8

Programação Funcional

"I must reduce myself to zero."

—Mahatma Gandhi

Durante os anos 30 surgiram dois modelos computacionais abstratos:

1. a máquina de Turing, proposta por Alan Turing; e
2. o λ-cálculo, proposto por Alonzo Church.

A base da máquina de Turing é o conceito de endereços de memória, cujo conteúdo é inspecionado e modificado por instruções. Existe, portanto, uma separação conceitual que coloca de um lado as variáveis e seu conteúdo e de outro os comandos que manipulam essas variáveis, lendo e modificando seu conteúdo.

A máquina de Turing serviu de fundamento para a arquitetura de computadores proposta por John von Neumann, bem como para o paradigma de programação conhecido como *programação imperativa*, que é estudado no Capítulo 9.

A base do λ-cálculo é o conceito de avaliação de funções matemáticas, sendo esse o fundamento essencial da *programação funcional*.

Um programa puramente funcional é uma expressão que caracteriza uma função matemática, juntamente com um elemento do domínio daquela função. A execução de um programa funcional é um processo computacional que permite determinar qual o elemento da imagem da função que corresponde ao elemento do domínio fornecido como parte do programa.

Esse processo se dá por meio de transformações sucessivas do programa, que devem convergir para o resultado desejado. Essas transformações recebem o nome de *reduções*. Devemos alertar o leitor que, apesar do nome, a redução de uma expressão funcional não necessariamente torna essa expressão "mais curta". Sem fazer uso de uma notação específica para programas funcionais, podemos por exemplo considerar a função $f(X) = X^X$. Se $X = 4$, a expressão inicial 4^4 é *reduzida* para o valor 256, embora 4^4 tenha somente dois dígitos e 256 tenha três dígitos.

Neste capítulo, nos concentramos na programação puramente funcional, ou seja, consideramos somente programas puramente funcionais conforme descrito acima e estudamos os elementos que devem fazer parte de uma linguagem de programação para codificar e executar de forma conveniente esses programas.

Alguns conceitos e recursos bastante usuais em programação não fazem parte da programação puramente funcional, como, por exemplo, recursos para entrada e saída de dados e compartilhamento de variáveis. Conforme será detalhado no Capítulo 9, esses conceitos são típicos de programas imperativos. A maioria das linguagens para construção de programas funcionais incorpora alguns desses recursos "extra-funcionais", o que facilita e agiliza a construção de programas, podendo, entretanto, dificultar bastante a análise dos programas construídos.

8.1 FUNDAMENTOS: O λ-CÁLCULO

Um programa funcional é composto por uma única expressão, que descreve detalhadamente uma função e um elemento do domínio daquela função. A redução de um

programa consiste em substituir uma parte da expressão original, obedecendo a certas regras de reescrita.

Em geral, três propriedades essenciais são exigidas de qualquer conjunto de regras de reescrita:

1. **correção:** cada uma das regras, individualmente, deve preservar o significado das expressões. Ou seja, a interpretação de uma expressão antes e depois de uma redução deve ser a mesma.

2. **confluência:** se mais de uma regra for aplicável a uma expressão, a ordem de aplicação deve ser indiferente.

3. **terminação:** o conjunto de regras não deve permitir, para qualquer expressão, que uma seqüência de reduções produza uma expressão idêntica a ela, caso contrário tal expressão não poderia ser calculada.

Correção é uma propriedade semântica. Para que ela seja válida para qualquer expressão, é exigido que alguma convenção para codificar funções e seus parâmetros seja obedecida.

Confluência e terminação são propriedades sintáticas, pois elas independem de como os símbolos no programa são interpretados. A combinação dessas duas propriedades permite que sejam construídas estratégias genéricas para redução de expressões, as quais sejam computacionalmente eficientes e garantidamente resolvam qualquer problema codificado como um programa puramente funcional.

A base formal para construir um conjunto de regras de reescrita com as propriedades acima, conveniente para a programação puramente funcional, é o *λ-cálculo*, desenvolvido pelo matemático Alonzo Church nos anos 30.

O λ-cálculo tem três operações básicas, denominadas **substituição**, **aplicação** e **abstração**.

A substituição indica a troca textual de todas as ocorrências de uma variável em uma expressão por uma outra expressão. Se F e G são expressões, a substituição $F[X \leftarrow G]$ denota a expressão F com todas as ocorrências da variável X trocadas por G.

A aplicação indica a avaliação de uma função para um dado elemento de seu domínio. Se F é a expressão que codifica uma função e A é um elemento de seu domínio, a aplicação $(F)\,A$ denota o valor de F em A.

A abstração é o recurso básico para codificar funções, indicando quais símbolos em uma expressão são variáveis (no sentido matemático). Se F é uma expressão, então $\lambda X\,.\,F$ é o mapeamento de todos os valores que a variável X pode assumir para as substituições das ocorrências de X em F por aqueles valores.

Exemplo 8.1 –

Seja F a expressão $X+5$, sendo X uma variável. Essa expressão é indeterminada, no sentido que a variável X está livre para assumir qualquer valor arbitrário.

- A substituição $[X \leftarrow 4]$ denota a troca das ocorrências de X por 4. O resultado da substituição $F[X \leftarrow 4]$ é a expressão $4+5$.
- A abstração $\lambda X\,.\,F$ – ou seja, $\lambda X.(X+5)$ – denota a função matemática que, para cada valor de X, associa o valor $X+5$. Nessa expressão, a variável X não está mais livre para assumir um valor arbitrário: ela denota a aplicabilidade da expressão $X+5$ a *todos* os valores possíveis para X.
- A aplicação $(\lambda X.(X+5))4$ "particulariza" a expressão genérica $\lambda X.(X+5)$ para o valor $X=4$. Essa última expressão é um típico programa puramente funcional. Um sistema de reduções deve permitir que se produza a seqüência de expressões $(\lambda X.(X+5))4 \Rightarrow 4+5 \Rightarrow 9$.

□

Substituições, aplicações e abstrações compõem expressões a partir de constantes e variáveis. No λ-cálculo básico, as variáveis e constantes não têm tipos definidos. Seja $\mathcal{C} = \{c_1, c_2, ...\}$ um conjunto de constantes e $\mathcal{V} = \{X_1, X_2, ...\}$ um conjunto de variáveis. Um λ-*termo* é definido indutivamente da seguinte maneira:

- qualquer elemento de $\mathcal{C}$ é um λ-termo;
- qualquer elemento de $\mathcal{V}$ é um λ-termo;
- se $X \in \mathcal{V}$ e F é um λ-termo, então $[X \leftarrow F]$ também é um λ-termo;

- se F e G são λ-termos, então a aplicação $(F)\,G$ também é um λ-termo;
- se F é um λ-termo e $X \in \mathcal{V}$ é uma variável, então a abstração $\lambda X\,.\,F$ também é um λ-termo.

O conjunto de *variáveis livres* em um λ-termo é o conjunto de variáveis que ocorrem "sem um λ à sua frente". Formalmente, esse conjunto é construído assim. Seja $\mathcal{L}_F$ o conjunto de variáveis livres do λ-termo F:

- se $F = X \in \mathcal{V}$, então $\mathcal{L}_F = \{X\}$;
- se $F = c \in \mathcal{C}$, então $\mathcal{L}_F = \{\}$;
- se $F = (G)H$, então $\mathcal{L}_F = \mathcal{L}_G \cup \mathcal{L}_H$;
- se $F = \lambda X\,.\,G$, então $\mathcal{L}_F = \mathcal{L}_G\text{-}\{X\}$;

Um λ-termo é chamado de *fechado* se o seu conjunto de variáveis livres for vazio. Os recursos com os quais contamos para "fechar" um λ-termo são a abstração (conforme visto acima, na definição de λ-termos fechados) e as reduções (conforme visto a seguir).

O λ-cálculo é extremamente econômico. Somente duas regras de redução são definidas, denominadas de *redução* α e *redução* β:

1. α : $\lambda X_1\,.\,F \Rightarrow \lambda X_2\,.\,F[X_1 \leftarrow X_2]$, ou seja, a troca de nomes de variáveis é permitida;
2. β : $\lambda(X\,.\,F)\,G \Rightarrow F[X \leftarrow G]$, ou seja, uma abstração seguida de uma aplicação correspondem à definição de uma função (abstração) e de sua avaliação em um ponto específico (aplicação).

Essas duas reduções, entretanto, são de uma engenhosidade admirável:

- O aninhamento de abstrações permite definir funções com múltiplos argumentos.
- O sistema de reduções composto pelas reduções α e β apresenta as propriedades de confluência e terminação.

- Finalmente, conforme demonstrado por Church, os números naturais e todas as funções computáveis podem ser representados utilizando o λ-cálculo com somente essas duas regras de redução.

As demonstrações de todos esses resultados, conforme pode-se imaginar, não são imediatas. Ao leitor interessado nos aspectos matemáticos fundamentais da teoria da computação, recomendamos consultar [15] para uma exposição clara das demonstrações desses resultados.

A conseqüência prática desses resultados é a garantia de satisfação das três propriedades básicas mencionadas anteriormente (correção – ao menos para as funções computáveis – confluência e terminação).

Segundo a teoria do λ-cálculo, definimos um programa funcional (juntamente com seus dados de entrada) como uma expressão representando um λ-termo fechado. A execução de um programa funcional consiste na aplicação exaustiva das regras de redução. A resposta do programa é o λ-termo resultante do processo de redução exaustiva.

Apresentada dessa forma, porém, dificilmente a programação funcional poderia ser considerada como mais do que um exercício teórico de reconstrução matemática dos fundamentos da computação. Para que possam ser construídos programas de porte e interesse "práticos", os seguintes elementos são em geral pré-construídos em uma linguagem de programação funcional:

- Tipos de dados básicos, já codificados utilizando uma notação adequada. Em geral, esses tipos são:
  - números inteiros;
  - números de ponto flutuante;
  - valores booleanos ("verdadeiro" e "falso");
  - constantes simbólicas;
  - caracteres e cadeias de caracteres;
  - listas.

- Funções e relações usuais para esses tipos de dados (aritmética para os tipos numéricos, relações de ordem para os tipos numéricos e baseados em caracteres, concatenação e seleção de elementos para listas, etc.) são também predefinidas e codificadas utilizando uma notação adequada.

- Declaração de subexpressões com nomes, caracterizando dessa forma algo semelhante ao conceito usual de procedimentos.

- Uma estratégia cuidadosamente construída para desenvolver seqüências de reduções, visando maximizar a eficiência das reduções para a maioria dos programas funcionais.

8.2 VARIÁVEIS E TIPOS DE DADOS

Conforme mencionado acima, os tipos de dados em programação funcional são na realidade "atalhos" para facilitar a construção de programas. De um ponto de vista matematicamente "purista", eles poderiam ser construídos passo a passo cada vez que um programa fosse construído, mas evidentemente essa não é uma alternativa sensata para construir programas grandes.

Os seguintes tipos de dados fazem parte da maioria das linguagens de programação funcional, juntamente com as respectivas operações e relações implementadas como funções:

- números inteiros;

- números de ponto flutuante;

- valores booleanos ("verdadeiro" e "falso");

- constantes simbólicas;

- caracteres e cadeias de caracteres.

Adicionalmente, as linguagens de programação funcional apresentam também como tipo predefinido listas. As operações predefinidas para listas são concatenação

e seleção de elementos, a partir das quais outras operações podem ser definidas. Outros tipos compostos de dados (árvores, grafos, etc.) em geral ficam a cargo do programador, para serem construídos com base em listas.

Um terceiro tipo de dados presente nas linguagens de programação funcional é o tipo *função*. Esse tipo permite que subexpressões recebam um nome, possibilitando dessa forma a construção de procedimentos (estamos propositalmente evitando nesse capítulo de chamar a construção de procedimentos de um processo de abstração, para evitar confusão com a operação de abstração do λ-cálculo).

Algumas linguagens de programação funcional são fortemente tipadas, ou seja as variáveis são criadas com tipos definidos e esses tipos são verificados antes de qualquer vinculação de valores. A maioria das linguagens de programação funcional, entretanto, obedece à formulação original do λ-cálculo, de forma que as variáveis adotam livremente os tipos do que estiver sendo atribuído a elas.

8.3 EXPRESSÕES E PROGRAMAS

Um programa funcional é uma única expressão. Para facilitar a construção de programas funcionais, essa expressão pode ser "quebrada" em subexpressões, utilizando o recurso do tipo de dados *função*: são criados nomes de funções, que são variáveis cujos valores são funções. Quando essas variáveis são colocadas dentro de uma expressão maior, os seus valores são substituídos dentro daquela expressão.

Todos os conceitos vistos até aqui devem ser melhor esclarecidos se tomarmos alguns pequenos exemplos concretos.

Exemplo 8.2 – **Polinômio:** dado um número inteiro $X_0 = 4$ e o polinômio do segundo grau $X^2 + X + 1$, calcular o valor do polinômio para $X = X_0$.

Esse programa funcional é a abstração do polinômio, aplicada ao valor X_0. Ou seja,

$$\lambda X.(X^2 + X + 1)4$$

Deve ser observado que, nesse exemplo, já estamos assumindo como preexistentes as operações aritméticas para inteiros. Sendo assim, a redução β nos fornece o resultado esperado:

$$\lambda X.(X^2 + X + 1)4 \Rightarrow 4^2 + 4 + 1 \Rightarrow 21$$

A última redução na realidade embute diversas reduções, referentes às avaliações dos operadores aritméticos presentes na expressão original.

O polinômio pode ser "armazenado" em uma variável como um procedimento, que pode ser "invocado" a partir de uma expressão:

$$\text{polinômio} = \lambda X.(X^2 + X + 1)$$

Esse λ-termo pode então ser utilizado, resultando na mesma seqüência de reduções acima:

$$\text{polinômio } 4 = \lambda X.(X^2 + X + 1)4 \Rightarrow 4^2 + 4 + 1 \Rightarrow 21$$

□

Exemplo 8.3 – **Fatorial:** dado um número natural $X_0 = 4$, calcular o seu fatorial.

Esse exemplo é semelhante ao anterior, exceto que a função fatorial é recursiva, sendo assim a sua abstração deve expressar o caso base e a generalização indutiva. O caso base é uma constante – se $X = 0$ a função deve retornar o valor 1. Para a generalização indutiva, é necessário referenciar explicitamente a mesma abstração para um novo valor dependente de X – o fatorial de X quando $X > 0$ vale $X \times$ fatorial de X-1.

O recurso de armazenar λ-termos em variáveis permite expressar o fatorial de forma bastante compacta:

$$\text{fatorial} = \lambda X.(\text{se } X = 0 \text{ então } (1) \text{ senão } (\text{fatorial}(X\text{-}1)))$$

A utilização desse λ-termo para o valor 4 pode então ser expressa como:

$$
\begin{aligned}
\text{fatorial } 4 &\Rightarrow 4 \times \text{fatorial } 3 \\
&\Rightarrow 4 \times 3 \times \text{fatorial } 2 \\
&\Rightarrow 4 \times 3 \times 2 \times \text{fatorial } 1 \\
&\Rightarrow 4 \times 3 \times 2 \times 1 \times \text{fatorial } 0 \\
&\Rightarrow 4 \times 3 \times 2 \times 1 \times 1 \\
&\Rightarrow 4 \times 3 \times 2 \times 1 \\
&\Rightarrow 4 \times 3 \times 2 \\
&\Rightarrow 4 \times 6 \\
&\Rightarrow 24
\end{aligned}
$$

Existe aqui, implícita, a definição de uma estratégia para a seqüência de reduções. A função fatorial é considerada "prioritária", e nada mais é avaliado enquanto ela não desaparece do λ-termo que está sendo reduzido. Para formar uma imagem visual, é como se fôssemos mergulhando e nos aprofundando no λ-termo inicial (fatorial 4) até atingir o caso base da definição do fatorial, para só então passar a efetuar as multiplicações ao "retornar gradativamente à superfície" (figura 8.3).

Alternativamente, poderíamos efetuar multiplicações toda vez que tivéssemos uma oportunidade. Nesse caso, teríamos a resposta desejada assim que a última ocorrência do termo "fatorial" fosse reduzida (figura 8.3):

fatorial 4		24
$\Downarrow$		$\Uparrow$
$4 \times$ fatorial 3		4×6
$\Downarrow$		$\Uparrow$
$4 \times 3 \times$ fatorial 2		$4 \times 3 \times 2$
$\Downarrow$		$\Uparrow$
$4 \times 3 \times 2 \times$ fatorial 1		$4 \times 3 \times 2 \times 1$
$\Downarrow$		$\Uparrow$
$4 \times 3 \times 2 \times 1 \times$ fatorial 0	$\Rightarrow$	$4 \times 3 \times 2 \times 1 \times 1$

Figura 8.1 Reduções "de dentro para fora" ou "a priori" – fatorial

$$
\begin{aligned}
\text{fatorial } 4 &\Rightarrow 4 \times \text{fatorial } 3 \\
&\Rightarrow 4 \times 3 \times \text{fatorial } 2 \\
&\Rightarrow 12 \times \text{fatorial } 2 \\
&\Rightarrow 12 \times 2 \times \text{fatorial } 1 \\
&\Rightarrow 24 \times \text{fatorial } 1 \\
&\Rightarrow 24 \times 1 \times \text{fatorial } 0 \\
&\Rightarrow 24 \times \text{fatorial } 0 \\
&\Rightarrow 24 \times 1 \\
&\Rightarrow 24
\end{aligned}
$$

A primeira estratégia é denominada **de dentro para fora** ou **a priori**. A segunda estratégia é denominada *de fora para dentro* ou *sob demanda*. Elas serão melhor explicadas na próxima seção.

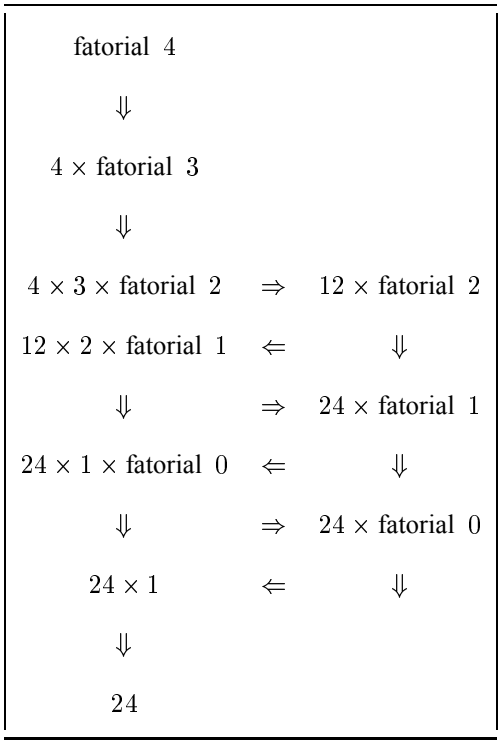

Figura 8.2 Reduções "de fora para dentro" ou "sob demanda" – fatorial

□

Exemplo 8.4 – **Manipulação de listas:** dada uma lista de números inteiros, fornecer:

1. o primeiro elemento da lista;
2. o último elemento da lista;
3. a lista contendo os mesmos elementos dispostos na ordem inversa.

Assumimos aqui como preexistentes o tipo de dados *lista* e algumas operações sobre listas. Para esse exemplo específico, utilizamos as operações de agregação e de concatenação de listas. Utilizaremos também a comparação literal de listas:

- denotamos uma lista como uma seqüência de elementos delimitados por colchetes e separados por vírgula. Por exemplo, a lista composta pelos inteiros consecutivos de 1 a 5 é escrita como $[1, 2, 3, 4, 5]$.
- a construção de uma lista ocorre, na verdade, pela composição de uma função de *agregação* de elementos, que denotamos como $(:)$. Assim, uma notação alternativa para a lista acima – que indica como ela foi construída – é $(1: (2: (3: (4: (5: [])))))$.
- a *concatenação* de duas listas é uma terceira lista composta pelos elementos da primeira lista seguidos dos elementos da segunda lista. Denotamos essa operação como $(++)$. Por exemplo, o resultado da operação $[1, 2, 3] ++ [4, 5]$ é a lista $[1, 2, 3, 4, 5]$.
- a *comparação literal* de duas listas é denotada como $(==)$. Se duas listas forem literalmente idênticas, o resultado da comparação é "verdadeiro", caso contrário é "falso".

O primeiro elemento de uma lista é definido para listas com pelo menos um elemento. Ele é expresso pelo seguinte λ-termo:

$$\text{primeiro} = \lambda(X : Lista).X$$

O último elemento de uma lista é um pouco mais complexo. Ele é expresso pelo seguinte λ-termo:

$$\text{último} = \lambda(X\colon Lista).(\text{se } \ Lista == [] \ \text{ então } \ X \ \text{ senão } \ (\text{último} Lista))$$

Para inverter uma lista, precisamos de um λ-termo como o apresentado abaixo:

$$\text{inverte} = \lambda(X\colon Lista).(\text{se } \ Lista == [] \ \text{ então } \ [X] \ \text{ senão } \ (\text{inverte} Lista + +[X]))$$

A aplicação desses λ-termos à lista $[1,2,3,4,5]$ resulta nas seqüências de reduções delineadas abaixo:

- primeiro $[1,2,3,4,5] \Rightarrow 1$.
- último $[1,2,3,4,5] \Rightarrow$ último $[2,3,4,5] \Rightarrow$ último $[3,4,5] \Rightarrow$ último $[4,5] \Rightarrow$ último $[5] \Rightarrow 5$.

A aplicação do λ-termo inverte depende da estratégia de redução utilizada. Se a estratégia for **de dentro para fora (a priori)**, o resultado é:

- inverte $[1,2,3,4,5] \Rightarrow$ inverte $[2,3,4,5]++[1] \Rightarrow$ inverte $[3,4,5]++[2]++[1] \Rightarrow$ inverte $[4,5]++[3]++[2]++[1] \Rightarrow$ inverte $[5]++[4]++[3]++[2]++[1] \Rightarrow [5]++[4]++[3]++[2]++[1] \Rightarrow [5,4]++[3]++[2]++[1] \Rightarrow [5,4,3]++[2]++[1] \Rightarrow [5,4,3,2]++[1] \Rightarrow [5,4,3,2,1]$.

Se a estratégia for *de fora para dentro (sob demanda)*, o resultado é:

- inverte $[1,2,3,4,5] \Rightarrow$ inverte $[2,3,4,5]++[1] \Rightarrow$ inverte $[3,4,5]++[2]++[1] \Rightarrow$ inverte $[3,4,5]++[2,1] \Rightarrow$ inverte $[4,5]++[3]++[2,1] \Rightarrow$ inverte $[4,5]++[3,2,1] \Rightarrow$ inverte $[5]++[4]++[3,2,1] \Rightarrow$ inverte $[5]++[4,3,2,1] \Rightarrow [5]++[4,3,2,1] \Rightarrow [5,4,3,2,1]$.

□

8.4 ESTRATÉGIAS PARA REDUÇÕES

As propriedades de confluência e terminação garantem que qualquer seqüência de reduções deve levar ao mesmo resultado final. Diferentes estratégias para organizar essas seqüências, entretanto, podem levar a seqüências menores para λ-termos específicos.

Uma área de pesquisas interessante e desafiadora é o desenvolvimento de estratégias para construir eficientemente seqüências de reduções para λ-termos genéricos, e/ou o desenvolvimento de estratégias especializadas e identificadores para decidir a melhor estratégia a utilizar dependendo de alguma característica de formação de um λ-termo.

As duas estratégias fundamentais e mais freqüentemente utilizadas nas linguagens para programação funcional são as estratégias apresentadas nos exemplos acima, denominadas de **de dentro para fora (a priori)** e **de fora para dentro (sob demanda)**. Os termos em inglês utilizados para denominar essas duas estratégias são *eager* (gananciosa) e *lazy* (sossegada).

As estratégias em si puderam ser depreendidas a partir dos exemplos apresentados. O motivo dessa curiosa denominação pode ser apreciado se fizermos uma analogia com estratégias observadas na prática para colecionar livros e formar uma biblioteca particular:

- É freqüente encontrarmos bibliófilos que apreciam colecionar livros para leitura posterior. Muitos desses bibliófilos gostam de sonhar com um dia em que, aposentados, poderão finalmente ler todos os livros adquiridos ao longo da vida. Em casos extremos, os bibliófilos não lêem nada até o dia que se aposentam.

- Outro perfil encontrado – especialmente nos corredores das universidades – são compradores de livros que contam com recursos financeiros escassos. Esses compradores, para reduzir gastos, freqüentemente se auto-infligem disciplinas rigorosas, como, por exemplo, nunca adquirir um novo livro sem ter explorado completamente os anteriormente comprados.

A primeira estratégia é "gananciosa" e a segunda estratégia é "sossegada", utilizando a denominação acima. Nenhuma das estratégias é universalmente melhor que a outra: um "ganancioso" pode ter muitos livros para adquirir e passar a vida toda comprando livros, sem nunca ler nada. Um "sossegado", por outro lado, pode um dia se deparar com um livro muito extenso e complexo e ficar "preso" naquele livro.

O mesmo ocorre com os programas funcionais. Se a estratégia adotada for **de dentro para fora** e existir no λ-termo inicial uma função que nunca desapareça utilizando reduções, a tentativa repetida de aplicar aquela função a novos termos antes de qualquer outra redução pode levar a um laço de repetições infinitas, sem a apresentação de quaisquer resultados intermediários que poderiam ser úteis. Da mesma forma, se a estratégia adotada for **de fora para dentro** e surgir em algum λ-termo produzido durante as reduções alguma função cuja aplicação seja computacionalmente difícil e demorada, a computação das reduções pode ficar "presa" naquela função.

De maneira geral, as reduções geradas **de fora para dentro** permitem colecionar resultados intermediários, mas gerar reduções **de dentro para fora** é mais fácil de implementar – dessa forma permitindo a construção de interpretadores e compiladores mais compactos.

8.5 A LINGUAGEM HASKELL

A linguagem Haskell surgiu em 1987, como resultado das pesquisas principalmente de Simon Peyton Jones. Desde então, essa linguagem de programação tem se firmado como uma excelente implementação dos conceitos de programação funcional. Naturalmente existem outras linguagens para programação funcional disponíveis, que são inclusive melhor conhecidas que Haskell. Dentre essas linguagens, podemos destacar as linguagens SML e Lisp.

Escolhemos a linguagem Haskell para destacar neste capítulo porque ela implementa de maneira mais cuidadosa os conceitos originais do λ-cálculo. Comparando com SML e LISP, Haskell é a única linguagem que apresenta uma estratégia para redução de λ-termos **de fora para dentro (sob demanda)**, à parte da Lazy ML.XS

Essa linguagem apresenta muitos recursos, os quais possibilitam que programas de porte relativamente grande sejam executados com eficiência. Assim, ela tem sido utilizada para resolver com sucesso problemas significativos na indústria. Mais informações sobre as aplicações da linguagem Haskell podem ser encontradas em `http://www.haskell.org`.

Nessa seção apresentaremos somente a implementação em Haskell dos exemplos vistos na seção anterior. Recomendamos ao leitor interessado que instale um interpretador Haskell em seu computador, insira o código apresentado abaixo no interpretador e efetue as reduções das expressões inseridas. Essa é uma boa forma de conduzir o leitor a resultados práticos com base na teoria apresentada.

Em Haskell, um λ-termo representando uma função recebe sempre um nome. Associado a esse nome, é indicado o **tipo da função**, ou seja os tipos respectivamente dos elementos do domínio e da imagem da função. Por exemplo, a função `polinômio` da seção anterior recebe um número inteiro e responde com outro número inteiro. O tipo dessa função é declarado em Haskell como:

- `polinômio :: Integer -> Integer`

O λ-termo propriamente dito é declarado de forma muito semelhante à apresentada anteriormente. O λ-termo polinômio $= \lambda X.(X^2 + X + 1)$ é escrito em Haskell como:

- `polinômio x = x * x + x + 1`

O programa em Haskell que implementa o cálculo do polinômio apresentado anteriormente é composto por essas duas declarações. Esse programa é ativado determinando o valor inteiro ao qual se quer aplicar a função:

- `polinômio 4`.

A redução dessa expressão resulta no valor esperado `21`.

A função `fatorial` é implementada em Haskell como:

- `fatorial :: Integer -> Integer`
- `fatorial x = if x==0 then 1 else x * fatorial (x - 1)`

A ativação

- `fatorial 4`

produz o resultado esperado `24`.

A função `primeiro` tem como domínio o conjunto das listas de inteiros, e fornece como respostas valores inteiros. O tipo dessa função é declarado como:

- `primeiro :: [Integer] -> Integer`.

A notação `[Integer]` denota o conjunto das listas cujos elementos são números inteiros. A notação `Integer`, como nos exemplos anteriores, denota o conjunto dos números inteiros.

A declaração correspondente ao λ-termo que caracteriza essa função é:

- `primeiro (x:lista) = x`.

De maneira semelhante, as declarações de tipos e dos λ-termos correspondentes às funções `último` e `inverte` são:

- `último :: [Integer] -> Integer`
- `último (x:lista) = if lista==[] then x else último lista`
- `inverte :: [Integer] -> [Integer]`
- `inverte (x:lista) = if lista==[] then [x] else (inverte lista) ++ [x]`

8.6 LEITURA RECOMENDADA

Os aspectos teóricos da programação funcional, principalmente relativos ao λ-cálculo, são apresentados de forma resumida em [16]. Uma apresentação bastante mais extensa pode ser encontrada em [15]. Em [43] encontramos outra exposição dos fundamentos de processos computacionais com base no λ-cálculo.

Em [64] encontramos uma apresentação mais "leve" da programação funcional, que exige menos maturidade matemática do leitor. Esse material pode ser utilizado como um texto introdutório ao assunto.

Existe um endereço WWW dedicado à linguagem Haskell, que é `http://www.haskell.org`. Diversas implementações dessa linguagem podem ser encontradas a partir desse endereço, incluindo o interpretador / compilador HUGS, que pode ser instalado em diferentes plataformas (Unix/Linux, Windows, MacOS, etc.).

8.7 EXERCÍCIOS

1. Construa um λ-termo correspondente à função de Fibonacci. A função de Fibonacci é uma função definida para os números naturais, tal que (1) Fib(0) = 0, (2) Fib(1) = 1 e (3) Fib(n+2) = Fib(n) + Fib(n+1) para n ≥ 0. Implemente a sua solução como um programa em Haskell e teste o programa.

2. Construa um λ-termo correspondente à função $f(X) = \sum_0^X$, sendo X um número natural. Implemente a sua solução como um programa em Haskell e teste o programa.

3. Construa um λ-termo correspondente à função que, dada uma lista de números inteiros, retorna o maior elemento dessa lista. Implemente a sua solução como um programa em Haskell e teste o programa.

4. Construa um λ-termo correspondente à função que, dada uma lista de números inteiros, retorna uma lista contendo os mesmos elementos da lista original colocados em ordem crescente. Implemente a sua solução como um programa em Haskell e teste o programa.

5. Todas as funções vistas nos exemplos e exercícios até aqui são funções com um único argumento. Funções com múltiplos argumentos podem ser expressas pelo aninhamento de abstrações. Além disso, podemos construir em Haskell listas de listas, que são o recurso fundamental para a construção de tipos de dados mais complexos – como, por exemplo, árvores e grafos. Construa um λ-termo correspondente à função que, dada uma lista de números inteiros e um número inteiro, retorna uma lista de listas de números inteiros, cujos elementos são duas listas. A primeira dessas listas deve conter os elementos da lista originalmente fornecida, que são menores que o número fornecido, e a segunda lista deve

conter os elementos da lista original, que são maiores que ou iguais ao número fornecido. Implemente a sua solução como um programa em Haskell e teste o programa.

6. Construa um λ-termo correspondente à função que, dada uma lista de números inteiros, retorna uma lista contendo os mesmos elementos da lista original colocados em ordem crescente. Utilize os resultados do exercício anterior para que o método de ordenação implementado seja o *quicksort*. Implemente a sua solução como um programa em Haskell e teste o programa.