor definido, porque o segundo braço do **if** foi omitido. Neste caso, podemos determinar que a semântica de nossa linguagem atribua o valor unitário, que representamos por (), a x.

Incluímos **unit** entre os tipos, e a constante () na gramática.

$\langle V \rangle ::= ()$

$\langle T \rangle ::= \textbf{unit}$

incluímos também uma regra para tipagem.

$$\Gamma \vdash () : \textbf{unit}$$

Podemos usar o tipo unitário agora para, por exemplo, atribuí-lo a toda variável não inicializada.

$$\mathcal{E}[\![v]\!]\sigma = \begin{cases} \sigma(v) & \text{se } \sigma(v) \text{ é definido} \\ () & \text{caso contrário} \end{cases}$$

9.5.2 Booleanos

Incluímos os valores **true** e **false**, e o operador **if** . **then** . **else**.

$\langle M \rangle ::= \textbf{true} \mid \textbf{false} \mid \textbf{if } M \textbf{ then } M \textbf{ else } M$

$\langle T \rangle ::= \textbf{bool}$

Adicionamos um novo contexto de avaliação:

$\langle E \rangle ::= \textbf{if } [\,] \textbf{ then } M \textbf{ else } M$

E duas regras de redução:

$$\textbf{if true then } M_1 \textbf{ else } M_2 \rightarrowtail M_1$$
$$\textbf{if false then } M_1 \textbf{ else } M_2 \rightarrowtail M_2$$

Para inferência de tipos, temos a seguinte regra.

$$\Gamma \vdash \textbf{true} : \textbf{bool} \quad \Gamma \vdash \textbf{false} : \textbf{bool} \quad \frac{\Gamma \vdash M : \textbf{bool} \;\; \Gamma \vdash N : \tau \;\; \Gamma \vdash O : \tau}{\Gamma \vdash (\textbf{if } M \textbf{ then } N \textbf{ else } O) : \tau}$$

Em Common Lisp, apenas o elemento do tipo unitário () é considerado falso; todos os outros valores, inclusive T, são verdadeiros. Em C, booleanos são representados como inteiros, e apenas o valor zero é considerado falso.

9.5.3 Produto

Para representar estruturas podemos usar o produto cartesiano dos conjuntos de elementos de diferentes tipos; se há os tipos τ e σ, e temos elementos $x : \tau$, $y : \sigma$, então o tipo $\tau \times \sigma$ é o tipo prpduto, e o par ordenado $\langle x, y \rangle$ é do tipo $\tau \times \sigma$.

Definiremos também os operadores de projeção π_1 e π_2, que extraem o primeiro e o segundo elemento de

um par. Se $\langle x, y\rangle$ é do tipo $\tau \times \sigma$, então

$$\pi_1 \langle x, y\rangle = x : \tau,$$
$$\pi_2 \langle x, y\rangle = y : \sigma.$$

Nossa gramática é aumentada para incluir a construção de pares com o operador $\langle \cdot, \cdot \rangle$ e os operadores de projeção.

$\langle M\rangle ::= \langle M, N\rangle \mid \pi_1 \langle M\rangle \mid \pi_2 \langle M\rangle$

$\langle T\rangle ::= \langle T\rangle \times \langle T\rangle$

As regras de tipagem para tipos produto são mostradas a seguir.

$$\pi - \texttt{intro} \quad \frac{\Gamma \vdash x : \tau,\ y : \sigma}{\Gamma \vdash \langle x, y\rangle : \tau \times \sigma} \qquad \pi - \texttt{elim}^1 \quad \frac{\Gamma \vdash x : \langle \tau \times \sigma\rangle}{\Gamma \vdash \pi_1 x : \tau} \qquad \pi - \texttt{elim}^2 \quad \frac{\Gamma \vdash x : \langle \tau \times \sigma\rangle}{\Gamma \vdash \pi_2 x : \sigma}$$

Adicionamos duas regras à nossa semântica estrutural.

$$\pi_1 \langle m, n\rangle \rightarrowtail m$$
$$\pi_2 \langle m, n\rangle \rightarrowtail n$$

9.5.4 Soma

O tipo *soma* representa um valor que pode pertencer a um dentre vários tipos – mas somente a um deles. Conceitualmente, é semelhante ao tipo `union` na linguagem *C*, "registros variantes" em Pascal, e `Either` em Haskell.

O termo $\text{inj}_1\, M$ resulta em um termo do tipo $\sigma + \tau$, assim como o termo $\text{inj}_2\, N$.

O termo

$$\textbf{case}\ \text{inj}_i\, t\ \textbf{of}\ m; n$$

resulta em m se $i = 1$, e n se $i = 2$.

A gramática é aumentada da seguinte maneira.

$\langle M\rangle ::= \text{inj}_1 \langle M\rangle \mid \text{inj}_2 \langle M\rangle \mid \textbf{case}\ \langle J\rangle\ \textbf{of}\ \langle V\rangle; \langle V\rangle$

$\langle T\rangle ::= \langle T\rangle + \langle T\rangle$

$$\texttt{soma}^1 \quad \frac{\Gamma \vdash m : \tau}{\Gamma \vdash \text{inj}_1\, m : \tau + \sigma} \qquad \texttt{soma}^2 \quad \frac{\Gamma \vdash n : \sigma}{\Gamma \vdash \text{inj}_2\, n : \tau + \sigma}$$

Adicionamos duas novas regras para β-redução.

$$\textbf{case}\ \text{inj}_1\, t\ \textbf{of}\ m; n \rightarrowtail mt$$
$$\textbf{case}\ \text{inj}_2\, t\ \textbf{of}\ m; n \rightarrowtail nt$$

E um novo contexto de avaliação

$\langle C\rangle ::= \textbf{case}\ []\ \textbf{of}\ \langle M\rangle; \langle M\rangle$

Só permitimos a avaliação de termos na primeira posição (entre **case** e **of**, porque dependendo deste valor, um dos outros termos será descartado.

Unicidade

Suponha que tenhamos os tipos $\sigma + \tau$ e $\sigma + \psi$. Ainda que saibamos que M é do tipo σ, não temos como determinar o tipo do termo $\text{inj}_1\, M$, que poderia ser tanto $\sigma + \tau$ como $\sigma + \psi$.

Há diferentes maneiras de resolver este problema:

- delegar o problema ao algoritmo de inferência de tipos, deixando o segundo tipo indeterminado;
- permitir que o segundo tipo represente *todos* os possíveis tipos naquela posição;
- adicionar anotações em inj_i, de forma que seja possível determinar, de imediato, o tipo do termo:

$$\text{inj}_1^{\sigma+\tau}\, M \neq \text{inj}_1^{\sigma+\psi}\, M.$$

9.6 Recursão

O combinador Y não pode ter tipo definido no λ-Cálculo simplesmente tipado:

$$\lambda f \cdot (\lambda x \cdot f(xx))(\lambda x \cdot f(xx))$$

Como x é aplicado a si mesmo, não podemos determinar (com o sistema de tipos simples que descrevemos) o tipo desta variável: suponha que x é do tipo τ. Como aplicamos (xx), então x deve também ser do tipo $\tau \to \sigma$, levando a uma contradição, porque não podemos na teoria simples de tipos acomodar tipos recursivos ($\tau = \tau \to \ldots$).

Podemos, no entanto, adicionar artificialmente à linguagem um operador de ponto fixo.

$$(\mu f : \tau \cdot x \cdot M)V \rightarrowtail M[V/x][\mu f : \tau \cdot \lambda x \cdot M/f]$$

Note que a redução troca f por sua expansão, "$\mu f : \tau \cdot \lambda x \cdot M$".

Introduzimos também uma nova regra para tipagem:

$$\frac{\Gamma, f : \tau \to \sigma \vdash \lambda x : \tau \cdot M : \tau \to \sigma}{\Gamma \vdash \mu f : \tau \to \sigma \lambda c \cdot M : \tau \to \sigma}$$

9.7 Isomorfismo de Curry-Howard

Existe uma correspondência biunívoca entre derivações no λ-Cálculo tipado e provas em certas Lógicas Intuicionistas, da forma a seguir.

proposições	tipos
$P \Rightarrow Q$	tipo $P \to Q$
$P \wedge Q$	tipo $P \times Q$
prova de P	termo do tipo P
P é demonstrável	tipo P é habitado por algum termo

Um estudo detalhado do isomorfismo de Curry-Howard fica fora do escopo deste texto.

Notas

Uma exposição bastante detalhada de sistemas de tipos em linguagens de programação é dada no livro de Benjamin Pierce [Pie02].

Exercícios

Ex. 68 — Demonstre o Lema 9.3.

Ex. 69 — A semântica estrutural do **case** .. **of** é parecida com a do **if**. Tente definir booleanos e a semantica do comando de seleção **if** usando apenas os tipos unitário e soma.

Capítulo 10

Concorrência e Sistemas Reativos

Nos Capítulos anteriores, tratamos da semântica de programas não-concorrentes e onde interessava basicamente o efeito do programa após seu término: tratamos o não-término de programas como algo indesejável. No entanto, há diversos exemplos de programas que são projetados para não terminar: sistemas operacionais, servidores, drivers e programas de controle, por exemplo.

Sistemas reativos são inerentemente paralelos, e sua comunicação com o ambiente determina grande parte de seu comportamento.

Apesar de termos feito uma breve incursão à semântica operacional de execução paralela, com memória compartilhada, não tratamos da semântica de programas concorrentes sem memória compartilhada.

Agora voltamos a atenção a programas reativos, que incluem os programas concorrentes sem memória compartilhada, onde a troca de mensagens é o mecanismo básico de comunicação e sincronização.

Há diversas linguagens para a descrição formal de programas concorrentes. Aqui trataremos apenas de duas – o *Calculus of Communicating Systems* (CCS) e o *Communicating Sequential Processes* (CSP).

10.1 CCS

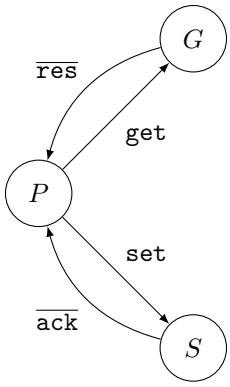
O *Calculus of Communicating Systems* (CCS) foi criado por Robin Milner, que observa que a troca interna de informação em um processo não difere, essencialmente, da troca de mensagens entre processos (a atribuição de valor a uma variável não é diferente do envio de um valor de um processo para outro). Assim, a comunicação entre agentes é o fundamento do CCS.

Começamos nossa abordagem do CCS através de um exemplo: suponha que queiramos modelar um buffer unitário (um registrador em hardware, por exemplo). O protocolo para acesso ao registrador permite duas operações, que descrevemos a seguir. As ações que representam entrada são descritas sem notação extra, enquanto as que representam saída são descritas com uma barra acima do nome ("$\overline{a}$", por exemplo).

- enviamos um valor (em uma mensagem que chamamos de `set`) para ser armazenado, e o regitrador responde com uma mensagem $\overline{\texttt{ack}}$;

- enviamos um pedido (em uma mensagem `get`) e o registrador nos responde com o valor armazenado, em uma mensagem $\overline{\texttt{ans}}$.

Estes são os dois comportamentos possíveis que modelaremos. O agente pode estar em três estados:



Note que não há diferença entre interpretar o diagrama acima como se representasse um processo que pode estar em três estados, mudando de um para outro a partir de eventos discretos (get,$\overline{\texttt{ans}}$, etc), ou como se fossem *três* agentes, cada um enviando mensagens aos outros. No CCS, não vemos diferença entre o *estado de um processo* e um *agente trocando mensagens*.

O agente P pode executar duas ações, get ou set;

- se executar get, deverá depois comportar-se como G;
- se executar set, deverá depois comportar-se como S;

No CCS, "executar uma ação a e depois comportar-se como P" é o mesmo que "enviar uma mensagem a para P". A notação para prefixo (sequencia de uma mensagem seguida de um processo) é "$a.P$". A notação para escolha (um dentre dois comportamentos, ou processos) é "P+Q".

Podemos definir nomes para processos no CCS usando a notação "$P \stackrel{\text{def}}{=} \cdots$" Assim, o registrador é modelado em CCS como a seguir.

$$P \stackrel{\text{def}}{=} (\texttt{set}.S) + (\texttt{get}.G)$$
$$G \stackrel{\text{def}}{=} \overline{\texttt{ans}}.P$$
$$S \stackrel{\text{def}}{=} \overline{\texttt{ack}}.P$$

Podemos abreviar esta descrição, escrevendo

$$P \stackrel{\text{def}}{=} (\texttt{get}.\overline{\texttt{ans}}.P) + (\texttt{set}.\overline{\texttt{ack}}.P)$$

O processo P, do ponto de vista externo, aceita duas possíveis mensagens de entrada e duas de saída. Algumas vezes representamos processos com *portas* de entrada e saída, como no diagrama a seguir.

- $\mathcal{A}$ é um conjunto infinito de nomes de ações, e $\overline{\mathcal{A}}$ é o conjunto de co-nomes de ações, de forma que
$$a \in \mathcal{A} \Leftrightarrow \overline{a} \in \overline{\mathcal{A}}.$$
- $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ é o conjunto de rótulos (que dão nomes a portas). Ainda que os elementos de $\mathcal{L}$ sejam os mesmos de $\mathcal{A}$ e $\overline{\mathcal{A}}$, enfatizamos que tratamos de rótulos escrevendo $\ell, \ell', \ell_i \in \mathcal{L}$.

Por convenção, $\overline{\overline{a}} = a$.

Existe um processo nulo, denotado por $\emptyset$. Este tem significado semelhante a $\perp$ na semântica denotacional: o processo nulo não evolui, e nenhuma ação é executada a partir dele.

- *sequência:* pode-se determinar a execução do processo que começa com uma ação a e depois se comporta como um proceso P; a notação para isto é $a.P$.

- *composição paralela:* $P|Q$ denota a o processo que se comporta como a execução dos dois processos, P e Q, simultaneamente;
- *escolha:* denotamos por $P + Q$ a o processo que pode se comportar como um dentre P ou Q. Não se trata de decisão ou de paralelismo. $P + Q$ simplesmente significa que o processo poderá se comportar como P, ou como Q. No entanto, note que quando escrevemos $aP + bQ$, determinamos que se a ação a for realizada, o comportamento será o de P, e se a ação b for realizada, será o de Q;
- *renomeação:* $P[f]$ é como o processo P, exceto que a função de renomeação f é aplicada sobre suas ações. Também é comum denotar $P[b/a]$.
- *restrição:* νaP é um processo que se comporta como P, exceto que a ação a não pode ser observada.

É comum denotar $P_1 + P_2 + \cdots + P_n$ por $\sum_{i=1}^{n} P_i$.
Definimos os conjuntos a seguir:

- $X \in \mathcal{X}$, variáveis de agente;
- $A \in \mathcal{K}$, constantes de agente;
- $P \in P$, expressões de agentes;
- $\mathcal{A}$, um conjunto de ações (ou *rótulos*);
- $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$
- $\mathbf{Act} = \mathcal{L} \cup \{\tau\}$

A gramática descrevendo processos é:

$\langle P \rangle ::= \emptyset \mid K \mid a.P \mid P + P \mid P \mid P \mid P[b/a] \mid \nu aP$

Exemplo 10.1. Um *semáforo* é uma construção de programação usada em controle do uso de recursos compartilhados em programas concorrentes. O semáforo consiste essencialmente de um conjunto de *tokens* (semelhantes a "fichas"), implementado como um contador, e dois procedimentos para acesso, que são usados pelos processos que pretendem acessar o recurso:

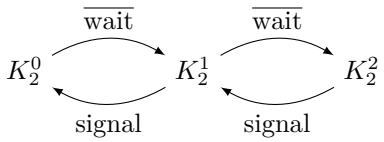
- **wait**, ou P, que toma uma das fichas, se houver. Caso não haja, o processo que executou o **wait** adormece e espera até que haja uma ficha disponível;
- **signal**, ou V, que devolve uma ficha (ou insere uma nova).

Usualmente, semáforos são criados com um número fixo de fichas, que são retiradas e devolvidas por diferentes processos. Quando as n fichas de um semáforo se esgotam, temos certeza de que há n processos com acesso ao recurso que ele protege. Outros processos podem tentar tomar uma ficha chamando o procedimento **wait**, mas o procedimento só retornará depois que uma das fichas for devolvida.

Modelaremos um semáforo com duas fichas com o CCS. Se um semáforo tem um total de n fichas, e p delas já foram tomadas por processos, denotaremos[1] seu comportamento por K_n^p.

Para facilitar a modelagem, interpretaremos cada chamada a wait como uma *saída* do processo semáforo, e cada signal como *entrada*. Assim, denotamos $\overline{\text{wait}}$ e signal.

O diagrama a seguir ilustra o comportamento de um semáforo do tipo K_2.



[1] Esta notação é diferente da usada por Davide Sangiorgi em seu livro [San12] – ele denota por K_n^p o semáforo com n *estados internos* e p fichas tomadas. Nosso semáforo K_2^0 seria, na notação dele, "K_3^0".

A partir do diagrama, fica clara a definição do processo.

$$K_2^0 = \overline{\text{wait}}.K_2^1$$
$$K_2^1 = \overline{\text{wait}}.K_2^2 + \text{signal}\,.K_2^1$$
$$K_2^2 = \text{signal}\,.K_2^1$$

O exercício 71 pede a demonstração de equivalência entre um semáforo com quatro fichas e dois semáforos com duas fichas cada um. ◀

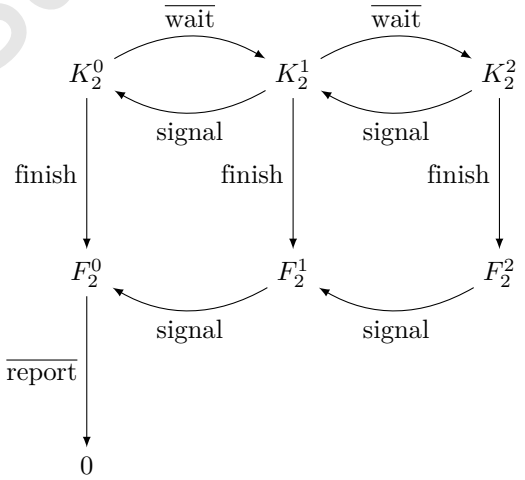
Exemplo 10.2. Neste exemplo estudaremos, de forma abstrata, o comportamento de um servidor que realize votações via rede.

- um eleitor envia um voto;
- o servidor envia um comprovante de voto – *sem* identificar o voto.
- um administrador envia um comando de finalização da eleição;
- o servidor devolve ao administrador um relatório de votos.

O comando de finalização não pode abortar votos em andamento – ele deve esperar até que todos os eleitores atualmente em processo de voto terminem, e só depois enviar o relatório.

Construimos agora um modelo bastante simples, com dois tipos de processo, o de votação (que poderá ter várias instâncias rodando) e o de administração (que será único).

- para que mais de um processo de votação ocorra em paralelo, usamos um semáforo: cada processo de voto toma uma ficha, e a devolve após o término do voto.
- uma mensagem finish faz o processo mudar o comportamento: ele passa a não aceitar novos votos (o semáforo passa a não aceitar mais mensagens *wait*).
- uma mensagem report envia o relatório.



Traduzindo para o CCS, temos as definições a seguir.

$$
\begin{aligned}
V &= \text{wait} \,.\, \text{voto} \,.\overline{\text{rec}}.\overline{\text{signal}}.V \\
K_2^0 &= \overline{\text{wait}}.K_2^1 + \text{finish} \,.F_2^0 \\
K_2^1 &= \overline{\text{wait}}.K_2^2 + \text{signal} \,.K_2^0 + \text{finish} \,.F_2^1 \\
K_2^2 &= \text{signal} \,.K_2^1 + \text{finish} \,.F_2^2 \\
F_2^0 &= \overline{\text{report}}.0 \\
F_2^1 &= \text{signal} \,.K_2^0 \\
F_2^2 &= \text{signal} \,.K_2^1
\end{aligned}
$$

Podemos simplificar esta descrição algébrica, já que cada F_2^i é igual a $\nu\ \overline{\text{wait}} K_2^i$ (os comportamentos do tipo F são iguais aos K, exceto que não permitimos operações $\overline{\text{wait}}$):

$$
\begin{aligned}
V &= \text{wait} \,.\, \text{voto} \,.\overline{\text{rec}}.\overline{\text{signal}}.V \\
K_2^0 &= \overline{\text{wait}}.K_2^1 + \text{finish} \,.\overline{\text{report}}.0 \\
K_2^1 &= \overline{\text{wait}}.K_2^2 + \text{signal} \,.K_2^0 + \text{finish} \,.\nu\ \overline{\text{wait}}\ K_2^1 \\
K_2^2 &= \text{signal} \,.K_2^1 + \text{finish} \,.\nu\ \overline{\text{wait}}\ K_2^2
\end{aligned}
$$

O processo de votação, com dois processos paralelos é

$$P = V|V|K_2^0.$$

Com n processos paralelos, teríamos

$$P = \overbrace{V|V|V|\cdots|V}^{n \text{ processos}}\ |\ K_n^0.$$ ◀

10.1.1 Semântica Estrutural

A semântica estrutural do CCS é dada a seguir, onde $\alpha \in \mathbf{Act}$ e $a \in \mathcal{L}$

$$
\begin{array}{lll}
\texttt{ação} & \alpha.P \to^{\alpha} P & \\
\texttt{soma} & \dfrac{P \to^{\alpha} P'}{P + Q \to^{\alpha} P' + Q} & \texttt{soma}\ \dfrac{Q \to^{\alpha} Q'}{P + Q \to^{\alpha} P + Q'} \\
\texttt{con} & \dfrac{P \to^{\alpha} P',\ \ (K \stackrel{\text{def}}{=} P)}{K \to^{\alpha} P'} & \texttt{rest}\ \dfrac{P \to^{\alpha} P'\ \ (\alpha \neq a)}{\nu a P \to^{\alpha} \nu a P} \\
\texttt{com}^1 & \dfrac{P \to^{\alpha} P'}{P|Q \to^{\alpha} P'|Q} & \texttt{com}^2\ \dfrac{Q \to^{\alpha} Q'}{P|Q \to^{\alpha} P|Q'} \\
\texttt{com}^3 & \dfrac{P \to^{a} P',\ \ Q \to^{\overline{a}} Q'}{P|Q \to^{\tau} P'|Q'} & \\
\texttt{ren} & \dfrac{P \to^{\alpha} P'}{P[f] \to^{f(\alpha)} P'[f]} &
\end{array}
$$

10.1.2 Propriedades de bissimulação

A relação de bissimulação $\approx$ entre processos é definida da mesma forma que no Capítulo 6, exceto que aqui descrevemos simulações de um processo CCS por outro processo, também CCS.

Teorema 10.3. *As seguintes propriedades valem para quaisquer processos P, Q e R.*

$$\begin{array}{rr} P+0 \approx P & P|0 \approx P \\ P+Q \approx Q+P & P|Q \approx Q|P \\ (P+Q)+R \approx P+(Q+R) & (P|Q)|R \approx P|(Q|R) \\ P+P \approx P & \end{array}$$

Teorema 10.4. *Para todos P, Q,*

$$\begin{array}{rr} \nu b(\nu a P) \approx \nu a(\nu b P) & \\ \nu a(P|Q) \approx (\nu a P)|Q & (\text{se } a \text{ não é livre em } Q) \\ \nu a P \approx \nu b(P[b/a]) & (\text{se } b \text{ não é ligada em } Q) \\ \nu a(\alpha.P) \approx 0 & (\text{se } \alpha = a \text{ ou } \alpha = \overline{a}) \\ \nu a(\alpha.P) \approx \alpha(\nu a P) & (\text{se } \alpha \neq a \text{ e } \alpha \neq \overline{a}) \end{array}$$

Teorema 10.5. *Se $K \stackrel{\text{def}}{=} P$ então $K \approx P$.*

O Exercício 74 pede a demonstração do Teorema 10.6.

Teorema 10.6. *Para todos processos P e Q,*

$$\begin{aligned} a.P|b.Q &\approx a.(P|b.Q) + b.(a.P|Q) \\ a.P|\overline{a}.Q &\approx a.(P|\overline{a}) + \overline{a}.(a.P|Q) + \tau(P|Q) \end{aligned}$$

Teorema 10.7. *Para todo contexto C, e todos processos P e Q,*

$$P \approx Q \Rightarrow C[P] \approx C[Q]$$

10.1.3 Recursão e ponto fixo

Para definir comportamento recorrente, usamos a definição recursiva de processos, como

$$A \stackrel{\text{def}}{=} a.A. \tag{10.1}$$

Embora esta definição seja apropriada para modelagem, ela nos trará problemas em seu tratamento teórico, como já vimos na semântica do comando **while** na seção 3.3 do Capítulo 3.

Mesmo quando não estamos construíndo uma semântica operacional, podemos usar o operador de ponto fixo. A definição de processo na equação 10.1 passaria a ser escrita da seguinte forma:

$$A \stackrel{\text{def}}{=} \text{lfp}(X = a.X).$$

Precisamos agora de uma nova regra para definições recursivas.

$$\texttt{rec} \quad \frac{E[\text{lfp}(X \stackrel{\text{def}}{=} E)/X] \rightarrow^{\alpha} E'}{\text{lfp}(X \stackrel{\text{def}}{=} E) \rightarrow^{\alpha} E'}$$

10.2 CSP

A linguagem de modelagem *Communicating Sequential Processes* (CSP), desenvolvida por Tony Hoare, não é essencialmente diferente do CCS, visto na seção anterior. As diferenças são principalmente nas facilidades sintáticas: o CSP tenta oferecer sintaxe diferente para diferentes variações da mesma idéia, enquanto o CCS é sintaticamente mais econômico.

Algumas similaridades entre o CCS e o CSP são mostradas nas tabelas a seguir.

CCS	CSP
agentes	processos
ações	eventos
portas	canais
rótulos	símbolos

	CCS	CSP
prefixação	$a.P$	$e \to P$
escolha	$P + Q$	$P \Box Q$
concorrência	$P \vert Q$	$P \vert\vert Q$
ocultação	$P\{a_1, \ldots, a_n\}$	$P\{e_1, \ldots, e_n\}$
renomeação	$P[f]$	$f(P)$
definições	$N \stackrel{\text{def}}{=} P$	$N = P$
recursão	$\text{lfp}(X = P)$	$\mu X.P$
processo nulo	0	$STOP$
saída	$\overline{a}(v).P$	$c!e \to P$
entrada	$a(v).P(v)$	$c?v \to P(v)$

(esta seção está incompleta)

10.2.1 Semântica Denotacional

Notas

Há diversas técnicas para a especificação de programas concorrentes usando troca de mensagens para comunicação e sincronização: *Communicating Sequential Processes* (CSP) [Hoa85]; *Calculus of Communicating Systems* (CCS) [Mil89], e seu sucessor, o π-Cálculo [Hen07]; *Algebra of Communicating Processes* (ACP) [BK87]; Join Calculus [FG00] e PEPA [Hil05] são alguns exemplos.

O livro de Davide Sangiorgi [San12] trata de bissimulação e coindução, usando largamente o CCS como exemplo.

Os livros de Glynn Winskell [Win94] e de Aaron Stump [Stu14] contém uma introdução muito simplificada à semântica de programas concorrentes.

Neste texto demos para o CCS uma semântica operacional, e para o CSP uma semântica denotacional. Estas foram as abordagens originais de Robin Milner para o CCS e de Tony Hoare para o CSP – no entanto, posteriormente diferentes estilos de semântica foram propostos para todas as álgebras de programas concorrentes.

De todas as tentativas de formalização de programas concorrentes, o CSP foi o de maior impacto no projeto de linguagens: há bibliotecas para implementação de primitivas baseadas no CSP em grande parte das linguagens modernas, e algumas delas (como Go) foram desenvolvidas já com suporte a estas primitivas.

Exercícios

Ex. 70 — Construa uma semântica denotacional para o CCS, e uma estrutural para o CSP.

Ex. 71 — Mostre que $K_2^0 | K_2^0 \approx K_4^0$.

Ex. 72 — Mostre que se $K \stackrel{\text{def}}{=} a.K$ então $K|K \approx K$.

Ex. 73 — Verifique se $a.(b + a) \approx ab + ac$ (prove que sim ou que não).

Ex. 74 — Prove o Teorema 10.6.

Ex. 75 — Modele, usando o CCS, um servidor de nomes. Inicialmente, ele realiza dois tipos de comunicação:

- um administrador modifica uma entrada, e recebe um recibo de transação (`ack`);
- um usuário envia um apergunta, e recebe a resposta

Depois modifique o modelo para, a cada modificação feita pelo administrador, enviar ao servidor de *logs* uma notificação de que a mudança foi feita. Uma última modificação consiste em registrar também nos logs as perguntas feitas por clientes quando não há resposta (um nome que não está registrado, por exemplo).

Capítulo 11

π-Cálculo Aplicado e Corretude de Protocolos

Aqui damos uma breve introdução ao π-Cálculo Aplicado [AF01], desenvolvido por Martín Abadí e Cédric Fournet. O π-Cálculo Aplicado é uma linguagem derivada do π-Cálculo construída especificamente para descrição e verificação de protocolos criptográficos, mas que pode ser usada para a descrição e prova de corretude de protocolos e algoritmos concorrentes e distribuídos.

11.1 Descrição informal

O π-Cálculo Aplicado foi desenvolvido para modelar protocolos criptográficos, a fim de facilitar verificações de corretude.

Um protocolo criptográfico é executado por diversos *participantes*, mas o modelo que usamos para descrevê-lo e demonstrar sua corretude inclui também um *adversário*, que coleta informações do "ambiente" – por exemplo, mensagens enviadas por canais inseguros.

Para facilitar a modelagem de protocolos criptográficos, o π-Cálculo Aplicado trata de forma separada

- *termos*, que representam a informação que transita entre processos durante a execução do protocolo;
- *processos simples*, que são os processos do protocolo que estamos modelando, sem considerar sua interação com o ambiente;
- *processos estendidos*, os processos do protocolo, já levando em conta sua interação com o ambiente – do ponto de vista da Criptografia, isto é o mesmo que "interação com o adversário".

A informação que o protocolo expõe ao ambiente é modelada como um conjunto de *substituições ativas* $[M/x]$. As substituições ativas tem este nome porque se comportam como se tivessem vida própria, "contaminando" os processos que executam em paralelo com ela:

$$P \mid [M/x] \quad \equiv \quad P[M/x] \mid [M/x],$$

ou seja, o processo P rodando em paralelo com o processo $[M/x]$ é equivalente ao processo $P[M/x]$, também executando em paralelo com o processo $[M/x]$.

Assim, podemos visalizar a execução de um protocolo no π-Cálculo Aplicado contendo *processos simples*[1], realizado pelos participantes, além de processos de substituição ativa, que "flutuam" ao redor dos processos simples. O conjunto de substituições ativas ao redor de um protocolo em execução é chamado de *quadro* (frame), representando a informação tornada pública. Processos que incluam subprocessos que se comunicam com estes processos externos são chamados de *processos estendidos*.

[1] "Plain processes" em Inglês.

[haverá uma figura aqui]

Eventualmente, ao tratar de demonstrações de segurança, incluiremos também processos representando o adversário.

11.2 Descrição formal do π-Cálculo Aplicado

O conjunto de nomes d efunção que podem ser usados nos termos é chamado de *assinatura*[2], e usualmente denotado por Σ.

A gramática para termos é

$\langle termo \rangle ::= \langle nome \rangle \mid \langle var \rangle \mid \langle F \rangle(\langle termo \rangle, \langle termo \rangle, \ldots, \langle termo \rangle)$

onde $\langle F \rangle \in \Sigma$.

Ou, como é comum encontrar na literatura,

M, N	::=	termos
$a, b, c. \ldots, s$		nomes
x, y, z		variáveis
$g(M_1, M_2, \ldots, M_k)$		aplicação de função ($g \in \Sigma$)

A gramática para *processos simples* é

P, Q, R	::=	processos
0		nulo
$P \mid Q$		composição paralela
$!P$		replicação
$\nu n.P$		restrição de nome
if $M = N$ **then** P **else** Q		condicional
$u(x).P$		entrada
$\overline{u}\langle M \rangle .P$		saída

A gramática para processos estendidos é mostrada a seguir.

A, B, C	::=	processos estendidos
P		processo simples
$A \mid B$		composição paralela
$\nu n.A$		restrição de nome
$\nu x.A$		restrição de variável
$[M/x], [x = M]$		substituição ativa

A descrição da semântica do π-Cálculo Aplicado é descrita em partes:

- uma *teoria equacional* que, grosso modo, define uma relação de equivalência entre processos, *sem modelar passos de computação*;
- uma *semântica operacional interna*, que descreve transições (passos de computação) dentro do próprio protocolo;
- uma *semântica operacional estendida*, que descreve transições que incluem interação com o ambiente.

A teoria equacional é dada a seguir, definindo a relação de equivalência $\equiv$. Esta teoria depende da assinatura Σ, e denotamos $\Sigma \vdash M = N$ quando M e N são equivalentes na teoria associada com Σ.

[2] O nome "assinatura" não tem, neste contexto, relação com assinaturas criptográficas.

$$
\begin{array}{rcll}
\texttt{par.0} & A \equiv A|0 & & \\
\texttt{par.A} & A \mid (B \mid C) \equiv (A \mid B) \mid C & & \\
\texttt{par.C} & A \mid B \equiv B \mid A & & \\
\texttt{repl} & !P \equiv P \mid !P & & \\
\texttt{new.0} & \nu n.0 \equiv 0 & & \\
\texttt{new.C} & \nu u\ \nu v.A \equiv \nu v\ \nu u.A & & \\
\texttt{new.par} & A|\nu u.B \equiv \nu u(A \mid B) & & (\text{se } u \notin fv(A) \cup fn(A)) \\
\texttt{alias} & \nu x.[M/x] \equiv 0 & & \\
\texttt{subst} & [M/x] \mid A \equiv [M/x] \mid A[M/x] & & \\
\texttt{rewrite} & [M/x] \equiv [N/x] & & (\text{se } \Sigma \vdash M = N)
\end{array}
$$

A semântica operacioanl interna é dada a seguir.

$$
\begin{array}{rl}
\texttt{comm} & \overline{a}\,\langle x\rangle\,.P|a(x).Q \rightarrowtail P|Q \\
\texttt{then} & \textbf{if } M = M \textbf{ then } P \textbf{ else } Q \rightarrowtail P \\
\texttt{else} & \dfrac{\Sigma \not\vdash M = N}{\textbf{if } M = N \textbf{ then } P \textbf{ else } Q \rightarrowtail Q}
\end{array}
$$

A seguir está a semântica estendida do π-Cálculo Aplicado.

$$
\begin{array}{rl}
\texttt{in} & c(x).P \rightarrowtail^{c(M)} P[M/x] \\
\texttt{out} & \overline{u}.P \rightarrowtail^{\overline{u}} P \\
\texttt{open} & \dfrac{A \rightarrowtail^{\overline{c}(u)} A',\ \ u \neq c}{\nu u.A \rightarrowtail^{\nu u.\overline{c}(u)} A'} \\
\texttt{scope} & \dfrac{A \rightarrowtail^{\alpha} A',\ \ u \notin var(\alpha)}{\nu u.A \rightarrowtail^{\alpha} \nu u.A'} \\
\texttt{par} & \dfrac{A \rightarrowtail^{\alpha} A',\ \ bv(\alpha) \cap fv(B) = bn(\alpha) = fn(B) = \emptyset}{A|B \rightarrowtail^{\alpha} A'|B} \\
\texttt{struct} & \dfrac{A \equiv B\ \ B \rightarrowtail^{\alpha} B'\ \ B' \equiv A'}{A \rightarrowtail^{\alpha} A'}
\end{array}
$$

Notas

O π-Cálculo aplicado é uma versão mais geral do *Spi-Calculus [AG99]*, desenvolvido por Martín Abadi e Andrew Gordon.

Cas Cremers e Sjouke Mauw descrevem [CM12] o uso semântica operacional na verificação de protocolos criptográficos, de forma diferente da que mostramos.

O conceito de teoria equacional, do ponto de vista de Ciência da Computação e Teoria de Linguagens de Programação, é desenvolvido no Capítulo 3 do livro de John Mitchell [Mit96].

Apêndice ℸ

Linguagens Formais

Este Apêndice traz uma brevíssima apresentação do formalismo usado na descrição da sintaxe de linguagens de programação.

Os livros de Michael Sipser [Sip07], de Peter Linz [Lin11] e de Aho, Hopcroft e Ullman [Joh06] tratam deste assunto. Para uma segunda leitura, há o livro de Jeffrey Shallit [Sha08].

Definição ℸ .1 (alfabeto)**.** Um *alfabeto* é um conjunto finito de símbolos. ⧫

Definição ℸ .2 (cadeia, palavra)**.** Uma *cadeia* ou *palavra* é uma sequência finita de elementos de um alfabeto.

Denotamos por ε a palavra vazia. ⧫

Exemplo ℸ .3. O alfabeto de bits é $\{0, 1\}$. Alguns exemplos de cadeias de bits são 0, 1, 000, 0110010101. ◂

Exemplo ℸ .4. Sobre o alfabeto $\{a, b, c, d\}$ são cadeias ϵ, a, b, $aaadddcb$, $abbc$. ◂

Exemplo ℸ .5. Podemos usar quaisquer conjunto de símbolos que queiramos como alfabeto. Um exemplo é $\{\diamondsuit, \clubsuit, \heartsuit, \spadesuit\}$. Sobre este alfabeto podemos formar, por exemplo, as cadeias $\diamondsuit\heartsuit\diamondsuit\heartsuit$, $\diamondsuit\clubsuit\clubsuit$ e $\clubsuit\clubsuit\clubsuit$. ◂

Definição ℸ .6 (comprimento de palavra)**.** O *comprimento* de uma palavra é a quantidade de símbolos nela. ⧫

Exemplo ℸ .7. Os comprimentos de algumas cadeias são dados a seguir.

ε	0
a	1
b	1
aaa	3
$babb$	4

◂

Definição ℸ .8 (concatenação de palavras)**.** Sejam $\alpha = \alpha_1\alpha_2 \ldots \alpha_k$ e $\beta = \beta_1\beta_2 \ldots \beta_n$ duas palavras. Então $\alpha\beta = \alpha_1\alpha_2 \ldots \alpha_k\beta_1\beta_2 \ldots \beta_n$ é a *concatenação* das palavras α e β.

Denotamos por $(\alpha)^n$ a concatenação de n cópias da palavra α. Para um símbolo isolado s, não usamos os parênteses, denotando s^n. Quando usamos o mesmo contador mais de uma vez na mesma expressão, significa que as quantidades devem ser iguais: em "$a^n b^n$", as quantidade de a's e b's são iguais; em $a^k b^n$, não. ⧫

Evidentemente, $\varepsilon\alpha = \alpha\varepsilon = \alpha$ para qualquer palavra α.

Exemplo ℸ .9. A concatenação de abb com ca é $abbca$. ◂

Exemplo ⍑ **.10.** Denotamos por $a^n b^n c^n$ as palavras contendo sequências de a, b e c, nesta ordem, sendo que cada palavra tem exatamente a mesma quantidade de a's, b's e c's: $\{\varepsilon, abc, aabbcc, aaabbbccc, \ldots\}$. ◀

Exemplo ⍑ **.11.** $L = a^n b^k c^n d^k$ é a linguagem das sequências de n a's, k b's, n c's e k d's, nesta ordem:

$$\{\varepsilon, ac, bd, abcd, aabccd, abbcdd, aabbccdd, \ldots\}.$$

◀

Definição ⍑ **.12** (linguagem)**.** Uma *linguagem* é um conjunto de palavras. ⧫

Definição ⍑ **.13** (concatenação de linguagens)**.** A *concatenação* de duas linguagens A e B, denotado AB, é o conjunto das concatenações de palavras, sendo a primeira de A e a segunda de B:

$$AB = \{ab \ : \ a \in A, \ b \in B\}.$$ ⧫

Definição ⍑ **.14** (fecho de Kleene)**.** Se A é um alfabeto, palavra ou linguagem, A^* é seu *fecho de Kleene*, ou simplesmente *fecho*. Pertencem ao fecho de Kleene todas as concatenações iteradas dos elementos de A, inclusive a palavra vazia.

Denotamos por A^+ o fecho transitivo de A, idêntico ao fecho de Kleene, exceto por não incluir a palavra vazia. ⧫

Exemplo ⍑ **.15.** Se $\Sigma = \{a, b, c\}$, então

$$\Sigma^* = \{\varepsilon, a, b, c, aa, bb, cc, ab, ac, bc.ba, ca., aaa, bbb, ccc, aab, aac, bba, bbc, \ldots\}$$
$$\Sigma^+ = a, b, c, aa, bb, cc, ab, ac, bc.ba, ca., aaa, bbb, ccc, aab, aac, bba, bbc, \ldots\}$$

Se $L = \{abc, def, g\}$, então

$$L^* = \{\varepsilon, abc, def, g, abcdef, defabc, abcg, gabc, defg, gdef, \ldots\}$$
$$L^+ = \{abc, def, g, abcdef, defabc, abcg, gabc, defg, gdef, \ldots\}$$ ◀

Usamos *gramáticas* como uma forma finita de especificar linguagens possivelmente infinitas. Uma gramática é a definição recursiva de uma linguagem, usando a concatenação de palavras na recursão. Esta definição recursiva é composta de *regras de produção*. Cada regra de produção é da forma $\alpha \to \beta$, indicando que a frase α pode ser trocada por β.

Exemplo ⍑ **.16.** A seguir temos uma gramática que representa um pequeno conjunto de frases e Língua Portuguesa.

$$\begin{aligned}
\textbf{frase} &\to \textbf{sujeito} \ \ \textbf{predicado} \\
\textbf{sujeito} &\to \text{João} \mid \text{Paulo} \mid \text{Luíza} \\
\textbf{verbo} &\to \text{fez} \mid \text{comprará} \mid \text{come} \\
\textbf{predicado} &\to \textbf{verbo} \mid \textbf{objeto} \\
\textbf{objeto} &\to \textbf{artigo} \mid \textbf{substantivo} \\
\textbf{artigo} &\to \text{o} \mid \text{um} \\
\textbf{substantivo} &\to \text{montanha} \mid \text{peixe}
\end{aligned}$$

Os símbolos em negrito são variáveis auxiliares, que não fazem parte da linguagem – são chamados de *não-terminais*. Os outros são palavras da linguagem, e chamados de *terminais*.

A seguir usamos a gramática para gerar uma frase. Em cada linha, substituímos uma variável usando

alguma regra de produção, até chegar a uma palavra sem variáveis.

frase
sujeito predicado
sujeito verbo objeto
sujeito come **objeto**
Paulo come **objeto**
Paulo come **artigo substantivo**
Paulo come um **substantivo**
Paulo come um peixe ◀

Definição ᛉ **.17** (gramática)**.** Uma *gramática* $G = (T, N, P, S)$ é composta de

- um conjunto T de *símbolos terminais*;
- um conjunto N de *símbolos não terminais*;
- uma relação $P \subseteq (N \cup T)^+ \times (N \cup T)^*$. Esta relação determina *regras de transição* para símbolos;
- um *símbolo inicial* S.

As regras de produção determinadas pela relação P podem ser escritas

$$\alpha \to \beta,$$

onde $\alpha \in (N \cup T)^+$ e $\beta \in (N \cup T)^*$ são sequencias de símbolos. ⧫

Definição ᛉ **.18** (substituição, derivação)**.** A aplicação de uma regra de produção é chamada de *substituição*.

Se uma regra é $A \to \alpha$, denotamos a substituição de A por α como "$\ldots A \ldots \Rightarrow \ldots \alpha \ldots$".

Uma *derivação* é uma sequência de substituições que leva uma palavra, possivelmente contendo símbolos não-terminais, até uma palavra só com símbolos terminais. Uma derivação $\alpha \Rightarrow \cdots \Rightarrow \beta$ é denotada $\alpha \longrightarrow^* \beta$. ⧫

Exemplo ᛉ **.19.** Uma gramática simples é mostrada a seguir.

$$\begin{aligned} A &\to Ab \\ A &\to Bc \\ B &\to A \\ B &\to x \end{aligned}$$

Usamos | para agrupar regras, e podemos reescrever a gramática:

$$\begin{aligned} A &\to Ab \mid Bc \\ B &\to A \mid x \end{aligned}$$

Derivamos uma palavra da gramática.

$$\begin{aligned} A &\Rightarrow Ab \\ &\Rightarrow Bcb \\ &\Rightarrow xcb \end{aligned}$$ ◀

Exemplo ᛉ **.20.** Uma gramática pode ter regras com mais que apenas um não-terminal à esquerda, como

na que segue.

$$A \to BaB \quad (1)$$
$$B \to bbb \quad (2)$$
$$B \to Ab \quad (3)$$
$$Aba \to c \quad (4)$$

A seguir usamos esta gramática para derivar uma palavra da linguagem que ela representa.

$$\begin{aligned} A &\Rightarrow BaB && (1)\\ &\Rightarrow Babbb && (2)\\ &\Rightarrow Ababbb && (3)\\ &\Rightarrow cbbb. \end{aligned}$$

Neste texto não usaremos este tipo de gramática. ◀

ᛉ.1 Linguagens Regulares

As linguagem mais simples que estudamos são as chamadas *linguagens regulares.*

Uma linguagem regular pode incluir palavras com a repetição de uma subpalavra um número indefinido de vezes: por exemplo, a linguagem $\{aa, aba, abba, abbba, abbbba, \ldots\}$, onde cada palavra é composta de dois a's, com qualquer quantidade de b's entre eles, é regular.

Palavras de linguagens regulares, no entanto, não podem ser geradas casando quantidades ou usando qualquer tipo de memória: a linguagen $\{b, aba, aabaa, aaabaaa, \ldots\}$, onde cada palavra é um b, cercado *pela mesma quantidade* de a's à direita e à esquerda, *não* é regular.

ᛉ.1.1 Gramáticas regulares; definição de linguagem regular

Definição ᛉ.21 (linguagem regular)**.** Uma gramática é *regular* se suas produção são todas da forma

$$A \to \alpha B$$

ou se são todas da forma

$$A \to B\alpha,$$

onde A, B são símbolos não terminais, e α é uma cadeia de terminais.

Uma linguagem é regular se é gerada por uma gramática regular. ♦

Exemplo ᛉ.22. A seguinte gramática é regular.

$$\begin{aligned} S &\to xA\\ S &\to yB\\ A &\to aS\\ B &\to bS\\ B &\to b \end{aligned}$$

Dentre outras, esta gramática gera as palavras

$$yb, ybb, ybbb, ybbbbb, xayb, xaybb, xaxaybbbbb.$$ ◀

Exemplo ᛉ.23. A seguinte gramática regular gera cadeias de dígitos representando números divisíveis

por 25.

$$S \to 0S \mid 1S \mid 2S \mid 3S \mid 4S$$
$$S \to 5S \mid 6S \mid 7S \mid 8S \mid 9S$$
$$S \to 00 \mid 25 \mid 50 \mid 75$$

Como exemplo, mostramos uma derivação.

$$\begin{aligned} S &\Rightarrow 3S \\ &\Rightarrow 37S \\ &\Rightarrow 3775 \end{aligned}$$ ◀

Teorema ⊺ **.24.** *Linguagens regulares são fechadas para concatenação, união, e fecho de Kleene.*

Uma *expressão regular* é uma forma de especificar uma linguagem regular. Expressões regulares são definições recursivas de linguagens regulares, mas não usam apenas concatenação – como sabemos que estas linguagens também são fechadas para união, concatenação e fecho transitivo, estes são usados explicitamente nos casos recursivos da definição.

Definição ⊺ **.25** (expressão regular)**.** *Expressões regulares sobre um alfabeto* são definidas recursivamente.

1. O símbolo $\emptyset$ é uma expressão regular, que não representa palavra nenhuma.
2. O símbolo ε é uma expressão regular que representa a palavra vazia.
3. Um símbolo isolado do alfabeto é uma expressão regular.
4. Se α é expressão regular, então α^* e α^+ também são.
5. Se α e β são expressões regulares, então também são:
   - $\alpha\beta$, representa o conjunto de concatenações de cadeias, uma de α e uma de β.
   - $\alpha + \beta$ representa a união das palavras representadas por α e β. ⧫
6. Concatenação tem precedência sobre união: $\alpha\beta|\gamma$ significa "um dentre $\alpha\beta$ e γ".
7. Parênteses podem ser usados para alterar a precedência: $\alpha(\beta|\gamma)$ significa "α seguido de um dentre β ou γ".

Usa-se parênteses para determinar precedência.

Exemplo ⊺ **.26.** A expressão regular $(ab)^*c$ representa a linguagem $\{c, abc, ababc, abababc, \ldots\}$ ◀

Exemplo ⊺ **.27.** A expressão regular $(ab|xy)(fg|jk)^*z$ representa a linguagem das palavras que começam com ab ou xy, seguida de uma sequência de fg's e jk's, seguida por um único z ◀

Exemplo ⊺ **.28.** A expressão regular a^*b^* representa a linguagem das palavras compostas por uma sequência de a's seguida por uma sequência de b's – mas *a linguagem não determina qualquer relação entre a quantidade de a's e de b's.* ◀

Teorema ⊺ **.29.** *O conjunto de linguagens representáveis por expressões regulares é exatamente o das linguagens regulares.*

Exemplo ⊺ **.30.** A linguagem gerada pela gramática do exemplo ⊺ .22 é também descrita pela expressão regular $((xa)^*yb^*)yb$. ◀

⍑.1.2 Autômatos Finitos

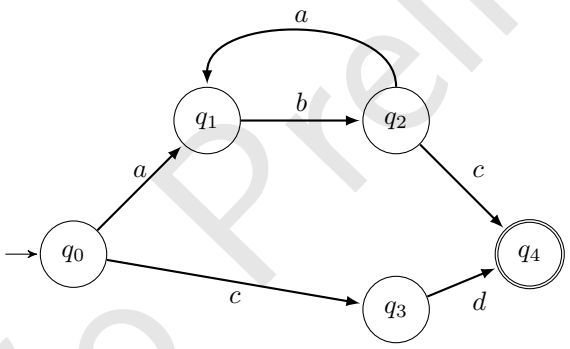
Linguagens são também descritas por máquinas conceituais, chamadas de *autômatos*. Linguagens regulares são descritas por *autômatos finitos*.

Um autômato finito é uma máquina que pode estar em diferentes *estados internos* (a quantidade de estados deve ser finita). O autômato começa a operar em um *estado inicial*, e lê símbolos de um alfabeto, escritos em uma fita. Não há restrição ao tamanho da fita. A cada símbolo lido, o autômato pode mudar de estado – o próximo estado depende do estado anterior e do símbolo lido. Quando acabarem os símbolos, o autômato verifica seu estado interno. Se for um de seus *estados finais*, ele para e *aceita* a palavra lida; caso contrário, *rejeita* a palavra.

Definição ⍑.31 (autômato finito)**.** Um *autômato finito* é composto de um alfabeto Σ, um conjunto de estados S, um estado inicial $s \in S$, um conjunto de estados finais $F \subseteq S$, e um programa δ, que relaciona pares de estado/programa com estados. ⧫

É comum representar autômatos finitos como grafos orientados. Cada nó representa um estado interno; um rótulo b em uma aresta $s_1 \to s_2$ significa que se o estado for s_1 e o símbolo lido for b, o próximo estado será s_2. Estados finais são marcados como nós especiais (usualmente, são representados como circunferência com traço duplo); o nó inicial é marcado tendo uma aresta entrando, vindo de nenhum nó.

Exemplo ⍑.32. A figura a seguir mostra um autômato finito.



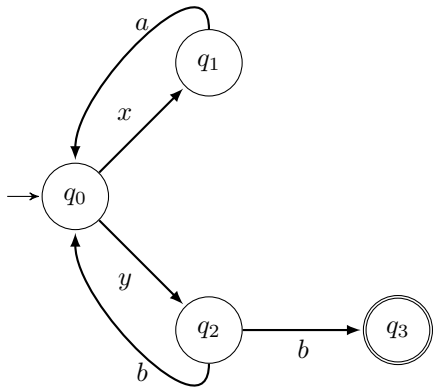
A linguagem deste autômato é claramente a mesma da expressão regular $((ab)^+c)|(cd)$. Este autômato é também descrito pela tupla $(\Sigma, S, s, F, \delta)$, onde

$$\begin{aligned}
\Sigma &= \{a, b, c, d\} \\
S &= \{q_0, q_1, q_2, q_3, q_4\} \\
s &= q_0 \\
F &= \{q_2, q_4\} \\
\delta &= \{(q_0, a, q_1), (q_1, b, q_2), (q_2, a, q_1), \\
&\qquad (q_2, c, q_4), (q_0, c, q_3), (q_3, d, q_4)\}\,.
\end{aligned}$$

Note que δ é uma função, embora não tenhamos imposto esta restrição (poderia ser uma relação). ◀

Teorema ⍑.33. *O conjunto de linguagens aceitas por autômatos finitos é exatamente o das linguagens regulares.*

Exemplo ⍑.34. O autômato a seguir representa a mesma linguagem da gramática do exemplo ⍑.22, também descrita pela expressão regular $(xa)^*yb^*$, dada no exemplo ⍑.30.



Os componentes do autômato são

$$\begin{aligned}\Sigma &= \{a, b, c, d\}\\ S &= \{q_0, q_1, q_2, q_3\}\\ s &= q_0\\ F &= \{q_3\}\\ \delta &= \{(q_0, x, q_1), (q_1, a, q_0), (q_0, y, q_2),\\ &\qquad (q_2, b, q_0), (q_2, b, q_3)\}\,.\end{aligned}$$

Aqui δ é relação, mas não função, porque o par estado/símbolo (q_2, b) é mapeado em dois estados diferentes. ◀

Definição ⍑ **.35** (determinismo em autômato finito)**.** Um autômato finito é *determinístico* quando seu programa é uma função $\delta : S \times \Sigma \to S$, e é *não-determinístico* quando seu programa é uma relação, mas não uma função: $\delta \in S \times \Sigma \times S$.

Abreviamos os dois tipos de autômato por "AFD" (autômato finito determinístico) e "AFN" (autômato finito não-determinístico). ⧫

Exemplo ⍑ **.36.** O autômato do exemplo ⍑ .32 é determinístico; o do exemplo ⍑ .34 é não-determinístico. ◀

O próximo Teorema diz essencialmente que o não-determinismo não aumenta o conjunto de linguagens aceitas por autômatos finitos (os AFNs aceitam linguagens regulares).

Teorema ⍑ **.37.** *Seja A um AFN. Então é possível construir um AFD que aceita a mesma linguagem que A.*

⍑ .1.3 Provando que uma linguagem não é regular

Para demonstrar que uma linguagem *não* é regular, usa-se o Lema do Bombeamento (⍑ .38).

Lema ⍑ **.38** (do bombeamento, para linguagens regulares)**.** *Se uma linguagem L é regular, então existe algum comprimento $p \in \mathbb{N}$ tal que, para toda palavra $s \in L$ com $|s| \geq p$, a palavra s pode ser decomposta em $s = xyz$, tal que*

- *para todo $i \in \mathbb{N}$, $xy^iz \in L$*
- $|y| > 0$

- $|xy| \leq p$

Exemplo ᛉ **.39.** A linguagem $L = a^n b^n$, onde a quantidade de a's é igual a de b's, não é regular, como demonstraremos usando o Lema do Bombeamento.

Suponha que L seja regular. Então vale para L o Lema do Bombeamento; seja p o comprimento determinado pelo lema para L.

Toda palavra $s \in L$ pode portanto ser decomposta em $s = xyz$, e conforme determina o lema, $xy^i z$ deve pertencer a L, com $|y| > 0$ e $|xy| \leq p$.

Dividimos o resto da demonstração em três casos:

i) y só contém a's. Neste caso, $xyyz$ não pode pertencer a L, porque teríamos mais a's do que b's;

ii) y só contém b's. Neste caso, $xyyz$ não pode pertencer a L, porque teríamos mais b's do que a's;

iii) y contém a's e bs. Mas neste caso a palavra $xyyz$ não pode pertencer a L, porque os a's e b's icariam fora de ordem.

O Lema do Bombeamento não se aplica a L. Chegamos a uma contradição, e temos que concluir que L não é regular. ◀

ᛉ .2 Linguagens Livres de Contexto

Linguagens regulares não podem incluir em suas especificações restrições que relacionem uma parte da palavra a outra.

- a linguagem $a^n b^n$, contendo palavras com uma sequência de a's seguida por uma sequência de b's, não é regular.
- a linguagem $(([(\cdots[(x)]\cdots)]))$, onde x é cercado por parênteses e chaves casando corretamente, não é regular
- A linguagem que inclui os pares. "begin-if, end-if", "begin-while, end-while", "begin-comandos, end-comandos", corretamente aninhados, não é regular.

Podemos especificar linguagens desse tipo se permitirmos regras de produção da forma $S \to aSb$ (sempre que um a for gerado, um b também será).

ᛉ .2.1 Gramáticas livres de contexto; definição de linguagem livre de contexto

Definição ᛉ **.40** (gramática e linguagem livre de contexto)**.** Uma gramática é *livre de contexto* se suas produções são da forma

$$A \to s$$

onde A é um *único* símbolo não terminal, e s é uma sequência *qualquer* de símbolos.

Uma linguagem é livre de contexto se é gerada por uma gramática livre de contexto. ⧫

Exemplo ᛉ **.41.** A gramática a seguir é livre de contexto.

$$\begin{aligned} S &\to AXB \\ A &\to < \\ B &\to > \\ X &\to x \\ X &\to S \end{aligned}$$

Esta gramática gera palavras da forma $<<<\cdots x \cdots>>>$. ◀

Definição ᛉ **.42** (derivação mais à esquerda)**.** Uma *derivação mais à esquerda* é feita escolhendo sempre o símbolo não terminal mais à esquerda da palavra para aplicar uma regra de produção. ♦

Exemplo ᛉ **.43.** Se uma gramática inclui, por exemplo as regras

$$A \to abc$$
$$X \to xyz$$

e queremos realizar uma substituição em $mnApqXrs$, há duas opções:

- $mnApqXrs \Rightarrow mnabcpqXrs$ (A foi substituído).
- $mnApqXrs \Rightarrow mnApqxyzrs$ (X foi substituído).

A primeira delas (substituíno o A) é a substituição mais à esquerda, porque o A é o não-temrinal mais à esquerda na frase $mnApqXrs$. ◄

Definição ᛉ **.44** (ambiguidade)**.** Uma gramática G é ambígua se há uma palavra α na linguagem de G que pode ser gerada por mais de uma derivação à esquerda:

$$G \overset{1}{\longrightarrow}^* \alpha$$
$$G \overset{2}{\longrightarrow}^* \alpha,$$

onde as sequências $\overset{1}{\longrightarrow}^*$ e $\overset{2}{\longrightarrow}^*$ são diferentes. ♦

Exemplo ᛉ **.45.** A gramática regular a seguir é ambígua: $S \to aS \mid Sa \mid \varepsilon$

A palavra a pode ser gerada por mais de uma derivação mais à esquerda:

$$S \to aA \to aAa \to aa$$
$$S \to Aa \to Aaa \to aa$$ ◄

Teorema ᛉ **.46.** *Há linguagens livres de contexto que são* inerentemente ambíguas*, isto é, não podem ser geradas por gramáticas não ambíguas.*

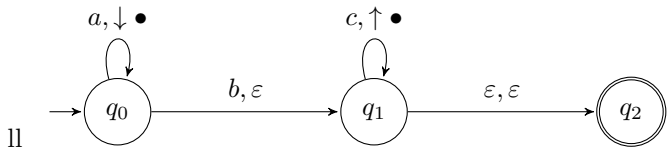
ᛉ .2.2 Autômatos com pilha

Assim como autômatos finitos reconhecem linguagens regulares, *autômatos com pilha* reconhecem linguagens livres de contexto.

Um *autômato com pilha* éum autômato finito equipado com uma pilha de símbolos. Além dos componentes de um autômato finito, há um conjunto Q de símbolos de pilha. O programa do autômato de pilha permite a cada passo empilhar ou desempilhar um símbolo: δ relaciona estados, símbolos da fita e símbolos da pilha com estados e símbolos da pilha. Quando δ indicar que é necessário desempilhar um símbolo e este não estiver no topo da pilha, a palavra é rejeitada.

Definição ᛉ **.47** (autômato com pilha)**.** Um *autômato com pilha* é composto de um alfabeto de fita Σ; um alfabeto de pilha Γ; um conjunto S de estados internos; um estado inicial s; um conjunto de estados iniciais F; e um programa δ, que relaciona triplas estado/símbolo de fita/símbolo de pilha com pares estado/símbolo de fita. ♦

Exemplo ᛉ **.48.** O autômato com pilha a seguir aceita a linguagem $a^n bc^n$. Neste autômato, representamos "empilhar x" como "$\downarrow x$", e "desempilhar y" como "$\uparrow y$. Usamos um só símbolo de pilha, $\bullet$.



A seguir mostramos os componentes do autômato.

$$\begin{aligned}
\Sigma &= \{a, b, c\} \\
\Gamma &= \{\bullet\} \\
S &= \{q_0, q_1, q_2\} \\
s &= q_0 \\
F &= \{q_2\} \\
\delta &= \{(q_0, a, \uparrow \bullet, q_0), (q_0, b, \varepsilon, q_1), \\
&\qquad (q_1, c, \downarrow \bullet, q_1), (q_1, \varepsilon, \varepsilon, q_2)\} .
\end{aligned}$$ ◀

Definição ⍑ **.49** (determinismo em autômato com pilha)**.** Um autômato com pilha é *determinístico* quando seu programa é uma função, e é *não-determinístico* quando seu programa é uma relação, mas não uma função. ⧫

Teorema ⍑ **.50.** *O conjunto de linguagens aceitas por autômatos com pilha* não-determinísticos *é exatamente o das linguagens livres de contexto.*

Teorema ⍑ **.51.** *O conjunto de linguagens aceitas por autômatos com pilha* determinísticos *é subconjunto estrito do das linguagens livres de contexto.*

Teorema ⍑ **.52.** *Uma linguagem livre de contexto inerentemente ambígua não pode ser aceita por qualquer autômato determinístico de pilha.*

⍑ .2.3 Provando que uma linguagem não é livre de contexto

Assim como para linguagens regulares, há um "lema do bombeamento" que permite provar que uma linguagem não é livre de contexto.

Lema ⍑ **.53** (do bombeamento, para linguagens livres de contexto)**.** *Se L é uma linguagem livre de contexto, então existe um comprimento p tal que, se s é palavra de L com $|s| \geq p$, então s pode ser decomposta em $s = uvxyz$, tal que*

i) $uv^n xy^n z \in L$*, para qualquer n;*

ii) $|vy| > 0$*;*

iii) $vxy < p$*.*

Exemplo ⍑ **.54.** A linguagem $L = a^n b^n c^n$ não é livre de contexto. Intuitivamente, conseguimos perceber que um autômato com pilha não poderia reconhecer esta linguagem, porque após empilhar n símbolos contando os a's, deverá desempilhá-los para verificar a quantidade de b's – e depois esquecerá a quantidade n, não podendo verificar a cadeia de c's. Mas isto não é uma demonstração rigorosa. Para construir uma demonstração usaremos o Lema do Bombeamento.

Presumimos que L é livre de contexto, e que vale o Lema do Bombeamento. Seja portanto p o comprimento que determina o Lema para esta linguagem, e suponha que qualquer palavra possa portanto ser decomposta em $uvxyz$ conforme o Lema.

Escolhemos agora $s = a^p b^p c^p$, que claramente pertence a L. Observamos também que o tamanho de s é $3p > p$, logo o Lema determina que esta palavra pode se bombeada.

Dividimos o restante da demonstração em dois casos:

i) se v e y contém o mesmo símbolo, então nem v nem y podem ter a's e b's juntos, nem b's e c's juntos. Mas então $yvvxyyz$ não pode pertencer à linguagem, porque as quantidades de símbolos não estarão balanceadas;

ii) se v e y contém mais de um símbolo, $yvvxyyz$ pode conter a quanidade certa de cada símbolo, mas eles não estarão na ordem correta.

Ao pressupor que L é livre de contexto, mostramos que não vale o Lema do Bombeamento para esta linguagem. Como chegamos a uma contradição, concluímos que L não é livre de contexto. ◀

Exemplo ↾ **.55.** A linguagem $L = \{ww \mid w \in \{0,1\}^*\}$ não é livre de contexto.

Presumimos que L é livre de contexto, e que vale o lema do bombeamento. Seja então p o comprimento determinado pelo Lema para L, e seja

$$s = 0^p 1^p 0^p 1^p$$

De acordo com o Lema, temos $s = uvxyz$. Agora tratamos de três casos, e nos três usaremos o fato de $|vxy| \leq p$:

- a cadeia vxy está toda contida na primeira metade de s. Ao bombearmos s uma vez, obtemos $uvvxyyz$. Bombeamos somente na primeira metade, logo a nova palavra terá dígitos "movidos" da priemira para a segunda metade. *Como $|vxy| \leq p$, a quantidade de símbolos que introduziremos com o bombeamento é no máximo p*, e portanto, os dígitos que podem ter sido movidos são todos uns. Assim, a nova cadeia terá um dígito um no começo da segunda metade, e portanto não pertence a L.

- a cadeia vxy está toda contida na segunda metade de s. Este caso é similar ao anterior, e a palavra bombeada $uvvxyyz$ terá um zero "empurrado" do lado direito para o final da primeira parte da palavra;

- a cadeia vxy passa pela metade de s. Observe que como $|vxy| \leq p$, então teremos necessariamente em vxy uma sequência de uns seguidos de zeros (ou seja, a primeira e a quarta parte da palavra não entram em vxy.
  Realizamos então o bombeamento para baixo: a palavra uxz deveria pertencer a L, mas isto não acontece, porque $uxz = 0^p q^k 0^k 1^p$, com $k < p$.

Havíamos pressuposto que L é livre de contexto e chegamos a uma contradição, logo concluímos que nossa suposição é falsa, e que L não é livre de contexto.

◀

Notas

Normalmente a sintaxe de linguagens de programação é livre de contexto, e sua descrição técnica é feita por uma gramática livre de contexto, em uma forma padrão conhecida como "Forma de Backus-Naur" (BNF). Esta é simplesmente uma transliteração da notação que usamos para gramáticas, de forma que seja representável usando caracteres ASCII[1]

Há outras classes de linguagem e de autômato, que não descrevemos aqui.

Exercícios

Ex. 76 — (Aquecimento) Derive três palavras usando a seguinte gramática:

$$S \to AxB \mid y$$
$$A \to AxB \mid y$$
$$B \to Bx \mid x$$

Ex. 77 — (Aquecimento) Construa um autômato finito que aceite qualquer palavra que comece com "ra" ou com "pa", e que contenha "ro" em algum lugar. O alfabeto é $\{a, b, \ldots, z\}$.

[1] A forma BNF foi desenvolvida quando não havia sequer intenção de acomodar novos símbolos, e ASCII era tudo o que se podia representar em um computador.

Ex. 78 — Construa um autômato finito com o alfabeto $\Sigma = \{0, \ldots, 9\}$ que aceite os números pares que sejam estritamente maiores que 9997 e também os números (pares e ímpares) que sejam estritamente menores que 102.

Ex. 79 — Linguagens regulares são fechadas para interseção? E linguagens livres de contexto?

Ex. 80 — Prove que a linguagem $a^k b^n c^k d^n$ não é livre de contexto.

Ex. 81 — Prove que o problema de decidir se a linguagem gerada por uma gramática linear contém um quadrado é indecidível. Faça o mesmo para gramática livres de contexto.

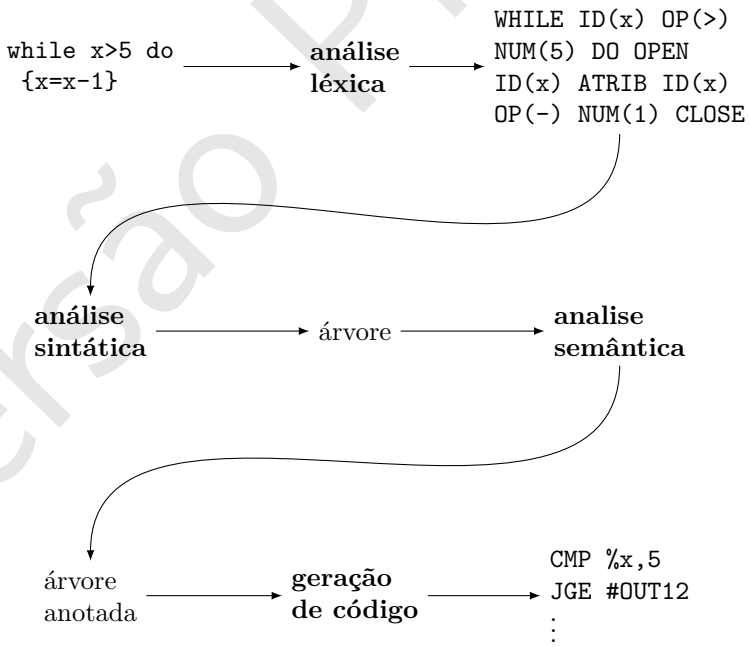
Apêndice ⛩

Visão Geral Rudimentar do Processo de Compilação

Este Apêndice traz uma descrição em alto nível e muito breve do processo de compilação de linguagens de programação. Para uma introdução ao assunto, veja os livros de Kenneth Louden [Lou04] e de Torczon e Cooper [TC13].

Descrevemos apenas o funcionamento de um compilador, deixando de lado interpretadores e tradutores. A descrição que damos é extremamente simplificada.

Um compilador tem como tarefa ler um texto (que recebe na forma de sequência de caracteres) e produzir como saída código executável – ou reportar um erro, se não for possível compilar o programa. Este processo acontece, de maneira geral, em algumas fases:



i) na *análise léxica* o texto do programa é percorrido, identificando-se ali sequências de caracteres como *tokens*. Cada token é representado internamente como um objeto ou estrutura, e não como a sequencia de caracteres originalmente lida.

ii) na *análise sintática* a sequência de tokens é percorrida. Verifica-se a sintaxe do programa e uma representação abstrata dele é construída (usualmente, uma árvore).

iii) na *análise semântica*, os tipos de dados e outras informações semânticas são verificadas; anotações semânticas são adicionadas à árvore abstrata para uso na geração de código. Nesta fase pode ser produzida uma *tabela de símbolos*, contendo identificadores, seus tipos, e quaisquer outras informações sobre eles que possam ser relevantes.

iv) a *geração de código* produz o código executável a partir da árvore abstrata anotada com as informações semânticas.

As fases de compilação não necessariamente acontecem isoladamente no tempo. É comum que um analisador sintático chame o analisador léxico cada vez que precisar de um novo token, por exemplo.

Esta é, claro, uma descrição muito simplificada. Há outras maneiras de compilar programas, e as fases são mais complexas que nossa descrição faz parecer. Em particular, há passos de otimização em diferentes partes das fases de análise semântica e geração de código.

O uso de linguagens formais em compiladores é resumido a seguir.

i) A especificação de tokens é feita por uma gramática regular ou expressões regulares;

ii) A especificação da sintaxe da linguagem é feita por uma gramática livre de contexto.

Apêndice

Alfabeto Grego

Uma vez que letras gregas são abundantes na Matemática, incluímos neste apêndice o alfabeto grego, com a pronúnica de cada letra.

maiúscula	minúscula	pronúncia
A	α	alfa
B	β	beta
Γ	γ	gama
Δ	δ	delta
E	ϵ, ε	épsilon
Z	ζ	zeta
H	η	eta
Θ	θ, ϑ	teta
I	ι	iota
K	$\kappa, \varkappa$	capa
Λ	λ	lambda
M	μ	mi
N	ν	ni
Ξ	ξ	csi
O	o	ômicron
Π	π, ϖ	pi
P	ρ, ϱ	rô
Σ	σ, ς	sigma
T	τ	tau
Υ	υ	úpsilon
Φ	ϕ, φ	fi
X	χ	qui
Ψ	ψ	psi
Ω	ω	ômega

Apêndice

Dicas e Respostas

Resp. (Ex. 1) — Em Haskell:

```
data ExpressaoBool = ValorBool Bool
                   | And       ExpressaoBool ExpressaoBool
                   | Or        ExpressaoBool ExpressaoBool
                   | Not       ExpressaoBool
```

Resp. (Ex. 17) — Há algumas. Alguns exemplos: (i) multiplicação por constante; (ii) qualquer potência fixa; (iii) função constante.

Resp. (Ex. 18) — (i) $\subseteq$ domínio ($\bot = \emptyset$); (ii) $\leq$, comparando um ponto do intervalo (o mais à esquerda, ou mais à direita, ou ponto médio (não é domínio); (iii) $=$ (não é domínio) (iv) cada função f integrável dá para o intervalo $[a, b]$ uma ordem parcial: calcule $\int_a^b f(x)dx$ e compare usando $\leq$ (não é domínio).

Resp. (Ex. 21) — Sejam $\bot_1$ e $\bot_2$ dois menores elementos de um conjunto parcialmente ordenado. Como $\bot_1$ é menor, então $\bot_1 \sqsubseteq \bot_2$. Mas $\bot_2$ é menor também, logo $\bot_2 \sqsubseteq \bot_1$. Como $\sqsubseteq$ é antissimétrica, $\bot_1 = \bot_2$.

Resp. (Ex. 36) — Basta passar o ambiente ρ já com $[p : f]$ quando construir a semântica da definição.

$$\mathcal{C}[\![\textbf{proc } p : c; d]\!] = \lambda\rho\mu \cdot \textbf{let } f = \mathcal{C}[\![c]\!]\mu\rho \textbf{ in } : \\ \mathcal{P}[\![d]\!](\mu[p : f])\rho[p : f]$$

Resp. (Ex. 39) — Cada comando deve retornar um valor adicional, que determina se a continuação deve continuar ou não.

Resp. (Ex. 49) — A regra para $e_1 \geq e_2$ depende de $tB[\![e_1 = e_2]\!]$ e de $\mathcal{TB}[\![e_1 \leq e_2]\!]$. Pode-se usar diretamente o assembly delas:

$$\mathcal{TB}[\![e_1 \geq e_2]\!] = \mathcal{TE}[\![e_1]\!] : \mathcal{TE}[\![e_2]\!] : \texttt{EQ} : \mathcal{TE}[\![e_1]\!] : \mathcal{TE}[\![e_2]\!] : \texttt{LE} : \texttt{NOT} : \texttt{OR}$$

Resp. (Ex. 52) — Suponha que P vale em um estado. Como $P \Rightarrow P'$, e como $\{P'\}c\{Q'\}$, temos consequentemente $\{P\}c\{Q'\}$. Mas $Q' \Rightarrow Q$, logo $\{P\}c\{Q\}$.

Resp. (Ex. 59) — Basta tomar o combinador Ω e incluir um elemento a mais no corpo da abstração: $(\lambda x \cdot zxx)(\lambda x \cdot zxx)$.

Resp. (Ex. 63) — $\lambda ab \cdot ab$ **false**.

Resp. (Ex. 65) — m^n é computado por $\lambda m \cdot \lambda n \cdot nm$.

Resp. (Ex. 67) — (i) Qualquer termo é ponto fixo da função identidade, portanto podemos escolher $F = \lambda x \cdot x$: temos $F_1 Y = (\lambda x \cdot x)Y = Y$. (ii) Tente também $F_2 w = \lambda h(w\ h)$.

Resp. (Ex. 73) — Não!

Resp. (Ex. 81) — Faça reduções usando o problema da correspondência de Post.

Ficha Técnica

Este texto foi produzido inteiramente em LaTeX em sistema Debian GNU/Linux. Os diagramas foram criados sem editor gráfico, usando diretamente o pacote Ti*k*Z. O ambiente Emacs foi usado para edição do texto LaTeX. Os Apêndices foram numerados usando numerais babilônicos com a fonte Santakku, desenvolvida por Sylvie Vanséveren[1].

[1] A fonte pode ser usada livremente, sem alterações, e não pode ser vendida. `http://www.hethport.uni-wuerzburg.de/cuneifont/`.

Bibliografia

[AF01] Martín Abadi e Cédric Fournet. "Mobile values, new names, and secure communication". Em: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2001, pp. 104–115.

[AG99] Martín Abadi e Andrew Gordon. "A Calculus for Cryptographic Protocols: the Spi Calculus". Em: *Information and Computation* 148.1 (1999), pp. 1–70.

[All86] Lloyd Allison. *A Practical Introduction to Denotational Semantics*. Cambridge, 1986. ISBN: 0-521-31423-2.

[Bar12] Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. College Publications, 2012. ISBN: 978-1848900660.

[BK87] J.A. Bergstra e J.W. Klop. "ACPτ: A Universal Axiom System for Process Specification". Em: *CWI Quarterly* 15 (1987), pp. 3–23.

[CM12] Cas Cremers e Sjouke Mauw. *Operational Semantics and Verification of Security Protocols*. Springer, 2012. ISBN: 978-3-540-78635-1.

[End01] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Harcourt/Academic Press, 2001. ISBN: 012-238452-0).

[Fer14] Maribel Fernández. *Programming Languages and Operational Semantics*. Springer, 2014. ISBN: N 978-1-4471-6367-1.

[FG00] Cédric Fournet e Georges Gonthier. "The Join Calculus: A Language for Distributed Mobile Programming". Em: *Proceedings of the Applied Semantics Summer School (APPSEM)*. 2000.

[Fio96] Marcelo P. Fiore. *Axiomatic domain theory in categories of partial maps*. Cambridge, 1996. ISBN: 0-521-60277-7.

[Flo67] R. W. Floyd. "Assigning Meaning to Programs". Em: *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*. Vol. 19. 1967, pp. 19–31.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages: an introduction*. Springer-Vernag, 1979. ISBN: 3-540-90433-6.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: structures and techniques*. MIT Press, 1992. ISBN: 0-262-07143-6.

[Han04] Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. College Publications, 2004. ISBN: 978-0954300654.

[Hed04] Shawn Hedman. *A First Course in Logic: an introduction to model theory, proof theory, computability and complexity*. Oxford, 2004. ISBN: 0-19-852981-3.

[Hen07] Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, 2007. ISBN: 978-0521873307.

[Hil05] Jane Hillson. "Process Algebras for Quantitative Analysis". Em: *Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)*. 2005.

[Hoa69] Charles Antony Richard Hoare. "An Axiomatic Basis for Computer Programming". Em: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN: 978-0131532717.

[HS08] J. R. Hindley e J. P. Seldin. *Lambda Calculus and Combinators: an introduction*. Cambridge University Press, 2008.

[Joh06] Jeffrey D. Ullman John E. Hopcroft Rajeev Motwani. *Introduction to Automata Theory, Languages, and Computations*. 3ª ed. Prentice Hall, 2006. ISBN: 0321455363.

[Lea00] Christopher C. Leary. *A Friendly introduction to Mathematical Logic*. Prentice Hall, 2000.

[Lin11] Peter Linz. *An Introduction to Formal Languages and Automata*. 5ª ed. Jones & Bartlett Learning, 2011. ISBN: 144961552X, 9781449615529.

[Lou04] Kenneth C. Louden. *Compiladores: princípios e práticas*. Thomson, 2004. ISBN: 8522104220.

[Mar02] David Marker. *Model Theory: an introduction*. Springer, 2002. ISBN: 0-387-98760-6.

[Men15] Elliott Mendelson. *Introduction to Mathematical Logic*. CRC Press, 2015. ISBN: 978-1-4822-3778-8.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1980. ISBN: 0-387-10235-3.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN: 0-13-117984-9.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[NN07] Hanne Riis Nielson e Flemming Nielson. *Semantics with Applications: an appetizer*. Springer, 2007. ISBN: 978-1-84628-691-9.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 0-262-16209-1.

[Que96] Christian Queinnec. *Lisp in Small Pieces*. Cambridge, 1996. ISBN: 0-521-54566-8.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010. ISBN: 978-1848822573.

[San12] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge, 2012. ISBN: 978-1-107-00363-7.

[Sha08] Jeffrey Shallit. *A second course in formal languages and automata theory*. Cambridge University Press, 2008. ISBN: 0521865727, 9780521865722, 9780511438356.

[Sip07] Michael Sipser. *Introdução à Teoria da Computação*. 2ª ed. Thomson, 2007. ISBN: 8522104999.

[SLG94] Viggo Stoltenberg-Hansen, Ingrid Lindström e Edward R. Griffor. *Mathematical Theory of Domains*. Cambridge, 1994. ISBN: 978-521-06479-8.

[Sto81] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1981. ISBN: 0-262-69076-4.

[Stu14] Aaron Stump. *Programming Language Foundations*. Wiley, 2014. ISBN: 978-1-118-00747-1.

[TC13] Linda Torczon e Keith D. Cooper. *Construindo Compiladores*. Elsevier, 2013. ISBN: 9788535255652.

[Ten02] Robert D. Tennent. *Specifying Software*. 2002. ISBN: 0-521-00401-2.

[TG08] Franklyn Turbak e David Gifford. *Design Concepts in Programming Languages*. MIT Press, 2008. ISBN: 978-0-262-20175-9.

[Win94] Glynn Winskel. *The Formal Semantics of Programming Languages: an introduction*. The MIT Press, 1994. ISBN: 0-262-23169-7.

Índice Remissivo