Pedro Vasconcelos

Implementação de uma Linguagem Funcional usando Combinadores Compactos

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 1998

Pedro Vasconcelos

Implementação de uma Linguagem Funcional usando Combinadores Compactos

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 1998

Índice

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Programação funcional e implementações

O paradigma de programação funcional baseia-se na representação da computação como a aplicação de *funções* a *valores*. Um programa funcional consiste num conjunto de *definições de funções*; uma computação funcional consiste na redução duma *expressão* que representa a resposta desejada pelo programador.

A evolução na concepção de linguagens funcionais levou a linguagens mais "puras", mais próximas da teoria de λ-calculus, uma álgebra de computação baseada em funções como regras de re-escrita de termos. Como consequência estas linguagens ficaram mais distantes do modelo de execução dos computadores e linguagens convencionais.

As linguagem funcionais modernas partilham algumas características comuns:

- *Funções como "objectos de 1ª classe"* da linguagem: as funções podem ser criadas dinamicamente, passadas como argumentos e retornadas como valores por outras funções. Isto permite a definição de *funções de nível superior* que abstraem padrões de computação comuns.

- *Ausência de efeitos colaterais* à redução de uma expressão: não há modificações a um "estado global" como atribuições a variáveis ou memórias.

- *Transparência referencial*: uma expressão denota um valor que é imutável, *i.e.*, não depende de um "estado" que varia com o tempo. Isto permite "trocar iguais por iguais", ou seja, substituir uma expressão por outra que denota o mesmo valor.

- *Tipos estruturados* (também designados *tipos algébricos*) como listas, árvores ou simples enumerações.

- *Semântica de redução não-estrita* (ou equivalente: redução com "*lazy evaluation*"): uma expressão só é reduzida quando o seu resultado é efectivamente necessário para a progressão da computação; em particular, os argumentos duma função não são reduzidos antes de a invocar, mas apenas quando (e se) for necessário.

  A semântica não-estrita permite manipular *estruturas de dados infinitas* (como listas ou árvores infinitas); mais geralmente, permite separar a *descrição* duma computação conceptualmente infinita duma sua *execução* parcial que será necessariamente finita.

Estas características foram introduzidas gradualmente em várias linguagens como SASL (Turner, 1979b), ML e Miranda, por exemplo.

Mais recentemente, a comunidade de programação funcional fez convergir na linguagem Haskell a sistematização das linguagens funcionais modernas. Baseada em concepções nucleares bem-fundadas e incluindo as melhores características das experiências passadas, a linguagem Haskell é orientada de forma a servir simultaneamente como linguagem de ensino e promoção da programação funcional, e como uma linguagem de desenvolvimento avançada para projectos reais (Peterson & Hammond, 1996).

A par dos desenvolvimentos na concepção de linguagens funcionais evoluíram também as técnicas usadas na sua implementação. A programação funcional é um exemplo paradigmático duma área nas ciências de computação onde a interligação estreita entre teoria e prática levou a um crescimento frutuoso de ambas: os desenvolvimentos na teoria levam a aplicações práticas directas e as necessidades da implementação suscitam novas abordagens teóricas (Peyton Jones et al., 1992b).

A implementação de linguagens puramente funcionais teve desenvolvimentos importantes na última década: a *redução de grafos*, uma técnica de implementação que remonta aos finais da década de 70 (Turner, 1979b), associada à tradução para "*super-combinadores*" permitiu a execução eficiente de linguagens não-estritas em computadores de arquitecturas convencionais (Peyton Jones, 1987).

Em vez de investir em computadores com "*hardware*" específico, os implementadores de linguagens funcionais (e, em geral, de todas as "linguagens simbólicas") aprofundaram a transposição para computadores com arquitecturas de uso genérico (Kogge, 1991).

Esta filosofia tornou-se particularmente importante no contexto da "revolução RISC" nos finais da década de 80, baseada na ideia de optimizar a execução em "*hardware*" das instruções genéricas mais usadas (Henessy & Patterson, 1996). Os avanços de *performance* dos micro-processadores durante última década tornaram injustificado o desenvolvimento de processadores com arquitecturas específicas, sob pena de estes se tornarem obsoletos demasiado depressa.

Uma das contribuições mais importantes para a implementação de linguagens funcionais não-estritas em "*hardware*" genérico foi a "*G-machine*", uma *máquina abstracta* concebida para executar código de redução de grafos (Peyton Jones, 1987). O código da "*G-machine*" poderia facilmente ser emulado por um computador convencional ou mesmo ser transposto directamente para linguagem-máquina (Peyton Jones, 1987, cap. 19).

1.2 Resumo da dissertação

Nesta dissertação apresentamos uma implementação duma linguagem funcional minimal com semântica não-estrita usando uma nova tradução para "combinadores compactos" apresentada em (Broda, 1996) (no original: combinadores-$\mathcal{C}\ell(\mathcal{K})$).

Esta tradução apresenta algumas vantagens em relação à tradução clássica para combinadores-SK (Turner, 1979a; Turner, 1979b), nomeadamente reduzindo o número de combinadores introduzidos e tornando a redução em combinadores compactos particularmente eficiente. É também conceptualmente mais simples do que a tradução em super-combinadores e preserva propriedades do programa original.

O nosso objectivo é tomar partido desta tradução e gerar código em linguagem C para redução de grafos de um programa funcional.

No capítulo 2 apresentamos algumas definições e resultados na teoria do λ-calculus e de lógica combinatória em que se baseia o trabalho efectuado. Em particular, apresentamos um algoritmo de abstracção em combinadores-$\mathcal{C}\ell(\mathcal{K})$ que é um refinamento do algoritmo original em (Broda, 1996).

No capítulo 3 apresentamos a nossa linguagem funcional minimal e definimos a tradução em combinadores de um programa completo nesta linguagem. Discutimos também algumas propriedades desta tradução: complexidade e preservação de reduções.

No capítulo 4 abordamos a transposição dum programa em combinadores para a execução em linguagem C por meio do código de uma máquina abstracta, a *máquina*-ΦG.

A geração de código para a máquina-ΦG é discutida no capítulo 5, bem como a integração de algumas optimizações simples no nosso modelo de execução.

O capítulo 6 apresenta algumas conclusões finais e discussões sobre trabalho futuro.

Finalmente, os apêndices A e B descrevem, respectivamente, o conjunto de instruções da máquina-ΦG e a implementação de optimização de última chamada em linguagem C (essencial no modelo de execução apresentado no capítulo 4).

Capítulo 2

Preliminares

Neste capítulo apresentamos algumas definições e resultados da teoria subjacente à programação funcional (o λ-calculus) e à nossa implementação em particular (tradução em combinadores-$\mathcal{C}\ell(\mathcal{K})$). Todos os resultados apresentados sem demonstração estão provados na bibliografia (Hankin, 1994; Broda, 1996).

2.1 Convenções de notação

Começamos por fixar algumas notações para sequências, listas e tuplos de objectos.

2.1.1 Sequências

Usamos *sequências* para representar cadeias ordenadas de elementos; ao contrário das listas, as sequências não podem ser imbricadas:

- "ϵ" representa a sequência vazia;
- representamos uma sequência em extensão por justaposição dos seus elementos; por exemplo, a sequência com os elementos b, c, d (por esta ordem) é bcd[1];
- o operador "$\cdot$" concatena duas sequências, por exemplo: $a \cdot bcd = abcd$.

[1] Note-se que desta convenção resulta uma ambiguidade entre a representação do elemento a e a sequência de um só elemento a. Em geral esta ambiguidade não acarreta qualquer problema, tornando-se claro pelo contexto qual o sentido a adoptar.

2.1.2 Listas

Usamos *listas* para representar sequências homogéneas de elementos (*i.e.*, termos, instruções de máquina abstracta, etc.) com uma sintaxe idêntica à da linguagem Haskell:

- `[]` representa a lista vazia;
- `[`$a_1, a_2, \ldots, a_n$`]` representa uma lista em extensão;
- `[`$a_i \mid i = 1, \ldots, n$`]` representa uma lista em compreensão;
- o operador ":" concatena um elemento à cabeça duma lista, por exemplo: a : `[`b, c, d`]` = `[`a, b, c, d`]`;
- o operador "++" concatena duas listas, por exemplo: `[`a, b`]` ++ `[`c, d`]` = `[`a, b, c, d`]`.

2.1.3 Tuplos

Um *n-tuplo* $(a_1, a_2, \ldots, a_n)$ representa um elemento de um produto cartesiano $A_1 \times A_2 \times \ldots \times A_n$, onde $a_i \in A_i$, $1 \leq i \leq n$.

2.1.4 Sobre o uso de parêntesis

A proliferação de diferentes tipos de parêntesis em λ-calculus e programação funcional recomenda que façamos a discriminação dos seus usos:

- usamos parêntesis curvos simples $(\ldots)$ para agrupar expressões e tuplos;
- usamos as chavetas $\{\ldots\}$ para agrupar conjuntos;
- usamos parêntesis rectos em letra-de-máquina `[...]` para representar listas (quer na linguagem-fonte, quer manipuladas na análise de programa);
- usamos parêntesis rectos em letra normal $[\ldots]$ para representar *substituições* em λ-calculus, e mais genericamente, funções finitas (por exemplo: ambientes manipulados pelos esquemas de geração de código);
- usamos parêntesis rectos duplos $[\![\ldots]\!]$ para agrupar elementos sintácticos duma linguagem (por exemplo: expressões da linguagem-fonte ou instruções da máquina-abstracta) quando são manipuladas por esquemas de tradução, seguindo a notação usada em semânticas denotacionais e também nos esquemas de compilação em (Peyton Jones, 1987; Peyton Jones & Lestes, 1992).

2.2 Lambda calculus

O modelo formal para as linguagens de programação funcionais modernas é o λ-*calculus*, um cálculo simbólico desenvolvido por lógicos como Alonzo Church e Haskell Curry entre 1920–1930 como uma fundamentação matemática do cálculo funcional em que as funções são vistas como regras de re-escrita (Hankin, 1994). No λ-calculus as funções são representadas de forma *intencional* (*i.e.*, como expressões que denotam substituições) e não *extensional* (*i.e.*, como correspondências entre conjuntos).

Como modelo de computação o λ-calculus tem a expressividade equivalente à da máquina de Turing (Hankin, 1994), o que o torna, portanto, um modelo universal.

Reproduzimos aqui algumas definições e resultados básicos bem como simbologia que será utilizada nos capítulos seguintes.

2.2.1 Definição e sintaxe

Começamos por definir indutivamente o conjunto Λ de termos do λ-*calculus puro*:

Definição 1 Seja $\mathcal{V}$ um conjunto numerável de variáveis; o conjunto dos λ-termos Λ é definido como o menor conjunto tal que:

$$\begin{array}{lll} (i) & \mathcal{V} \subset \Lambda & (\mathit{Variáveis}) \\ (ii) & x \in \mathcal{V} \wedge M \in \Lambda \Rightarrow (\lambda x.M) \in \Lambda & (\mathit{Abstracção}) \\ (iii) & M, N \in \Lambda \Rightarrow (MN) \in \Lambda & (\mathit{Aplicação}) \end{array} \tag{2.1}$$

A *abstracção* $(\lambda x.M)$ de uma variável x num termo M é interpretada como a definição de uma função cujo argumento é x e cuja expressão de definição é M. Assim, o símbolo "λ" funciona como um quantificador da variável justaposta, indicando que esta é um "parâmetro formal" duma função.

A *aplicação* (MN) é interpretada como a aplicação de um argumento N à função representada pelo termo M. Para que esta interpretação faça sentido é necessário que M seja um λ-termo que representa uma função; considera-se (por construção) que (MN) é sempre um λ-termo quaisquer que sejam os λ-termos M, N, distinguindo depois λ-termos interpretáveis de λ-termos sem significado (designados "*mal-tipados*").

Seguimos as convenções usuais relativamente à omissão de parêntesis:

$$\begin{array}{lcl} \lambda x_1 \ldots x_n.M & \stackrel{def}{=} & (\lambda x_1(\lambda x_2(\ldots(\lambda x_n.M)\ldots))) \\ MN_1 \ldots N_n & \stackrel{def}{=} & (\ldots((MN_1)N_2)\ldots N_n) \end{array}$$

2.2.2 Variáveis e substituições

Vamos distinguir as ocorrências de variáveis conforme estão ou não abrangidas por um quantificador-λ:

Definição 2 Num termo da forma $\lambda x.M$ as ocorrências da variável x designam-se *ligadas*. Ocorrências de variáveis que não são ligadas designam-se *livres*.

Por exemplo, no λ-termo $((\lambda x.x)y)$, x ocorre ligada e y ocorre livre. Note-se que uma variável pode ocorrer simultaneamente livre e ligada, como x em $((\lambda x.xy)x)$.

O conjunto de todas as variáveis livres de um termo N é designado $FV(M)$ e o conjunto das variáveis ligadas é designado $BV(N)$. Assim, tem-se, por exemplo:

$$FV((\lambda x.xy)x) = \{x, y\} \quad BV((\lambda x.xy)x) = \{x\}$$

As variáveis ligadas são "anónimas": os seus nomes podem ser alterados mantendo o significado do λ-termo (no sentido de definir a mesma função). Esta noção é designada de *congruência-α*:

Definição 3 (*Congruência-α*) Dois λ-termos M, N são *congruentes-α* (escrevendo-se $M \equiv_\alpha N$) sse um deles pode ser obtido a partir do outro por mudanças de variáveis ligadas.

Devemos agora tornar mais precisa a mudança de variável ligada:

Definição 4 (*Mudança de variável ligada*) Uma *mudança da variável ligada* x num termo M substitui todos os sub-termos de M da forma $\lambda x.N$ por $\lambda y.N[x := y]$, onde y é uma variável que não ocorre em N e $N[x := y]$ é o termo resultante de substituir ocorrências livres de x em N por y.

Note-se que restringimos a mudança de variável a introduzir uma nova variável de forma a evitar a "captura" por outro quantificador-λ. Assim tem-se, por exemplo:

$$\begin{aligned} \lambda x.xy &\equiv_\alpha \lambda z.zy \\ \lambda x.xy &\not\equiv_\alpha \lambda y.yy \end{aligned}$$

Dois termos congruentes-α são indistintos do ponto da vista de redução que exprimem.

Para associar uma noção de redução aos λ-termos necessitamos de definir a *substituição* de uma variável livre por um termo:

Definição 5 (*Substituição*) A substituição das ocorrências livres da variável x em M por um novo termo N é representada por $M[x := N]$.

A substituição só é valida se x não ocorre livre num sub-termo de M da forma $\lambda y.P$ com $y \in FV(N)$; caso contrário teríamos de novo uma situação de captura da variável y, que passaria erradamente de livre a ligada.

Contudo, é sempre possível fazer uma substituição tomando para lugar de M um outro termo $M' \equiv_\alpha M$ e tal que $(BV(M')) \cap (FV(N)) = \emptyset$, o que corresponde a mudar as variáveis ligadas em M de forma a serem disjuntas das variáveis livres em N.

Assim, tem-se por exemplo:

$$\begin{aligned} ((\lambda x.xy)y)[y := x] &= ((\lambda z.zy)y)[y := x] && \text{(congruência-}\alpha\text{)} \\ &= ((\lambda z.zx)x) && \text{(substituição de } y\text{s livres)} \end{aligned}$$

Daqui em diante iremos considerar que qualquer substituição será válida, fazendo se necessário uma mudança de nomes de variáveis ligadas.

2.2.3 Redução e forma normal

A noção de redução em λ-calculus que corresponde à substituição de um parâmetro formal numa função é *redução-β*:

Definição 6 (*Redução-β*) Um λ-termo $(\lambda x.M)N$ reduz-se a $M[x := N]$ por *redução-β* e escreve-se $(\lambda x.M)N \rightarrow_{1\beta} M[x := N]$.

Um sub-termo da forma $(\lambda x.M)N$ é designado de *β-redex* e o termo reduzido $M[x := N]$ de *contractum*. Note-se que a substituição $M[x := N]$ pode envolver tomar um outro termo congruente-α com M para evitar a captura de variáveis.

Partindo da relação de redução num passo $\rightarrow_{1\beta}$ definimos a relação de redução-β em zero ou mais passos $\rightarrow_\beta$ como sendo o fecho reflexivo e transitivo de $\rightarrow_{1\beta}$.

Um λ-termo sem *β-redexes* diz-se em *forma-normal-β*; o resultado seguinte garante a unicidade da forma-normal quando esta existe:

Proposição 1 (Church-Rosser) Se $M \rightarrow_\beta N_1$ e $M \rightarrow_\beta N_2$ então existe um termo P tal que $N_1 \rightarrow_\beta P$ e $N_2 \rightarrow_\beta P$. ■

Corolário 1 Se $M \rightarrow_\beta N_1$ e $M \rightarrow_\beta N_2$ e N_1, N_2 estão em forma-normal-β então $N_1 \equiv_\alpha N_2$. ■

Nem todos os λ-termos admitem forma-normal-β; por exemplo:

$$(\lambda x.xx)(\lambda x.xx) \rightarrow_{1\beta} (\lambda x.xx)(\lambda x.xx) \rightarrow_{1\beta} \cdots$$

Mesmo quando um termo admite forma normal nem todas as sequências de reduções-β convergem para essa forma normal, ou seja, a escolha do *β-redex* a reduzir em cada passo não é indiferente. O resultado seguinte define uma escolha que garantidamente atinge a forma normal, caso esta exista:

Definição 7 (*Redução em ordem normal*) Uma sequência de reduções diz-se em *ordem normal* se em cada passo o *β-redex* escolhido for o mais à esquerda do λ-termo.

Proposição 2 Se um λ-termo M admite forma normal N então a sequência de reduções por ordem normal (*i.e.*, escolhendo em cada passo o *β-redex* mais à esquerda) e que começa em M converge para N. ■

A redução em ordem normal corresponde à redução "*lazy*" em linguagens de programação funcional: ao reduzir primeiro o *β-redex* mais à esquerda num λ-termo estamos a retardar a redução dos argumentos duma função até que estes sejam efectivamente necessários.

Contudo, a maior parte dos sistemas de linguagens funcionais não reduzem termos até à forma-normal-β; em vez disso terminam a redução numa forma-normal mais fraca designada "*weak head normal form*" (ver secção 2.3.4).

2.2.4 "*Currying*"

O λ-calculus não necessita duma sintaxe especial para definir funções de múltiplos argumentos; com efeito, uma função $f(x_1, \ldots, x_n) = M$ com n argumentos pode ser definida como

$$\begin{aligned} f &= \lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots)) \\ &= \lambda x_1 \ldots x_n.M \end{aligned}$$

isto é, f toma os n argumentos um de cada vez. Esta transformação é designada "*currying*" (em homenagem ao lógico Haskell Curry).

Esta convenção não só evita a necessidade de uma sintaxe especial como também torna mais simples tomar um "corte parcial" de uma função: por exemplo, no exemplo anterior, $(f\ x_1)$ é a função de $n-1$ argumentos que resulta de fixar o $1^{\underline{o}}$ argumento igual a x_1.

2.2.5 Constantes

Conceptualmente, o λ-calculus puro não necessita de qualquer constante primitiva para representar a computação: efectivamente é possível (por exemplo) definir os números naturais e os valores de verdade directamente como λ-termos puros e operar só com essas representações. De facto, qualquer função recursiva (ou equivalente: computável numa máquina de Turing) pode ser expressa no λ-calculus puro (Hankin, 1994).

Contudo, para modelar uma linguagem funcional faz mais sentido introduzir constantes primitivas para representar valores básicos (inteiros, por exemplo), uma vez que a implementação usará as operações internas à máquina física para manipular estes valores.

As operações binárias entre números inteiros são representadas por funções "*curried*". Por exemplo, a soma entre inteiros é uma função "*curried*" que dados dois inteiros calcula o inteiro correspondente à soma:

$$+ \ : \ \mathtt{Int} \longrightarrow \mathtt{Int} \longrightarrow \mathtt{Int}$$

Assim "$1 + 2 \times 3$" é representado pelo termo $(+\ 1\ (*\ 2\ 3))$ e a função sucessor é $(+\ 1)$, *i.e.*, a função que soma um ao seu argumento.

Definimos o conjunto $\Lambda_{\mathcal{C}}$ de λ-*termos com constantes* acrescentando à definição de λ-termos um conjunto $\mathcal{C}$ de constantes:

$$\mathcal{C} \stackrel{def}{=} \{\mathtt{True}, \mathtt{False}, 0, 1, -1, 2, -2 \ldots, +, -, *, / \ldots\}$$

O conjunto $\Lambda_{\mathcal{C}}$ é o menor conjunto tal que:

$$\begin{array}{lll}
(i) & \mathcal{V} \subset \Lambda_{\mathcal{C}} & (\mathit{Variáveis}) \\
(ii) & x \in \mathcal{V} \wedge M \in \Lambda_{\mathcal{C}} \ \Rightarrow \ (\lambda x.M) \in \Lambda_{\mathcal{C}} & (\mathit{Abstracção}) \\
(iii) & M, N \in \Lambda_{\mathcal{C}} \ \Rightarrow \ (MN) \in \Lambda_{\mathcal{C}} & (\mathit{Aplicação}) \\
(iv) & \mathcal{C} \subset \Lambda_{\mathcal{C}} & (\mathit{Constantes})
\end{array} \tag{2.2}$$

2.3 Lógica Combinatória

Paralelamente ao λ-calculus foi desenvolvido um outro formalismo para fundamentar o cálculo funcional, a *lógica combinatória*.

Os sistemas de lógica combinatória diferem do λ-calculus pela ausência do quantificador-λ e, consequentemente, de variáveis ligadas. Em vez disto temos um conjunto de constantes primitivas, designadas *combinadores*, à custa das quais podemos compor $\mathcal{C}\ell$-termos apenas usando aplicação:

Definição 8 Seja $\mathcal{V}$ um conjunto numerável de variáveis; S, K são dois combinadores primitivos. O conjunto CL dos $\mathcal{C}\ell$-termos é o menor conjunto tal que:

$$\begin{array}{lll}
(i) & \mathcal{V} \subset \mathsf{CL} & (\mathit{Variáveis}) \\
(ii) & \mathsf{S}, \mathsf{K} \in \mathsf{CL} & (\mathit{Constantes}) \\
(iii) & X, Y \in \mathsf{CL} \Rightarrow (XY) \in \mathsf{CL} & (\mathit{Aplicação})
\end{array} \tag{2.3}$$

Note-se que não havendo quantificadores-λ, qualquer variável $v \in \mathcal{V}$ que ocorra num termo é livre. Definimos assim $FV(U)$ como o conjunto de todas as variáveis (livres) que ocorrem num $\mathcal{C}\ell$-termo U.

Em vez de definir a redução à custa de substituições, definimos uma relação de redução elementar $\triangleright_{1\omega}$ para os combinadores S e K:

Definição 9 A redução elementar dos combinadores S e K é definida por

$$\begin{array}{lcl} \mathsf{K}xy & \triangleright_{1\omega} & x \\ \mathsf{S}xyz & \triangleright_{1\omega} & xz(yz) \end{array}$$

onde o lado esquerdo se designa *redex* e o lado direito *contractum*. A relação de redução elementar extende-se aos $\mathcal{C}\ell$-termos que contêm um *redex* como sub-termo, substituindo-o pelo respectivo *contractum*.

Analogamente ao que fizemos para o λ-calculus, definimos a relação $\triangleright_\omega$ como o fecho simétrico e transitivo de $\triangleright_{1\omega}$.

2.3.1 Relações com λ-calculus

Um resultado clássico na lógica combinatória (Hankin, 1994) é a transposição de um $\mathcal{C}\ell$-termo num λ-termo, dada pela função $()_\lambda : \mathsf{CL} \longrightarrow \Lambda$:

$$\begin{array}{lll} x_\lambda & = x & (x \in \mathcal{V}) \\ \mathsf{K}_\lambda & = \lambda xy.x & \\ \mathsf{S}_\lambda & = \lambda xyz.xz(yz) & \\ (XY)_\lambda & = X_\lambda Y_\lambda & \end{array} \tag{2.4}$$

Reciprocamente, podemos transpor um λ-termo num $\mathcal{C}\ell$-termo. Começamos por definir a transposição duma abstracção $\lambda x.M$ num $\mathcal{C}\ell$-termo, o que envolve um algoritmo de *abstracção combinatória* (originalmente apresentado por Schönfinkel).

Designamos por "$\mathsf{Abs}\ x\ M$" o $\mathcal{C}\ell$-termo resultante da abstracção combinatória da variável x no termo M.[2] Definimos a função Abs indutivamente na estrutura do termo M:

$$\begin{array}{lll} \mathsf{Abs}\ x\ x & = \mathsf{SKK} & \\ \mathsf{Abs}\ x\ (Ux) & = U & (\text{se } x \notin FV(U)) \\ \mathsf{Abs}\ x\ \ U & = \mathsf{K}U & (\text{se } x \notin FV(U)) \\ \mathsf{Abs}\ x\ (UV) & = \mathsf{S}(\mathsf{Abs}\ x\ U)(\mathsf{Abs}\ x\ V) & (\text{caso contrário}) \end{array} \tag{2.5}$$

[2] Não seguimos a notação original de Schönfinkel $[x]M$ para não sobrecarregar o uso de parêntesis rectos, em particular quando extendermos a abstracção para expressões da linguagem-fonte.

O resultado seguinte essencialmente mostra que a abstracção combinatória é a operação inversa da substituição:

Proposição 3 Sejam $M, N \in \mathsf{CL}$ e $x \in \mathcal{V}$. Então $(\mathsf{Abs}\ x\ M)N \rhd_\omega M[x := N]$. ■

A transposição entre λ-termos e $\mathcal{C}\ell$-termos, $()_H : \Lambda \longrightarrow \mathsf{CL}$ é definida indutivamente na estrutura do λ-termo; o caso de uma abstracção-λ é tratado usando a função Abs definida anteriormente:

$$\begin{aligned} x_H &= x, & (x \in \mathcal{V}) \\ (MN)_H &= M_H N_H \\ (\lambda x.M)_H &= \mathsf{Abs}\ x\ (M_H) \end{aligned} \tag{2.6}$$

Exemplo 1 A transposição do λ-termo $\lambda xy.x(yx)$ será

$$\begin{aligned} (\lambda xy.x(yx))_H &= \mathsf{Abs}\ x\ (\mathsf{Abs}\ y\ (x(yx))) \\ &= \mathsf{Abs}\ x\ (\mathsf{S}(\mathsf{K}x)(\mathsf{S}(\mathsf{SKK})(\mathsf{K}x))) \\ &= \mathsf{S}(\mathsf{S}(\mathsf{KS})\mathsf{K})(\mathsf{S}(\mathsf{K}(\mathsf{S}(\mathsf{SKK})))\mathsf{K}) \end{aligned}$$

Como o exemplo ilustra, esta "compilação" de λ-termos em combinadores S, K é muito ineficiente: no pior caso o tamanho do termo em combinadores cresce exponencialmente com o tamanho do λ-termo (Hankin, 1994) Além disso, o número de reduções em ordem normal no termo em combinadores S, K pode também crescer exponencialmente em relação às reduções-β no λ-termo. Isto implica que esta tradução elementar é demasiado ineficiente para qualquer implementação prática de uma linguagem funcional.

2.3.2 Combinadores $\mathsf{B}, \mathsf{C}, \mathsf{I}$

Podemos melhorar a complexidade desta tradução introduzindo três combinadores primitivos $\mathsf{B}, \mathsf{C}, \mathsf{I}$. Estes combinadores não aumentam a expressividade do cálculo (no sentido em que podem ser gerados usando só S, K) mas permitem reduzir o tamanho da tradução dum λ-termo para combinadores e o correspondente número de reduções em ordem normal.

Definição 10 Definimos as reduções elementares $\rhd_{1\omega}$ dos combinadores $\mathsf{B}, \mathsf{C}, \mathsf{I}$ como

$$\begin{aligned} \mathsf{B}xyz &\rhd_{1\omega} x(yz) \\ \mathsf{C}xyz &\rhd_{1\omega} xzy \\ \mathsf{I}x &\rhd_{1\omega} x \end{aligned}$$

Os combinadores B, C são "especializações" do combinador S em que o último argumento é aplicado apenas ao segundo ou primeiro argumento, respectivamente. O combinador I é a função identidade que no sistema S, K se codificava como SKK.

Podemos efectuar algumas optimizações ao esquema de tradução de λ-termos para combinadores usando as seguintes identidades elementares, provadas em (Hankin, 1994):

$$\begin{array}{lcl} \mathsf{SKK} & \equiv & \mathsf{I} \\ \mathsf{S}(\mathsf{K}M)(\mathsf{K}N) & \equiv & \mathsf{K}(MN) \\ \mathsf{S}(\mathsf{K}M)N & \equiv & \mathsf{B}MN \\ \mathsf{S}M(\mathsf{K}N) & \equiv & \mathsf{C}MN \end{array} \tag{2.7}$$

Usamos estas identidades para simplificar o termo resultante após cada abstracção; designando

$$\mathsf{Abs}'\ x = \mathsf{Opt}\ (\mathsf{Abs}\ x)$$

onde Opt substitui um $\mathcal{C}\ell$-termo dum lado-esquerdo das identidades 2.7 pelo respectivo lado-direito, podemos então usar Abs' em vez de Abs nos lados-direitos das equações 2.5 e 2.6 que definem Abs e $()_H$ respectivamente.

Exemplo 2 O λ-termo $\lambda xy.x(yx)$ do exemplo 1 é traduzido em:

$$\begin{aligned} (\lambda xy.x(yx))_H &= \mathsf{Abs}'\ x\ (\mathsf{Abs}'\ y\ x(yx)) \\ &= \mathsf{Abs}'\ x\ (\mathsf{B}x(\mathsf{CI}x)) \\ &= \mathsf{SB}(\mathsf{CI}) \end{aligned}$$

Pode-se mostrar que esta tradução para combinadores $\mathsf{S}, \mathsf{K}, \mathsf{I}, \mathsf{B}, \mathsf{C}$ obtém termos cujo comprimento é, no pior caso, $O(n^3)$ para λ-termos de comprimento n; uma relação idêntica existe entre o número de passos de redução (Broda, 1996).

2.3.3 Combinadores $\mathcal{C}\ell(\mathcal{K})$

Apresentamos agora o sistema de combinadores que será usado na implementação duma linguagem funcional: o sistema $\mathcal{C}\ell(\mathcal{K})$. Este sistema foi apresentado em (Broda & Damas, 1997; Broda & Damas, 1995) e ainda em (Broda, 1996).

O sistema $\mathcal{C}\ell(\mathcal{K})$ é definido a partir de um conjunto infinito e numerável de combinadores primitivos (*i.e.*, constantes) $\mathcal{K}$. Estes combinadores "codificam" sequências de combinadores elementares S, K, B, C, I (num sentido que será tornado mais preciso em seguida).

Cada abstracção-λ num λ-termo original pode então ser traduzida num termo em $\mathcal{C}\ell(\mathcal{K})$ com (quando muito) um novo combinador, o que evita o crescimento exponencial com abstracções sucessivas. De facto, prova-se em (Broda, 1996) que λ-termos de comprimento n traduzem-se em $\mathcal{C}\ell(\mathcal{K})$-termos de tamanho $O(n^2)$, enquanto uma prova abreviada é apresentada na secção 3.4.1. Além disto, o número de passos de redução até à forma normal do $\mathcal{C}\ell(\mathcal{K})$-termo não excede o número de passos de redução-β em ordem normal do λ-termo original.

Do ponto-de-vista duma implementação o sistema $\mathcal{C}\ell(\mathcal{K})$ concilia a redução simples em combinadores sem captura de variáveis com a complexidade em número de reduções do programa original.

$\mathcal{C}\ell(\mathcal{K})$-termos

Começamos por descrever o conjunto de índices $\mathcal{A}$ dos combinadores primitivos. Este conjunto é infinito e é definido indutivamente:

Definição 11 O conjunto de índices de $\mathcal{C}\ell$-termos $\mathcal{A}$ é o menor conjunto tal que:

$$\begin{array}{ll} (i) & \epsilon \in \mathcal{A} \\ (ii) & \alpha \in \mathcal{A} \Rightarrow c \cdot \alpha \in \mathcal{A} \\ (iii) & \alpha \in \mathcal{A} \Rightarrow b \cdot \alpha \in \mathcal{A} \\ (iv) & \alpha_1, \alpha_2 \in \mathcal{A} \Rightarrow (\alpha_1, \alpha_2) \in \mathcal{A} \end{array}$$

Definição 12 O conjunto dos combinadores primitivos no sistema $\mathcal{C}\ell(\mathcal{K})$ é

$$\mathcal{K} \stackrel{def}{=} \{\mathsf{K}\} \cup \{\Phi_\alpha,\ \alpha \in \mathcal{A}\}$$

O conjunto $\mathsf{CL}(\mathcal{K})$ dos $\mathcal{C}\ell(\mathcal{K})$-termos é então definido indutivamente como o menor conjunto tal que:

$$\begin{array}{lll} (i) & \mathcal{V} \subset \mathsf{CL}(\mathcal{K}) & (\textit{Variáveis}) \\ (ii) & \mathcal{K} \subset \mathsf{CL}(\mathcal{K}) & (\textit{Constantes}) \\ (iii) & X, Y \in \mathsf{CL}(\mathcal{K}) \Rightarrow (XY) \in \mathsf{CL}(\mathcal{K}) & (\textit{Aplicação}) \end{array} \tag{2.8}$$

Antes de apresentar as regras de redução elementar e um algoritmo de abstracção, vamos ilustrar com um exemplo a transposição de um λ-termo para combinadores $\mathcal{C}\ell(\mathcal{K})$.

Exemplo 3 O λ-termo $\lambda xy.x(yx)$ é transposto em combinadores $\mathcal{C}\ell(\mathcal{K})$ para $\Phi_{(b,\epsilon)}\Phi_{bc}$. As regras de redução dos combinadores elementares são, respectivamente

$$\begin{array}{lll} \Phi_{(b,\epsilon)}\ xy & \triangleright_{1\omega} & xyy \\ \Phi_{bc}\ xyz & \triangleright_{1\omega} & x(zy) \end{array}$$

pelo que podemos verificar que o termo $\Phi_{(b,\epsilon)}\ \Phi_{bc}\ x\ y$ se reduz de forma equivalente ao λ-termo original:

$$\Phi_{(b,\epsilon)}\ \Phi_{bc}\ x\ y\ \triangleright_{1\omega}\ \Phi_{bc}\ x\ x\ y\ \triangleright_{1\omega}\ x(yx)$$

Note-se em particular que efectuamos duas reduções em $\mathcal{C}\ell(\mathcal{K})$ tal como na redução do λ-termo original.

Reduções elementares

Como foi dito, cada combinador Φ_α irá corresponder a uma sequência dos combinadores elementares $\mathsf{B}, \mathsf{C}, \mathsf{S}, \mathsf{I}$. A redução elementar de cada combinador Φ_α é definida indutivamente pelo índice α.

Começamos por definir um potencial "lado-esquerdo" de uma redução elementar, *i.e.*, um *redex fraco*:

Definição 13 Designamos *grau* de um combinador primitivo Φ_α como o número de símbolos b, c que ocorrem no índice α (e representamos por $\#\alpha$); o grau do combinador K é 1.

Um termo da forma $\phi\ X_1 \ldots X_{n+1}$ (onde ϕ é um combinador primitivo de grau n) é um *redex fraco*.

O *contractum* de um *redex* fraco é definido indutivamente sobre o índice α:

$$
\begin{array}{ll}
k(\Phi_\epsilon\ X) & = X \\
k(\Phi_{c\cdot\alpha}\ X X_1 \ldots X_{\#\alpha+1}) & = (k(\Phi_\alpha\ X_1 \ldots X_{\#\alpha+1})\ X) \\
k(\Phi_{b\cdot\alpha}\ X X_1 \ldots X_{\#\alpha+1}) & = (X\ k(\Phi_\alpha\ X_1 \ldots X_{\#\alpha+1})) \\
k(\Phi_{(\alpha_1,\alpha_2)}\ X_1 \ldots X_{\#\alpha_1} Y_1 \ldots Y_{\#\alpha_2} Z) & = (k(\Phi_{\alpha_1}\ X_1 \ldots X_{\#\alpha_1} Z)\ k(\Phi_{\alpha_2}\ Y_1 \ldots Y_{\#\alpha_2} Z))
\end{array}
\tag{2.9}
$$

A relação de redução elementar fica então definida por

$$
\begin{array}{lll}
\mathsf{K}\ xy & \rhd_{1\omega} & x \\
\Phi_\alpha\ x_1 \ldots x_{\#\alpha+1} & \rhd_{1\omega} & k(\Phi_\alpha\ x_1 \ldots x_{\#\alpha+1})
\end{array}
\tag{2.10}
$$

Note-se que embora conceptualmente a definição de $\rhd_{1\omega}$ pressuponha um número infinito de possíveis reduções elementares, cada programa funcional concreto utilizará apenas um conjunto finito de combinadores Φ_α. A cada um desses combinadores corresponderá na implementação uma sequência de código que efectua a sua redução a partir das equações 2.9.

Abstracção em $\mathcal{C}\ell(\mathcal{K})$

O algoritmo original para abstracção de uma variável livre num $\mathcal{C}\ell(\mathcal{K})$-termo apresentado em (Broda, 1996) passa por uma abstracção intermédia em combinadores $\mathsf{S}, \mathsf{K}, \mathsf{B}, \mathsf{C}, \mathsf{I}$ para depois obter a representação em combinadores Φ_α. O algoritmo apresentado em seguida é um refinamento desse algoritmo original, obtendo directamente a abstracção em $\mathcal{C}\ell(\mathcal{K})$.

Tal como da abstracção em combinadores S,K a função Abs obtém o termo que resulta de abstrair a variável x em M:

$$
\begin{aligned}
&\mathsf{Abs}\ x\ U &&= \mathsf{K}U && (\text{se } x \notin FV(U))\\
&\mathsf{Abs}\ x\ (Ux) &&= U && (\text{se } x \notin FV(U))\\
&\mathsf{Abs}\ x\ U &&= \Phi_\alpha\ X_1 \dots X_n && (\text{caso contrário})\\
& && \text{onde } (\alpha,\ [X_1, \dots, X_n]) = \mathsf{index}\ x\ U &&
\end{aligned}
\tag{2.11}
$$

A função auxiliar index é usada para abstrair uma variável x de um termo U quando se sabe que x ocorre livre em U. A abstracção será da forma $\Phi_\alpha\ X_1 \dots X_n$; assim o resultado de index é um par ordenado $(\alpha, [X_1, \dots, X_n])$ em que α é o índice e $[X_1, \dots, X_n]$ é a lista de termos que devem ser aplicados a Φ_α:

$$
\begin{aligned}
&\mathsf{index}\ x\ x &&= (\epsilon,\ [\,]) &&\\
&\mathsf{index}\ x\ (UV) &&= (c \cdot \alpha,\ V\!:\!l) && (\text{se } x \in FV(U), x \notin FV(V))\\
& && \text{onde } (\alpha, l) = \mathsf{index}\ x\ U &&\\
&\mathsf{index}\ x\ (UV) &&= (b \cdot \alpha,\ U\!:\!l) && (\text{se } x \notin FV(U), x \in FV(V))\\
& && \text{onde } (\alpha, l) = \mathsf{index}\ x\ V &&\\
&\mathsf{index}\ x\ (UV) &&= ((\alpha_1, \alpha_2),\ l_1 +\!\!+ l_2) && (\text{se } x \in FV(U), x \in FV(V))\\
& && \text{onde } \ (\alpha_1, l_1) = \mathsf{index}\ x\ U &&\\
& && \phantom{\text{onde }}\ (\alpha_2, l_2) = \mathsf{index}\ x\ V &&
\end{aligned}
\tag{2.12}
$$

Note-se que os casos considerados nas equações 2.12 são exaustivos uma vez que se sabe que x ocorre pelo menos uma vez livre no termo.

Usando a função Abs define-se a transposição entre um λ-termo e $\mathcal{C}\ell(\mathcal{K})$-termos de forma análoga ao que foi feito para $\mathcal{C}\ell$-termos (equação 2.6, pág. 23).

2.3.4 Forma-normal fraca à cabeça

Já referimos que as implementações de linguagens funcionais não-estritas reduzem termos por ordem normal mas terminam antes de atingir a forma-normal-β: em particular, não reduzem termos dentro de λs (*i.e.*, se o resultado é da forma $\lambda x.M$, então M não será mais reduzido).

Estamos em condições de tornar precisa a noção de forma-normal associada à "resposta" duma redução funcional não-estrita. Uma vez que as implementações mais eficientes de redução não-estrita (e também a nossa implementação) se baseiam em tradução para combinadores, vamos definir a forma-normal para a redução em combinadores:

Definição 14 Dizemos que um termo $M \equiv \phi\ X_1 \ldots X_n$ está em "forma-normal fraca à cabeça" (do original: "*weak head normal form*" ou "whnf") sse ϕ é um combinador primitivo de k argumentos e $n < k$ (*i.e.*, o combinador ϕ não tem o número suficiente de argumentos para se efectuar uma redução).

Caso contrário, se $n \geq k$ então o termo não está em whnf; o *redex* fraco à cabeça do termo é então

$$\underbrace{(\phi\ X_1 \ldots X_k)}_{\textit{redex}\ \text{fraco}} X_{k+1} \ldots X_n$$

Note-se que o *redex* fraco à cabeça é sempre o mais à esquerda no termo, mas o recíproco nem sempre é verdade, como ilustra o exemplo seguinte:

Exemplo 4 No termo $\mathsf{K}(\mathsf{II})$, não há *redex* fraco à cabeça (*i.e.*, o termo está em whnf) pois K necessita de dois argumentos. O *redex* fraco mais à esquerda é (II).

Ao contrário da forma-normal-β, a whnf não é única: o termo em whnf do exemplo anterior, $\mathsf{K}(\mathsf{II})$ admite outra whnf KI.

Um sistema de redução não-estrito (*i.e.*, um sistema "*lazy*") terminaria a avaliação de $\mathsf{K}(\mathsf{II})$ sem fazer mais reduções; designamos esta forma de *whnf-principal* e vamos associá-la à resposta da redução não-estrita do termo:

Definição 15 A *whnf-principal* de um termo resulta da redução em cada passo do *redex* fraco à cabeça quando uma tal sequência de reduções converge.

Se a redução pela ordem do *redex* fraco à cabeça não convergir, então o termo não tem whnf (e por maioria de razão, não tem whnf-principal).

Capítulo 3

A linguagem-fonte

A linguagem-fonte que nos propomos compilar constitui uma linguagem de programação funcional minimal, que mantém as características nucleares de uma linguagem funcional moderna descritas na introdução, mas que evita a complexidade de uma linguagem de programação real como Haskell.

Esta linguagem-fonte (que é essencialmente o λ-calculus com algumas extensões) pode ser vista como um estágio intermédio num sistema de compilação de Haskell (Peyton Jones, 1996; Peyton Jones, 1993) ou como um ponto de partida para a tradução para uma linguagem-máquina. É esta última abordagem que nos propomos explorar.

3.1 Sintaxe e semântica

A linguagem-fonte é uma extensão do λ-calculus com constantes básicas (como inteiros e operações aritméticas) e algumas construções sintácticas para definições locais e selecção de alternativas. A figura 3.1 descreve a sintaxe completa em notação BNF.

3.1.1 Abstracção e aplicação

Tal como em λ-calculus, as duas construções fundamentais são a *abstracção funcional* que define uma função a partir de uma expressão e a *aplicação* de um valor como argumento a uma função.

Seguindo a sintaxe usada na linguagem Haskell, representamos uma abstracção-λ em x por $\backslash x \to E$ e a aplicação de uma função f a um valor x por justaposição simples $f\ x$.

Seguimos ainda a convenção usual no λ-calculus e nas linguagens funcionais modernas de re-

$Prog$	$::=$	$Globs$	programa
$Globs$	$::=$	$Glob_1; \cdots ; Glob_n;$	definições globais
$Glob$	$::=$	$f\ var_1 \cdots var_n = Expr$	
$Defs$	$::=$	$Def_1; \cdots; Def_n$	definições locais
Def	$::=$	$var = Expr$	
$Expr$	$::=$	$Expr_1\ Expr_2$	aplicação
	$\mid$	$\backslash var \rightarrow Expr$	abstracção-λ
	$\mid$	`case` $Expr$ `of` $Alts$	expressão de selecção
	$\mid$	`let` $Defs$ `in` $Expr$	definição não-recursiva
	$\mid$	`letrec` $Defs$ `in` $Expr$	definição recursiva
	$\mid$	$(\, Expr\,)$	
	$\mid$	num	constantes inteiras
	$\mid$	$prim$	operações primitivas
$Alts$	$::=$	$Alt_1 \mid \cdots \mid Alt_n$	alternativas
Alt	$::=$	$Cons\ var_1 \cdots var_n \rightarrow Expr$	alternativa com construtor
	$\mid$	$var \rightarrow Expr$	
	$\mid$	`default` $\rightarrow Expr$	
$Cons$	$::=$	`:` $\mid$ `[]`	construtores de listas
	$\mid$	`True` $\mid$ `False`	construtores booleanos
	$\mid$	$\cdots$	
var	$::=$	`x` $\mid$ `y` $\mid \cdots$	
num	$::=$	$\cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots$	
$prim$	$::=$	$+ \mid - \mid * \mid \cdots$	

Figura 3.1: Sintaxe BNF da linguagem-fonte

presentar uma função de múltiplos argumentos como uma função "*curried*" (cf. secção 2.2.4).

Para não sobrecarregar a notação, omitimos parêntesis em abstracções-λ consecutivas e na aplicação de uma função a múltiplos argumentos associando, respectivamente, à direita e à esquerda:

$$\begin{array}{lcl} \backslash x_1\ x_2\ \ldots\ x_n \to E & \stackrel{def}{=} & \backslash x_1 \to (\backslash x_2 \to (\ldots(\backslash x_n \to E)\ldots)) \\ f\ e_1\ e_2\ \ldots\ e_n & \stackrel{def}{=} & (\ldots((f\ e_1)\ e_2)\ \ldots e_n) \end{array}$$

3.1.2 Inteiros e operações primitivas

O único tipo básico considerado é o conjunto dos inteiros `Int` como as operações primitivas entre eles: $+, -, *, /, \mathrm{mod}$. Estas operações binárias $\odot$ são funções primitivas "*curried*":

$$\odot : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}$$

Além das operações internas entre inteiros temos operações $\propto$ de comparação (como $==, <, \leq$) cujo resultado é um valor de verdade (booleano) representado usando um *tipo estruturado* `Bool` (ver secção 3.1.6):

$$\propto : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Bool}$$

Por conveniência usamos algumas abreviaturas sintácticas nos operadores primitivos que não acrescentam expressividade à linguagem, sendo substituídas pelas formas elementares durante a análise sintáctica do programa-fonte:

- a utilização de operadores binários infixos:

  $$e_1 \odot e_2 \stackrel{def}{=} (\odot\ e_1\ e_2)$$

  onde $\odot$ é um operador infixo como $+, -, *, /, <, \leq, ==, \ldots$.

- a utilização da associatividade e precedência entre operadores aritméticos usuais, como por exemplo:

  $$e_1 + e_2 * e_3 \stackrel{def}{=} (+\ e_1\ (*\ e_2\ e_3))$$

3.1.3 Definições globais

Um programa na linguagem-fonte é um conjunto de *definições globais* (também designadas *super-combinadores*). Sintacticamente uma definição global é uma associação entre um *identificador* e uma *expressão*. Uma função como $f(x, y) = 2x + y$ pode ser definida como

```
f x y = 2*x+y ;
```

Em geral, um super-combinador f com n argumentos corresponderá a uma definição global

$$\texttt{f}\ x_1\ x_2\ \ldots\ x_n\ \texttt{=}\ \ldots\ ;$$

A definição do supercombinador f acima poderia igualmente ser escrita como

$$\texttt{f = }\backslash x_1\ x_2\ \ldots\ x_n \rightarrow \ldots\ ;$$

Embora a semântica das duas definições seja a mesma, os nossos esquemas de compilação irão gerar código diferente para as duas definições: a primeira define um supercombinador com n argumentos enquanto a segunda define um supercombinador sem argumentos associado a uma abstracção-λ.

Note-se que só pode existir uma definição global para um dado identificador: definições com alternativas devem ser escritas explicitamente como expressões-`case` (ver secção 3.1.6).

O âmbito de um identificador global é todo o texto do programa e as definições globais são interpretadas *mutuamente recursivas*. A ordem entre definições não é significativa e uma referência a um identificador global pode preceder a sua definição.

3.1.4 Definições locais

A primeira extensão ao λ-calculus é a possibilidade de introduzir *definições locais de variáveis*:

Definição 16 A expressão

$$\texttt{let}\ x_1 = e_1; x_2 = e_2; \ldots; x_n = e_n\ \texttt{in}\ E$$

representa o valor da expressão E num contexto em que as variáveis $x_1, x_2, \ldots, x_n$ são associadas às expressões $e_1, e_2, \ldots, e_n$ respectivamente.

Ao contrário do que sucede com as definições globais, as definições locais devem ser *simples*: para associar algum dos x_i a uma função é necessário usar abstracções-λ: `let` $f = \backslash x \rightarrow \ldots$ `in` ....

Tal como a abstracção-λ, esta construção sintáctica *liga* as variáveis $x_1, \ldots, x_n$. As (eventuais) ocorrências destas variáveis na expressão E dizem-se *ligadas* pelo construtor `let`.

Note-se ainda que ocorrências de $x_1, x_2, \ldots x_n$ nas expressões $e_1, e_2, \ldots, e_n$ *não* são ligadas pelo `let`: tais ocorrências serão livres ou ligadas por um outro quantificador exterior.

As definições locais poderiam ser expressas usando apenas abstracções-λ:

$$\begin{aligned} & \texttt{let}\ x_1 = e_1; x_2 = e_2; \ldots; x_n = e_n\ \texttt{in}\ E \quad \equiv \\ \equiv \quad & (\backslash x_1\ x_2\ \ldots\ x_n \to E)\ e_1\ e_2\ \ldots\ e_n \end{aligned}$$

Contudo, a introdução desta construção como primitiva à linguagem-fonte possibilitará em certas circunstâncias a geração de código mais eficiente do que para aplicações (como veremos nas secções 5.2.1 e 5.2.2). Além disto, algumas propriedades desejáveis na redução não-estrita podem não ser preservadas se o compilador não tratar de forma especial as expressões-`let` (como será analisado mais em profundidade na secção 3.4.2).

3.1.5 Definições recursivas

Uma variante da construção sintáctica para definições locais permite a definição de *expressões mutuamente recursivas*:

Definição 17 A expressão

$$\texttt{letrec}\ x_1 = e_1; x_2 = e_2; \ldots; x_n = e_m\ \texttt{in}\ E$$

representa o valor da expressão E num contexto em que as $x_1, x_2, \ldots, x_n$ são associadas às expressões $e_1, e_2, \ldots, e_n$ tomadas mutuamente recursivas.

Tal como nas expressões-`let` as definições locais recursivas devem ser simples.

A recursão pode ser expressa em λ-calculus usando um *operador* (ou *combinador*) ponto-fixo Y (Hankin, 1994; Peyton Jones, 1987):

$$\texttt{letrec}\ f = e\ \texttt{in}\ E \ \equiv\ \texttt{let}\ f = \mathsf{Y}\,(\lambda f.e)\ \texttt{in}\ E$$

Podemos tomar para operador ponto-fixo qualquer λ-termo Y como a propriedade

$$\mathsf{Y}\,x \ \to_\beta\ x\,(\mathsf{Y}\,x), \quad \forall x$$

O combinador ponto-fixo usual na literatura é o λ-termo $\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$. Múltiplas expressões mutuamente recursivas podem ser tratadas agrupando-as num tuplo ou usando diversos combinadores ponto-fixo (ver a secção 3.3.4).

Não iremos usar o operador ponto-fixo para tratar sistematicamente definições recursivas: em muitas circunstâncias será possível gerar código de redução de grafos que constrói directamente *grafos cíclicos* associados a um conjunto de definições mutuamente recursivas. Uma expressão-`letrec` será então deixada como uma *representação textual para grafos cíclicos* (Peyton Jones, 1987, cap. 14).

3.1.6 Tipos estruturados

Os *tipos estruturados*, também designados de *tipos algébricos* constituem um modo sistemático de agrupar tipos compostos (como listas, árvores, tuplos, etc.) definidos à custa dos tipos básicos.

Um tipo estruturado τ é visto como uma soma disjunta de n tuplos, cada qual identificado por um *construtor* σ_i:

$$\begin{array}{rll} \tau ::= & \sigma_1 & \alpha_1^1\ \alpha_2^1\ \cdots\ \alpha_{m_1}^1 \\ | & \sigma_2 & \alpha_1^2\ \alpha_2^2\ \cdots\ \alpha_{m_2}^2 \\ & \vdots & \\ | & \sigma_n & \alpha_1^n\ \alpha_2^n\ \cdots\ \alpha_{m_n}^n \end{array}$$

onde os α_i^k são outros tipos e $m_i \geq 0$ são as *aridades* dos construtores σ_i.

Os nomes σ_i são funções de aridade m_i que constroem uma expressão do tipo τ com o construtor respectivo.

A definição de *listas homogéneas* de αs (onde α é um tipo qualquer) pode ser escrita como:

$$\begin{array}{rl} \mathtt{List}_\alpha ::= & :\ \alpha\ \mathtt{List}_\alpha \\ | & \mathtt{[]} \end{array}$$

onde ":" é o construtor que acrescenta um elemento à cabeça de uma lista e e "[]" é o construtor da lista vazia.

Como outro exemplo, os valores de verdade `Bool` podem ser vistos como um tipo estruturado com dois construtores ambos sem argumentos:

$$\begin{array}{rl} \mathtt{Bool} ::= & \mathtt{True} \\ | & \mathtt{False} \end{array}$$

Neste último caso os construtores `True` e `False` são funções de aridade zero (*i.e.*, constantes) identificadas com os valores de verdade.

Expressões-`case`

A expressão `case` ... `of` ... permite seleccionar qual o construtor de um tipo estruturado e decompor as suas constituintes:

Definição 18 A expressão

$$
\begin{array}{lllll}
\texttt{case}\ E\ \texttt{of} & & & & \\
 & \sigma_1 & x_1 \ldots x_{m_1} & \rightarrow & e_1 \\
 & \mid \sigma_2 & x_1 \ldots x_{m_2} & \rightarrow & e_2 \\
 & & \vdots & & \vdots \\
 & \mid \sigma_n & x_1 \ldots x_{m_n} & \rightarrow & e_n
\end{array}
$$

reduz E (que deve ser um termo do tipo τ) e selecciona a expressão a reduzir entre $e_1, e_2, \ldots, e_n$ conforme E foi construída respectivamente por $\sigma_1, \sigma_2, \ldots, \sigma_n$.

Em cada e_i as variáveis $x_1, \ldots x_{m_i}$ são ligadas às correspondentes m_i componentes de E.

Note-se que para facilitar a exposição supõe-se, sem perda de generalidade, que as variáveis ligadas em todos os ramos do `case` são $x_1, x_2, \ldots$ mas efectivamente não é necessário que os nomes sejam iguais entre ramos. O âmbito das variáveis $x_1, \ldots, x_{m_i}$ é (apenas) a expressão e_i.

Note-se ainda que construtores de aridade zero não terão variáveis ligadas (*i.e.*, $m_i = 0$).

Como exemplo, vejamos uma possível definição duma função para obter o comprimento de uma lista:

```
length l = case l of
                 [] -> 0
                |:  h t -> 1+length t ;
```

Uma expressão `case` deve ser *nivelada*, isto é, apenas pode decompor um nível do tipo estruturado. Expressões `case` mais complexas devem ser traduzidas em vários `case` simples; vejamos um exemplo de uma função que verifica se uma lista é singular (*i.e.*, tem um só elemento):

```
singular l = case l of
                   [] -> False
                  |:  h t -> case t of
                                   [] -> True
                                  |:  h2 t2 -> False ;
```

Alternativas `var` e `default`

Introduzimos também duas abreviaturas sintáticas, as alternativas `var` e `default`:

- a alternativa "`var` $v \to E$" sucede sempre e liga a variável v ao tipo estruturado;
- a alternativa "`default` $\to E$" sucede sempre sem ligar variáveis.

Note-se que qualquer destas construções só faz sentido como últimas alternativas (pois sucedem sempre).

Como cada tipo estruturado tem um número finito de construtores, estas construções não acrescentam expressividade à linguagem. Poderemos, contudo, gerar código um pouco mais eficiente para estas situações como veremos no capítulo 5.

Expressões `if-then-else`

O resultado dos operadores condicionais (como `==`, `/=`, `<`, etc.) é um valor booleano representado pelo tipo estruturado `Bool`. Assim, uma expressão condicional `if-then-else` é um caso particular duma expressão-`case`:

$$\begin{aligned} & \texttt{if}\ cond\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \stackrel{def}{=} \\ \stackrel{def}{=} \quad & \texttt{case}\ cond\ \texttt{of True} \to e_1 \mid \texttt{False} \to e_2 \end{aligned}$$

Para facilitar a escrita e leitura de programas usaremos também a sintaxe `if-then-else` sub-entendendo-se que será transcrita para uma expressão-`case`. Assim os esquemas de compilação não tratarão de forma especial estas expressões.

3.2 Ambiente de execução

3.2.1 Execução de programa

Um programa na linguagem-fonte é um conjunto de *definições globais*. De entre estas, o identificador `main`, sem argumentos, corresponde ao ponto-de-entrada de execução do programa como um todo.

A redução desta expressão não efectua, contudo, qualquer operação de entrada ou saída de dados implícita. Para que sejam passados dados ou resultados entre o sistema operativo e o programa funcional é necessário empregar algumas primitivas de entrada/saída (ver secção 3.2.2).

Note-se ainda que `main` deve terminar explicitamente o ciclo de execução da linguagem usando a primitiva `exit`.

Acção	**Descrição**
`putInt` n	saída de um inteiro
`putChar` n	saída de um carácter (código ASCII)
`getInt`	entrada de um inteiro
`getChar`	entrada de um carácter (código ASCII)
a `>>` b	sequenciação: acção a antes de b
a `>>=` b	sequenciação com passagem de valor
`exit`	fim de execução

Tabela 3.1: Acções de entrada/saída

3.2.2 Primitivas de entrada/saída

Para efectuar acções a entrada/saída numa linguagem puramente funcional usamos dois *operadores de sequenciação* semelhantes aos que são usados na linguagem Haskell (Peterson & Hammond, 1996) mas sem a estrutura de tipos associada.

Tal como os operadores aritméticos, estes operadores são entendido pelo analisador sintáctico de forma infixa por conveniência, mas correspondem apenas a funções pré-definidas em C na biblioteca de execução. Não é necessária nenhuma modificação ao modelo abstracto de redução (ver capítulo 4, secção 4.2).

Os operadores de sequenciação são então:

a `>>` b reduz a e depois b; o resultado da primeira redução é desprezado e o resultado da expressão total é o de b.

a `>>=` b reduz a e depois b; o resultado da primeira redução é passado a b. Assim a semântica desta operação é a mesma de $(b\ a)$ mas onde a ordem de redução é fixa (primeiro a e depois b).

A tabela 3.1 descreve as primitivas de entrada/saída. Ilustramos a sua utilização com dois exemplos:

Exemplo 5 O programa funcional seguinte lê um inteiro da entrada-padrão e escreve o seu quadrado na saída-padrão:

```
main = (getInt >>= (\n -> putInt (n*n))) >> exit ;
```

Exemplo 6 Para ler ou escrever um objecto estruturado composto (como uma lista de inteiros) é necessário escrever operações de entrada/saída de mais alto nível. Neste exemplo ilustramos um "procedimento" que escreve uma lista de inteiros:

```
putInts l = case l of [] -> []
                    | :  h t -> putInt h >> putInts t ;
```

Em alternativa (e usando um estilo mais funcional) podemos ver este "procedimento" como uma composição de duas funções de ordem superior, `map` e `foldr`, cujas definições são clássicas em textos introdutórios à programação funcional como (Bird & Wadler, 1988):

```
putInts l = foldr (>>) [] (map putInt l) ;

map f l = case l of [] -> []
                  | :  h t -> (f h):(map f t) ;

foldr f e l = case l of [] -> e
                      | :  h t -> f h (foldr f e t) ;
```

Este exemplo ilustra o uso de `>>` e `putInt` como "objectos de 1ª classe" na linguagem: funções passadas como argumentos a `foldr` e `map`.

3.3 Tradução para combinadores compactos

A linguagem-fonte permite-nos definir funções em que ocorrem variáveis ligadas num âmbito lexical superior usando abstracções-λ. Como referimos na secção 2.2.2, a presença de variáveis livres pode implicar re-etiquetar variáveis antes de efectuar reduções para evitar "captura" de variáveis. Para tratar essas variáveis a implementação necessita de listas-ligadas de ambientes, como na máquina SECD (Kogge, 1991, cap. 7).

Esta situação não é específica da implementação de linguagens funcionais: uma situação análoga ocorre, por exemplo, na implementação da linguagem PASCAL que permite definir procedimentos embricados com visibilidade de variáveis não-locais (Aho et al., 1986, secção 7.4).

A redução em combinadores é comparativamente muito mais simples: não existindo variáveis ligadas, todas as reduções ficam definidas pelos *contractums* associados aos combinadores elementares. Isto permite que o ambiente de execução seja mantido numa pilha LIFO usual e o seu espaço libertado após cada redução. No contexto da redução de grafos, a redução de cada combinador elementar pode ser vista como *re-escrita local do grafo*.

Uma forma de evitar as complicações associadas à redução com abstracções-λ é transformar o programa original num programa equivalente em combinadores-SK (Turner, 1979a; Turner, 1979b); no nosso caso vamos usar os combinadores-$\mathcal{C}\ell(\mathcal{K})$ apresentados na secção 2.3.3.

A alternativa mais importante é o "*lambda lifting*" que transforma o programa directamente num conjunto de definições de *supercombinadores* (Peyton Jones, 1995). De facto, a nossa transformação não substitui todas as situações tratadas por "*lambda lifting*" pelo que deve ser considerada uma alternativa parcial a esta. Desenvolveremos mais a comparação entre as duas abordagens na secção 3.4.2.

3.3.1 A tradução em linhas gerais

O objectivo da nossa tradução é manter as definições globais do programa original (que correspondem a supercombinadores) e substituir abstracções-λ nessas definições por combinadores $\mathcal{C}\ell(\mathcal{K})$.

O programa transformado terá as mesmas definições globais que o original mas onde poderão ocorrer combinadores $\Phi_{\alpha_1}, \Phi_{\alpha_2}, \ldots, \Phi_{\alpha_n}$ substituindo abstracções-λ. Note-se que cada programa apenas usa um número *finito* e *reduzido* de combinadores Φ: cada abstracção no programa original introduz (quando muito) um combinador Φ.

Vamos tratar as expressões-`let(rec)` de forma distinta conforme ocorrem dentro ou fora do âmbito de λs:

- fora de λs, uma expressão-`let(rec)` é preservada pela tradução: corresponderá a uma *descrição textual de um grafo dirigido* (acíclico no caso de expressões-`let` ou cíclico no caso de expressões-`letrec`);
- dentro de um ou mais λs, uma expressão-`let` é transformada numa aplicação de combinadores para permitir abstrair as variáveis ligadas pelos λs.

Exemplo 7 Consideremos duas definições alternativas da mesma função

```
f x = let y=1+x in y*y ;
f' = \x -> let y=1+x in y*y ;
```

A definição de `f` não é alterada pela tradução para combinadores: define um supercombinador que constroi o grafo acíclico para a expressão `(1+x)*(1+x)` (onde a sub-expressão comum sublinhada é partilhada).

A definição de `f'` pode ser vista como:

```
f' = \x -> (\y -> y*y) (1+x) ;
```

e transformada usando o algoritmo de abstracção (secção 2.3.3, pág. 26) em:

f' = Φ_{bb} ($\Phi_{(b,\epsilon)}$ *) (+ 1) ;

A tradução para combinadores introduziu dois combinadores Φ correspondentes às duas abstracções-λ (a expressão-`let` conta como uma abstracção).

Note-se ainda que a função denotada pelas duas definições é a mesma: a redução de `f'` construirá o mesmo grafo que `f` (incluindo a mesma partilha da sub-expressão `1+x`). Do ponto-de-vista de implementação, contudo, a primeira será mais eficiente pois constroi directamente o grafo sem efectuar duas reduções intermédias de combinadores Φ.

Mais formalmente, vamos definir o esquema de tradução $\mathcal{H}_\lambda$ e um esquema auxilar $\mathcal{H}_{\lambda\texttt{let}}$:

$\mathcal{H}_\lambda$: elimina abstracções-λ duma expressão, transformado-as em termos com combinadores; expressões `let(rec)` e `case` são preservadas.

$\mathcal{H}_{\lambda\texttt{let}}$: elimina abstracções-$\lambda$ e expressões-`let`, transformado-as em termos com combinadores; não são admissíveis expressões `letrec` ou `case`.

3.3.2 O esquema $\mathcal{H}_\lambda$

O esquema $\mathcal{H}_\lambda$ é uma extensão da transposição clássica entre λ-termos e combinadores (cf. equação 2.6, pág. 23). As expressões-`let(rec)` e `case` são transpostas propagando o uso de $\mathcal{H}_\lambda$ para as sub-expressões. No tratamento de uma abstracção-λ usamos o esquema auxiliar $\mathcal{H}_{\lambda\texttt{let}}$ (exposto na secção 3.3.3) e o esquema de abstracção combinatória Abs (exposto na secção 2.3.3, pág. 26).

$$
\begin{array}{lcl}
\mathcal{H}_\lambda [\![\texttt{let(rec)}\ Defs\ \texttt{in}\ E]\!] & = & \texttt{let(rec)}\ (\mathcal{H}_\lambda\, Defs)\ \texttt{in}\ (\mathcal{H}_\lambda\, E) \\
\mathcal{H}_\lambda [\![x_1 = E_1; \ldots; x_n = E_n]\!] & = & x_1 = (\mathcal{H}_\lambda\, E_1); \ldots; x_n = (\mathcal{H}_\lambda\, E_n) \\
\mathcal{H}_\lambda [\![\texttt{case}\ E\ \texttt{of}\ Alts]\!] & = & \texttt{case}\ (\mathcal{H}_\lambda\, E)\ \texttt{of}\ (\mathcal{H}_\lambda\, Alts) \\
\mathcal{H}_\lambda [\![Alt_1 | \ldots | Alt_n]\!] & = & (\mathcal{H}_\lambda\, Alt_1) | \ldots | (\mathcal{H}_\lambda\, Alt_n) \\
\mathcal{H}_\lambda [\![\sigma\ x_1 \ldots x_n \to E]\!] & = & \sigma\ x_1 \ldots x_n \to (\mathcal{H}_\lambda\, E) \\
\mathcal{H}_\lambda [\![\texttt{var}\ x \to E]\!] & = & \texttt{var}\ x \to (\mathcal{H}_\lambda\, E) \\
\mathcal{H}_\lambda [\![\texttt{default} \to E]\!] & = & \texttt{default} \to (\mathcal{H}_\lambda\, E) \\
\mathcal{H}_\lambda [\![U\ V]\!] & = & (\mathcal{H}_\lambda\, U)\ (\mathcal{H}_\lambda\, V) \\
\mathcal{H}_\lambda [\![\backslash x \to E]\!] & = & \mathsf{Abs}\ x\ (\mathcal{H}_{\lambda\texttt{let}}\, E) \\
\mathcal{H}_\lambda [\![c]\!] & = & c, \quad \text{(se } c \text{ é constante ou variável)}
\end{array}
\tag{3.1}
$$

3.3.3 O esquema $\mathcal{H}_{\lambda\texttt{let}}$

No esquema $\mathcal{H}_{\lambda\texttt{let}}$ eliminamos não só abstracções-λ como também expressões-`let`, substituindo ambos os casos por termos em combinadores. O resultado de $\mathcal{H}_{\lambda\texttt{let}}$ é sempre um termo em combinadores-$\mathcal{C}\ell(\mathcal{K})$ com (eventualmente) constantes e variáveis livres.

$$
\begin{array}{lcl}
\mathcal{H}_{\lambda\texttt{let}}\,[\![U\ V]\!] & = & (\mathcal{H}_{\lambda\texttt{let}}\,U)\ (\mathcal{H}_{\lambda\texttt{let}}\,V) \\
\mathcal{H}_{\lambda\texttt{let}}\,[\![\backslash x \to E]\!] & = & \mathsf{Abs}\ x\ (\mathcal{H}_{\lambda\texttt{let}}\,E) \\
\mathcal{H}_{\lambda\texttt{let}}\,[\![c]\!] & = & c, \quad (\text{se } c \text{ é constante ou variável}) \\
\mathcal{H}_{\lambda\texttt{let}}\left[\!\!\left[\begin{array}{ll} \texttt{let} & x_1 = e_1 \\ & x_2 = e_2 \\ & \quad\vdots \\ & x_n = e_n \\ & \ \ \texttt{in}\ E \end{array}\right]\!\!\right] & = & (\mathsf{Abs}\ \texttt{[}x_1, x_2, \ldots, x_n\texttt{]}\ E')\ e'_1\ e'_2 \ldots\ e'_n \\
 & & \text{onde} \\
 & & E' = \mathcal{H}_{\lambda\texttt{let}}\,E \\
 & & e'_i = \mathcal{H}_{\lambda\texttt{let}}\,e_i, \quad i = 1, 2, \ldots n
\end{array}
\tag{3.2}
$$

Uma expressão-`let` (último caso da equação acima) é traduzida como uma aplicação: $\texttt{let}\ x_1 = e_1; \ldots; x_n = e_n\ \texttt{in}\ E$ é equivalente ao λ-termo

$$(\backslash x_1\, x_2 \ldots x_n \to E)\ e_1\, e_2 \ldots\ e_n$$

que se transpõe em combinadores como (cf. equação 2.6, pág. 23)

$$
\begin{array}{rl}
 & (\mathsf{Abs}\, x_1\, (\mathsf{Abs}\, x_2\, (\ldots (\mathsf{Abs}\, x_n\, E') \ldots)))\ e'_1\ e'_2 \ldots\ e'_n\ \equiv \\
\equiv & (\mathsf{Abs}\ \texttt{[}x_1, x_2 \ldots x_n\texttt{]}\ E')\ e'_1\ e'_2 \ldots\ e'_n
\end{array}
$$

onde $E', e'_1 \ldots e'_n$ são recursivamente traduzidos pelo mesmo esquema $\mathcal{H}_{\lambda\texttt{let}}$.

3.3.4 Pré-processamento do programa original

Os casos tratados por $\mathcal{H}_{\lambda\texttt{let}}$ não são exaustivos: não são tratadas expressões-`letrec` nem expressões-`case` (cf. equação 3.2). Uma vez que este esquema auxiliar é apenas usado para o corpo de uma abstracção-λ isto significa que essas expressões não podem ocorrer no âmbito de um λ.

Podemos lidar com estas situações transformando o programa original substituindo essas expressões por outras equivalentes.

Expressões “`\x -> ...(case ... of ...)...`”

Um dos nossos objectivos de geração de código em C é o de transpor uma expressão-`case` de escolha de alternativas directamente para um “`switch`” em C. Contudo, isto apenas é possível quando geramos *código estrito* para expressões-`case`, *i.e.*, apenas nas situações em que sabemos que a expressão-`case` será efectivamente reduzida. Por outro lado, a situação paradigmática de código não-estrito ocorre na compilação do corpo de uma abstracção-λ: o resultado é um grafo correspondente a uma “computação parcial” que só será reduzida quando este for aplicado a um argumento, *i.e.*, a abstracção-λ corresponde à construção duma “*closure*” para uma expressão (Kogge, 1991, secção 6.5).

No entanto, é sempre possível gerar código não-estrito “embrulhando” a expressão-`case` num super-combinador.

Exemplo 8 Consideremos o super-combinador `f`:

```
f x y = let z=x+y in \t -> if t>z then x+t else y+t ;
```

Podemos re-escrever o programa original como:

```
f x y = let z=x+y in \t -> if_then_else (t>z) (x+t) (y+t) ;
if_then_else cond e1 e2 = if cond then e1 else e2 ;
```

Onde `if_then_else` é o super-combinador que “embrulha” a expressão-`case` para booleanos. Abstraindo `t` no corpo de `f` fica então:

f x y = let z=x+y in $\Phi_{((bb,b),b)}$ if_then_else (< z) (+ x) (+ y) ;
```
if_then_else cond e1 e2 = if cond then e1 else e2 ;
```

A alternativa é usar “*lambda lifting*” obtendo um super-combinador `$f` com as variáveis livres na abstracção $\backslash t \to \ldots$ como parâmetros extra (Peyton Jones, 1995) (ver ainda a secção 3.4.2):

```
f x y = let z=x+y in $f x y z ;
$f x y z t = if t>z then x+t else y+t ;
```

Expressões “`\x -> ...(letrec ... in ...)...`”

A outra situação que o esquema $\mathcal{H}_{\lambda\mathtt{let}}$ não contempla é a abstracção de uma variável de uma expressão-`letrec`. Vamos ver como podemos alterar o programa original de forma a evitar esta situação.

Exemplo 9 Consideremos um exemplo genérico de um super-combinador `sc` onde pretendemos abstrair `x` de uma expressão-`letrec` com duas definições mutuamente recursivas:

```
sc ... = ...(\x -> letrec f = e₁
                          g = e₂
                   in E)... ;
```

Podemos substituir as definições recursivas por definições não-recursivas introduzindo dois *combinadores ponto-fixo* `Y1` e `Y2`; a abstracção em `x` passa a ser sobre uma expressão-`let`:

```
sc ... = letrec Y1 = \f g -> f (Y1 f g) (Y2 f g)
                Y2 = \f g -> g (Y1 f g) (Y2 f g)
         in ...(\x -> let f = Y1 (\f g -> e₁) (\f g -> e₂)
                          g = Y2 (\f g -> e₁) (\f g -> e₂)
                      in E)... ;
```

Definimos os dois combinadores ponto-fixo numa expressão-`letrec` no âmbito mais exterior do super-combinador `sc`, fora de qualquer λ; o `letrec` que causava problemas é agora transformado num `let` simples. As abstracções presentes no programa transformado podem agora ser substituídas por combinadores usando $\mathcal{H}_\lambda$.

Do ponto-de-vista de implementação, o código gerado para o `letrec` que define `Y1` e `Y2` irá construir dois grafos mutuamente cíclicos (ver secção 5.2.2, pág. 74). Isto significa que na expressão E, após uma redução de `Y1` e `Y2` obtêm-se grafos directamente cíclicos para `f` e `g` originais, sem mais referências a `Y1` e `Y2`, e ocupando um número finito de células (Turner, 1979b).

O exemplo acima é facilmente generalizável para transformar expressões-`letrec` com n definições mutuamente recursivas, bastando para tal usar n combinadores `Y` adequados.

3.4 Propriedades da tradução

Vamos agora analisar algumas propriedades da tradução da linguagem-fonte para combinadores compactos. Em particular vamos analisar a complexidade da tradução (no sentido de comparar o tamanho dos programas e o número de reduções) e a preservação de "*full-laziness*" na redução.

3.4.1 Complexidade

Tamanho de programas

Para medir o tamanho do programa-fonte devemos contabilizar:

- número m de abstracções-λ;
- número k de aplicações.

Contabilizamos uma expressões-`let` que liga n variáveis locais como equivalente a n abstracções e n aplicações. Não contabilizamos expressões-`case` pois estas não são alteradas pelo esquema de tradução.

Vamos relacionar o número m de abstracções-λ e k de aplicações com o número e comprimento dos índices dos combinadores Φ introduzidos pela tradução em combinadores compactos.

Começamos por notar que o esquema $\mathcal{H}_\lambda$ só modifica abstracções-λ usando os esquemas $\mathcal{H}_{\lambda\texttt{let}}$ e Abs. Por cada abstracção-λ introduzimos quando muito um novo combinador Φ_α pelo que o número de combinadores Φ será linear sobre o tamanho do programa original.

Vamos então analisar o comprimento total l dos índices α desses combinadores (que é uma medida assimptótica do tamanho de código de redução a eles associado — cf. a secção 2.9 e ainda 5.3).

No pior caso, a abstracção de uma variável introduz 3 símbolos no índice do combinador Φ_α por cada aplicação no termo original, e acrescenta uma aplicação (cf. equações 2.12, pág. 27). Assim, tem-se $l = 3k$ e $k' = k + 1$; o resultado de fazer m abstracções sucessivas vem:

$$l = 3k + 3(k+1) + \ldots + 3(k+m-1) = 3mk + \frac{3}{2}m(m-1) \tag{3.3}$$

Isto é, $l \in O(N^2)$ onde consideramos $N = m + k$ uma medida conjunta do comprimento do programa original. Assim o comprimento dos índices dos combinadores introduzidos pela abstracção é, no pior caso, quadrático sobre o comprimento do programa original.

Abstracções sobre expressões-`letrec`

A análise feita não contabiliza as abstracções sobre expressões-`letrec`: nessas situações a introdução de combinadores ponto-fixo Y pode piorar a complexidade da tradução.

Contabilizando localmente a transformação de uma definição recursiva com n ligações:

$$\begin{array}{ll} \texttt{letrec} & x_1 = e_1 \\ & \vdots \\ & x_n = e_n \\ \texttt{in} & E \end{array} \quad \text{em:} \quad \begin{array}{ll} \texttt{let} & x_1 = \mathsf{Y}_1\ e'_1 \ldots e'_n \\ & \vdots \\ & x_n = \mathsf{Y}_n\ e'_1 \ldots e'_n \\ \texttt{in} & E \\ \multicolumn{2}{l}{\text{onde } e'_i = \backslash x_1 \ldots x_n \to e_i} \end{array} \tag{3.4}$$

Introduzimos n^2 abstracções para obter os e'_i (pois $1 \leq i \leq n$ e para cada i abstraímos n variáveis) e n^2 aplicações dos e'_i aos combinadores ponto-fixo.

Assim, substituindo $m = n^2$ na equação 3.3, vemos que os índices dos Φs podem crescer no pior caso

$$l = 3n^2k + \frac{3}{2}n^2(n^2 - 1) \in O(n^4)$$

ou seja, a abstracção de uma definição recursiva de n ligações pode obter índices de comprimento n^4.

Se consideramos então que a métrica do comprimento do programa-fonte inclui o número n de ligações em expressões-`letrec`, concluímos que, no pior-caso, o comprimento dos índices dos combinadores Φ pode ser $O(n^4)$.

Contudo, esta situação não ocorre para os `letrec`s fora de λs (*i.e.*, aos quais não abstraímos variáveis) pois nesses casos os esquemas de tradução preservam a complexidade da expressão original.

Número de reduções

Vamos agora relacionar o número de reduções por ordem normal no programa original com as equivalentes reduções em combinadores no programa transformado. No programa original ocorrem:

- reduções de super-combinadores (*i.e.*, de definições globais) e de combinadores primitivos (*i.e.*, operações aritméticas);
- reduções de β-*redexes*: $(\backslash x \to E)e$;
- reduções em expressões-`let(rec)`.

A tradução $\mathcal{H}_\lambda$ preserva super-combinadores e combinadores primitivos e portanto estas reduções são mantidas no programa transformado. Notamos ainda que cada aplicação $U\ V$ se transforma numa aplicação $(\mathcal{H}_\lambda\, U)\ (\mathcal{H}_\lambda\, V)$ entre combinadores.

Como cada abstracção-λ introduz, no pior caso, um novo combinador Φ, cada β-redex no programa original fica um *redex* fraco com um combinador Φ à cabeça.

As expressões-`let` com n definições contabilizam, no pior caso, n reduções em combinadores. As expressões-`letrec` e `case` fora de λs são preservadas pela tradução. Nestes casos o número de reduções no programa original é preservado no programa em combinadores.

No caso de efectuarmos abstracções sobre uma expressão-`letrec` com n ligações o número de reduções pode aumentar (cf. equação 3.4): a redução de cada x_i implica reduzir Y_i mais n reduções associadas às n abstracções de e_i' (no pior caso). Assim a redução de n ligações recursivas pode traduzir-se em $n(n+1)$ reduções de combinadores.

3.4.2 "*Full-laziness*"

A implementação duma linguagem funcional pura deve evitar duplicar reduções do mesmo *redex*: o resultado duma expressão é único e pode ser usado em todas as suas ocorrências (esta propriedade é usualmente designada de *transparência referencial*).

Embora a redução de grafos forneça um mecanismo eficiente para partilhar sub-expressões comuns, mesmo estas implementações nem sempre partilham um resultado já reduzido, levando a uma perda de eficiência. Em particular isto ocorre quando o compilador transforma o programa original, por exemplo, na obtenção de super-combinadores por "*lambda-lifting*" (Peyton Jones, 1987).

Uma implementação que nunca duplique as reduções expressas no programa original designa-se "*full-lazy*". A noção de "*full-laziness*" surgiu originalmente associada aos algoritmos de "*lambda-lifting*" mas é de facto uma propriedade independente desta transformação, como é defendido em (Peyton Jones, 1995).

Começamos por apresentar um exemplo retirado de (Peyton Jones, 1995) de uma função em que a redução perde "*laziness*" quando a transformação de "*lambda-lifting*" é efectuada de forma directa, *i.e.*, sem qualquer optimização. Também mostramos que na nossa tradução em combinadores tal perda não ocorre.

Exemplo 10 Consideremos a função `f` definida por:

```
f = \x -> let g = \y -> x*x+y
              in (g 3 + g 4) ;
```

Efectuando "*lambda-lifting*" da abstracção em `y` definimos um novo super-combinador `$g` com as variáveis livres na abstracção como argumentos, a saber `x` e `y`:

```
f = \x -> let g = $g x
```

```
          in (g 3 + g 4) ;
$g x y = x*x + y ;
```

A função original `g` inclui uma expressão `x*x` que poderia ser reduzida e partilhada nas duas chamadas `(g 3)` e `(g 4)`. Contudo, o programa transformado perde "*laziness*" pois `($g x)` não é redutível e por isso as duas chamadas vão reduzir duas vezes `x*x`. Para evitar esta situação seria suficiente que `$g` fosse definido com `x*x` como argumento e não apenas `x`, *i.e.*, *definir o super-combinador com sub-expressões livres maximais* e não apenas variáveis livres (Peyton Jones, 1995).

Por outro lado, vejamos que a abstracção em combinadores preserva uma só redução de `x*x`. Para transpor a definição de `f` em combinadores, as expressões-`let` são vistas como abstracções e aplicações, e as variáveis abstraídas pela ordem no programa:

```
f = \x -> (\g->g 3 + g 4) (\y->x*x+y)
  = Abs x ((Abs g (g 3 + g 4)) (Abs y (x*x+y)))
  = Abs x ((Φ_(bc,c) (+) 3 4) ((+) (x*x)))
  = Φ_bb(b,ε) (Φ_(bc,c) (+) 3 4) (+) (*)
```

A redução deste termo em combinadores efectuar-se-á pela sequência inversa destas abstracções:

- o combinador exterior $\Phi_{bb(b,\epsilon)}$ constrói um *redex* para `x*x`;
- o combinador interior $\Phi_{(bc,c)}$ aplica o *redex* `x*x` a dois termos correspondentes a `(g 3)` e `(g 4)`; contudo, os dois grafos vão *partilhar um apontador* para o mesmo *redex*, que portanto será reduzido uma só vez (cf. secção 4.2).

O exemplo anterior suscita a questão de saber até que ponto a abstracção em combinadores preserva "*full-laziness*": a resposta é que a abstracção preserva "*laziness*" explícita no programa original mas não optimiza reduções de forma a obter "*full-laziness*" no sentido dado em (Peyton Jones, 1995).

Exemplo 11 Consideremos a função `f` de um outro exemplo em (Peyton Jones, 1995):

```
f = \x -> let g = \y -> (let z=x*x
                         in let p=z*z
                         in p+y)
          in (g 3 + g 4) ;
```

Nesta definição `z` e `p` dependem só de `x`, pelo que poderiam ser reduzidos uma só vez para as duas chamadas `(g 3)` e `(g 4)`. No entanto, quando transpomos a definição de `f` em combinadores obtemos:

$$
\begin{aligned}
&\texttt{f = \textbackslash x -> (\textbackslash g->g 3 + g 4) } \underline{\texttt{(\textbackslash y->(\textbackslash z->(\textbackslash p->p+y) z*z) x*x}}\texttt{)}\\
&\quad \texttt{= ...}\\
&\quad \texttt{= Abs x ((}\Phi_{(bc,c)}\texttt{ (+) 3 4) (}\underline{\Phi_{ccbcb}\texttt{ (x*x) (*) }\Phi_{b(b,\epsilon)}\texttt{ (+) }\Phi_{cb}\texttt{)}}\texttt{) = ...}
\end{aligned}
$$

O termo em combinadores sublinhado corresponde à abstracção em `y` sublinhada na definição original, *i.e.*, a definição de `g`.

Neste termo há uma só ocorrência do *redex* `(x*x)` associado a `z`, mas o *redex* `(z*z)` associado a `p` será construído durante redução dos combinadores Φ. Assim, as duas reduções de `(g 3)` e `(g 4)` irão construir dois *redexes* para `p` e, portanto este será reduzido duas vezes.

O que causa a duplicação do *redex*? O facto do programa original implicitamente indicar uma falsa dependência de `z` e `p` em relação à abstracção em `y`. Podemos re-escrever o programa noutro equivalente que evita essa falsa dependência:

```
f = \x -> let z=x*x
          in let p=z*z
          in let g = \y->p+y
                in (g 3 + g 4) ;
```

Nesta forma ambos os *redexes* `(x*x)` e `(z*z)` serão criados antes da redução de `(g 3)` e `(g 4)`, como a simples transposição em λ-calculus evidencia:

```
f = \x -> (\z->((\p->(\g->g 3 + g 4) (\y->p+y)) z*z) x*x)
```

Esta solução para evitar a duplicação de reduções é apontada em (Peyton Jones, 1995) no contexto do "*lambda-lifting*": *"migrar" as definições locais-*`let` *para fora do âmbito de abstracções-λ em que são livres.*

Em conclusão, a abstracção em combinadores compactos preserva "*laziness*" patente no programa original, embora não elimine as falsas dependências de forma a obter "*full-laziness*".

A "*full-laziness*" pode ser vista como uma optimização análoga à detecção de sub-expressões comuns numa linguagem imperativa clássica, onde uma expressão comum a dois procedimentos pode ser calculada uma só vez e o resultado partilhado entre duas chamadas.

Capítulo 4

Transposição para linguagem C

Neste capítulo descrevemos um modelo de redução de grafos em combinadores implementado em linguagem C. O programa em combinadores é traduzido num programa em C que pode ser compilado para linguagem máquina usando um compilador de C usual e executado separadamente.

A linguagem C é usada frequentemente como um passo intermédio na compilação de linguagens de alto-nível em compiladores académicos (Peyton Jones, 1993; M. Jones, 1994). Esta abordagem permite gerar código que é directamente executável na máquina, sem o peso adicional de interpretação de pseudo-código, e de uma forma independente do "*hardware*" subjacente: a linguagem C funciona como uma "linguagem de assemblador portável".

A compilação para linguagem C é obtida pela geração de código de uma *máquina abstracta*. Esta máquina abstracta é suficientemente baixo-nível para ser traduzida directamente em linguagem C, ou inclusive para a linguagem-máquina de um processador convencional.

Por outro lado, a máquina abstracta fornece um modelo de execução adequado à *redução de grafos* de combinadores, abstraindo as características da arquitectura física dum processador real como o número físico de registos, modos de endereçamento, alinhamento de memória, entre outras.

4.1 A máquina abstracta-ΦG

A máquina abstracta para a qual nos propomos gerar código é uma variação sobre a máquina abstracta clássica para implementação de redução de grafos, a "*G-machine*" (Peyton Jones, 1987; Peyton Jones & Lestes, 1992). A nossa variação designa-se *máquina*-ΦG[1].

[1]A pronuncia deve ser "máquina *fi-ji*".

Tal como a "*G-machine*", a máquina-ΦG foi concebida para executar código compilado de redução de grafos com combinadores: a cada definição de um super-combinador no programa-fonte e a cada um dos combinador Φ_α introduzido pelo processo de abstracção (apresentado na secção 3.3) irá corresponder uma sequência de instruções da máquina abstracta que implementará a redução associada.

4.1.1 Arquitectura

A máquina-ΦG tem uma arquitectura mista de *pilha* e *registos* (ao contrário da "*G-machine*" que é exclusivamente de pilha).

As suas estruturas de dados fundamentais são:

- o *grafo*, representando os termos do programa a reduzir, guardado numa área "*heap*" comum (ver secção 4.3);
- a *pilha de argumentos*, um vector de apontadores para nós do grafo, usada para aceder aos argumentos de um combinador;
- um conjunto ilimitado de *registos* `r0`, `r1`, `r2`, ... usados como apontadores temporários para a construção de partes do grafo; note-se que, embora conceptualmente ilimitado, cada programa usará apenas um número *finito* destes registos;
- a *pilha de execução* que mantém os contextos de activação de cada invocação de um combinador.

Na nossa implementação, a área de pilha de argumentos e a área de "*heap*" são reservadas como blocos de memória contíguos.

A pilha de argumentos é um vector de *apontadores* para nós no grafo. O registo `Stk` aponta para o topo da pilha, que cresce para endereços mais baixos. Assim, o endereço no topo da pilha será `Stk[0]`, o 2º endereço na pilha `Stk[1]` e assim sucessivamente. O registo `Stk` corresponde a uma variável global na linguagem C.

A área de "*heap*" é gerida com uma lista ligada de blocos livres; um algoritmo de recolha de lixo é responsável por refazer a lista quando se esgota o espaço livre, marcando os nós efectivamente usados e libertando os não-usados (ver secção 4.3).

Os registos `r0`, `r1`, `r2`, ... usados em cada bloco de código (uma função em C) são implementados como variáveis locais declaradas com atributo "`register`".

4.1.2 Instruções da máquina abstracta

As instruções da máquina abstracta dividem-se em quatro grupos principais:

- construção do grafo;
- manipulação da pilha de argumentos;
- transferência de execução;
- operações primitivas (como aritmética entre inteiros).

As instruções são de tipo "três endereços" (Aho et al., 1986) tomando até três operandos que podem ser registos, posições na pilha ou identificadores globais do grafo. Instruções que efectuam transferência de execução tomam ainda *endereços de código* (apontadores para funções em C).

A sintaxe duma instrução típica de construção do grafo é uma de uma das duas formas:

$$\texttt{opcode}_3(arg_0,\ arg_1,\ arg_2)$$
$$\texttt{opcode}_2(arg_0,\ arg_1)$$

cujas semânticas são respectivamente

$$arg_0 := (arg_1\ \texttt{opcode}_3\ arg_2)$$
$$arg_0 := (\texttt{opcode}_2\ arg_1)$$

Exemplo 12 A sequência de código seguinte constrói o grafo correspondente ao termo $(f\ 1\ 2)$; G_f é um identificador global que representa o combinador f (ver secção 4.1.5, pág. 57).

```
Mkint(r0, 1);              /* r0 := new INTG(1) */
Mkap(r0, G_f, r0);         /* r0 := new AP(f,r0) */
Mkint(r1, 2);              /* r1 := new INTG(2) */
Mkap(r0, r0, r1);          /* r0 := new AP(r0,r1) */
```

No final desta sequência fica no registo `r0` um apontador para o grafo construído.

A maioria das instruções da máquina-ΦG correspondem a uma ou mais instruções em C, expandidas directamente pelo pré-processador a partir de "*macros*"; instruções mais complexas são codificadas em pequenas funções em C.

O apêndice A contém uma descrição completa do conjunto de instruções da máquina-ΦG.

4.1.3 Estrutura de código

A representação de código de uma máquina abstracta dentro da linguagem C depara com a necessidade de representar endereços de sequências de código. Mais concretamente, é necessário:

1. Obter um endereço para uma sequência de instruções;
2. Guardar um endereço numa variável ou estrutura em memória;
3. Transferir a execução para um endereço (determinado em tempo de compilação ou guardado numa variável).

Várias soluções para este problema foram exploradas em diferentes compiladores que geram linguagem C. Em particular, na descrição da máquina abstracta "*STG-machine*" em (Peyton Jones, 1993) expõem-se algumas soluções:

Usar um "`switch`" global

A primeira solução consiste em representar os endereços como *inteiros*; o código é gerado numa única função com um "`switch`" que associa os valores inteiros às sequências de instruções:

```
int cont = 1;
while(TRUE) {
    switch(cont) {
        case 1:  ...        /* código para endereço 1 */
        case 2:  ...        /* código para endereço 2 */
        /* etc.  */
    }
}
```

Esta solução tem várias limitações: em primeiro lugar, a transferência de execução é interpretada indirectamente e não traduzida em instruções "*jump/call*" do processador físico. Em segundo lugar, não permite facilmente compilação separada de partes do programa fonte. Por último, a geração de uma única função de grande tamanho sobrecarrega o compilador de C: a linguagem C encoraja a decomposição dum programa em pequenas funções e os compiladores têm maior facilidade em fazer alocação de registos e "*scheduling*" de instruções nessa situação.

Uma variação sobre esta solução consiste em usar uma extensão à linguagem C implementada pelo compilador GNU (Stallman et al., 1997) que permite colocar "*labels*" locais em variáveis e efectuar saltos locais para o conteúdo de uma variável (usando "`goto *`"). Tal optimização permite obviar a primeira desvantagem enunciada acima, mas as restantes permanecem. Além disto, essa optimização é específica ao compilador de C GNU (embora este esteja extremamente difundido na comunidade académica).

Usar retorno de continuações

A segunda solução consiste em representar os endereços como *endereços de funções* em C: cada bloco-sequêncial de instruções corresponde a uma função em C cujo valor de retorno é o endereço da continuação de execução:

```
void *l1(void) {
    ...                    /* código para l1 */
    return &l...;          /* continuação de l1 */
}
void *l2(void) {
    ...                    /* código para l2 */
    return &l...;          /* continuação de l2 */
}
/* etc.  */
```

A função principal `main` apenas executaria um ciclo chamando a continuação de cada bloco:

```
typedef void *(code_ptr)(void); /* tipo para endereços */
code_ptr cont = &l1;
while(TRUE) cont = (*cont)();
```

Esta solução tem a vantagem de representar endereços como endereços da máquina física; tem ainda a vantagem de gerar várias funções cuja compilação é independente e que podem ser ligadas por um "*linker*" convencional.

Tem, contudo, desvantagens importantes: uma vez que as funções geradas têm que corresponder a *blocos-básicos*[2] de instruções esta solução tende a gerar muitas funções pequenas (com uma ou duas instruções da máquina abstracta); o custo de efectuar uma chamada de função e construir o contexto de activação é então significativo em relação à execução de código propriamente dito.

Em segundo lugar, todas as transferências de execução são efectuadas regressando primeiro ao ciclo em `main` e invocando depois a continuação com um salto indirecto, o que reduz o

[2] Usamos *bloco-básico* no sentido de uma sequência contígua de instruções da máquina abstracta, com uma único ponto de entrada (o início) e um único ponto de saida (o final), conforme definido em (Aho et al., 1986).

desempenho nas arquitecturas modernas de microprocessores RISC e mesmo CISC (Henessy & Patterson, 1996).

É possível evitar o custo dos saltos indirectos gerando uma "*macro*" `Jump(l)` que seria expandida para "`return &l`" numa implementação genérica ou para uma sequência específica de instruções de assemblador que efectua um salto directo (usando, por exemplo, a directiva `asm` do compilador GNU que permite introduzir instruções de assemblador entre o código C).

Esta optimização, além de obviamente depender da linguagem assemblador do processador físico, depende ainda das convenções de chamada e salvaguarda de registos do compilador. Ao nível do código C nem sempre é possível inferir exactamente quais os registos guardados na pilha e que devem ser restaurados antes de transferir a execução.

Usar a pilha de execução de C

A terceira solução envolve representar directamente a pilha de chamada/retorno da máquina abstracta usando a pilha de execução da linguagem C:

- Cada combinador do programa-fonte dá origem a uma função em C sem parâmetros e sem valor de retorno (*i.e.*, `void ...(void)`);
- Todos os argumentos são passados explicitamente na pilha de argumentos da máquina abstracta; o resultado da redução é a re-escrita do grafo;
- A redução recursiva de um sub-grafo (*i.e.*: execução da instrução "`Eval`") corresponde à chamada duma função em C (ou seja, a pilha de execução da máquina abstracta é representada implicitamente na pilha de execução de C);
- Um salto incondicional da máquina abstracta (instrução "`Jump(f)`") é implementado por uma sequência "`f(); return;`"

Tal como a anterior, esta solução representa endereços de código como endereços físicos de funções. Além disso, o código gerado por cada função não é apenas um bloco básico, podendo incluir saltos condicionais e alternativas.

De facto, uma expressão `case` pode ser implementada em C como um "`switch`" sobre a palavra identificadora dum objecto; por exemplo, o corpo da função `length` seria compilado como:

```
Eval(Stk[0]);
Switch(Stk[0]) {
```

```
    case CTAG_Nil:
        /* alternativa para construtor [] */
        ⋮
    case CTAG_Cons:
        /* alternativa para construtor :  */
        ⋮
}
```

Esta solução permite igualmente a compilação separada de código para combinadores e que pode ser ligado por um "*linker*" convencional.

A desvantagem (importante) desta solução está no consumo descessário da pilha de execução de C no caso de um salto incondicional: se a função `f1` termina com `Jump(f2)` o código gerado seria algo como:

```
void f1(void) {
    ...
    f2();       /* salto incondicional para f2 */
}
```

A execução em C não corresponde a um salto incondicional mas sim à chamada de uma função, o que consome mais um contexto de activação na pilha.

Como `f1` e `f2` não têm argumentos seria extremamente simples efectuar a optimização da última chamada ("*last call optimization*"): remover o contexto de activação de `f1` (deixando apenas o endereço de retorno) e transferir a execução para `f2`.

Infelizmente, tal optimização não é normalmente efectuada pelos compiladores de C (não é efectuada, por exemplo, pelo compilador GNU). É possível, contudo, efectua-la com um pós-processamento do texto de assemblador produzido pelo compilador de C (para mais detalhes remetemos para o apêndice B).

Concluimos justificando a nossa escolha pela terceira solução para a estrutura do código:

1. Permite associar endereços físicos às sequências de código;
2. Permite compilação separada;
3. Permite gerar funções de "granularidade alta" (*i.e.*: uma função por combinador);
4. Podemos optimizar os saltos incondicionais pós-processando o código assemblador sem alterar a geração de linguagem C (ver apêndice B).

4.1.4 Representação do grafo

O grafo é representado como um conjunto de *nós* interligados por apontadores; os nós são mantidos numa área de memória comum designada "*heap*".

Cada nó é constituido por uma sequência contígua de palavras da máquina subjacente (*i.e.*, o tamanho de um "`int`": 32-bits ou mais nos computadores pessoais e estações-de-trabalho correntes).

A primeira palavra de cada nó inclui ainda um campo de "*tag*" que identifica qual o nó em causa e inclui informação usada para gestão de memória (ver secção 4.3).

Os nós são de quatro tipos distintos:

INTG: um valor inteiro de tamanho limitado à palavra da máquina;

CC: (Código de Combinador) o endereço do código associado a um combinador (*i.e.*, o endereço de uma função em C);

AP: um nó de aplicação de uma função a um argumento;

PACK: uma estrutura correspondente a um tipo algébrico com um número variável de sub-componentes.

Os formatos dos nós em memória na nossa implementação são ilustrados na figura 4.1, onde cada palavra é precedida do seu índice relativo ao início do nó.

Nó-`CC`

Associamos um nó-`CC` a cada combinador do programa (quer seja definido pelo programador, resultado do algoritmo de abstracção ou uma operação primitiva).

O nó-`CC` contém um apontador para o código de redução do combinador. Este nó ocupa duas palavras para permitir actualizações correctas na redução de combinadores-CAF[3].

Para evitar a duplicação da redução de um combinador-CAF é necessário actualizar o resultado numa localização globalmente acessível de todos os nós com referências ao combinador. Uma vez que o segmento de código não é, em princípio, alterável durante a execução do programa, a solução é introduzir o nó-`CC` que serve de indirecção para o código do combinador ou para o resultado da redução (ver ainda secção 4.2).

Associamos *um só* nó-`CC` a cada combinador (e não um por cada *uso* do combinador). Assim

[3] "*Constant Applicative Form*": um combinador de aridade zero, *i.e.*, sem argumentos.

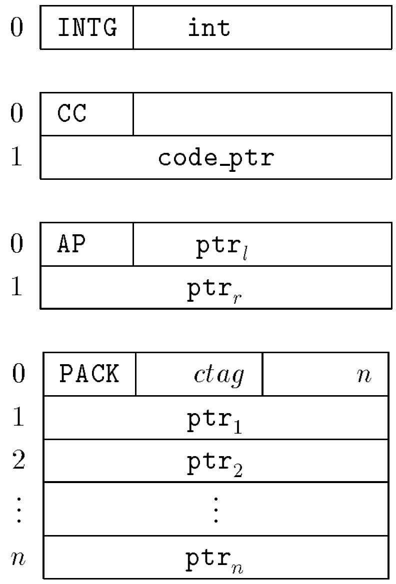

Figura 4.1: Formato dos nós do grafo em memória.

os nós-`CC` são reservados estaticamente antes do início da execução do programa propriamente dito.

Uma vez que os nós-`CC` são necessários nos combinadores-CAFs, optamos por usá-los em *todos* os combinadores: isto permite que os testes aos tipo dos nós sejam mais simples e eficientes.

Nó-`PACK`

Os nós-`PACK` têm tamanho variável, dependendo da aridade do construtor associado: se o construtor tem aridade n o nó-`PACK` ocupa $n + 1$ palavras. Assim, por exemplo, um nó do construtor de listas "`:`" ocupa 3 palavras, enquanto um nó de um valor booleano "`True`" ou "`False`" ocupa apenas uma palavra.

A 1ª palavra de uma estrutura contém, para além da "*tag*" que identifica o nó-`PACK`, uma outra "*tag*" que identifica o construtor (*ctag*) e a aridade (n). A *ctag* é necessária para identificar o construtor durante a execução do programa; a aridade n é apenas utilizada pelo sistema de gestão de memória (ver a secção 4.3).

4.1.5 Globais e primitivas

Os supercombinadores no programa-fonte e os combinadores primitivos da linguagem (incluindo os combinadores Φ) são representados no sistema de execução da máquina-ΦG em dois níveis: como *código em C* que efectua a redução de grafos e como *nó-global* usado para construir outros grafos. Um supercombinador ou primitiva f no programa-fonte tem associados:

- a função `C_`f que efectua a redução de grafos;
- a variável global `G_`f que aponta para o nó-`CC` usado para construir grafos.

Tanto a função como o nó-`CC` são reservados estaticamente antes de iniciar a redução de grafos propriamente dita: o código em C dos combinadores primitivos faz parte de uma biblioteca do sistema de execução; o código de redução dos supercombinadores e combinadores Φ é gerado pelo compilador. Em qualquer dos casos, os nós-`CC` associados são reservados antes de se iniciar a redução de `main` no programa-fonte.

4.2 Redução de grafos

O código de redução de grafos associado a cada combinador usa uma pilha de apontadores para aceder aos seus argumentos; o apontador para o topo da pilha é mantido numa variável global `Stk`.

Usamos um outro apontador `Frame` para a base do contexto actual na pilha: os argumentos acessíveis à execução dum combinador estão balizados entre `Frame` e `Stk`. Isto permite utilizar a mesma pilha de argumentos quando se efectua a redução recursiva de uma sub-expressão (*i.e.*, um sub-grafo).

Inicialmente temos `Frame` = `Stk`, ou seja, o contexto inicial é vazio.

4.2.1 "*Unwind*": Encontrar o *redex* fraco

A redução do grafo associado a um termo $(f\ A_1\ A_2 \ldots A_n)$ inicia-se com uma descida recursiva pelo lado-esquerdo dos nós-`AP` designada "*unwind*" do grafo:

1. Colocamos um apontador para o nó inicial na pilha de argumentos;
2. Efectuamos a descida no grafo pelos lados-esquerdos, colocando na pilha apontadores para os nós-`AP` visitados.

A descida no grafo termina quando atingimos um nó *não*-`AP`: se o programa-fonte estiver correctamente tipado então esse nó deverá ser o nó-`CC` correspondente ao combinador f. A figura 4.2 ilustra a configuração na pilha após o final da descida no grafo (onde `@` representa um nó-`AP`).

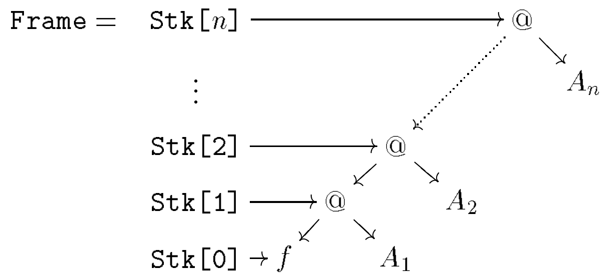

Figura 4.2: Configuração da pilha de argumentos após a descida no grafo.

A instrução `Unwind` da máquina-ΦG efectua um "*unwind*" a partir do grafo em `Frame`. A sua implementação é feita numa pequena função em C `doUnwind()` cujo valor de retorno é o endereço de código do combinador a reduzir em seguida (cf. apêndice A, secção A.4.1).

Notamos ainda que caso o grafo em `Frame` não for um nó-`AP` ou `CC` então o grafo já está em forma-normal fraca e a redução está concluída.

4.2.2 Redução do *redex* fraco

Supondo então que encontramos um *redex* fraco e o endereço de código do combinador f associado, executamos o código de f.

O número de argumentos colocados na pilha é $n =$ `Frame` $-$ `Stk` (pois a pilha cresce para baixo e o elemento `Stk[0]` aponta o nó-`CC`).

O código de redução de f efectua então os seguintes passos:

1. Verifica se dispõe do número necessário de argumentos na pilha para efectuar a redução;
2. Re-estrutura os apontadores na pilha de forma a referirem directamente os argumentos (e não os nós-`AP` intermédios).

Usamos duas instruções da máquina abstracta para efectuar estas operações. Suponhamos então que f é um combinador de k argumentos (conhecido em tempo de compilação); o código para f iniciar-se-ia com:

```
Label(C_f)          /* início da função em C associada:  C_f */
      ArgsCheck(k); /* verificar se # argumentos>=k */
      Mkargs(k);    /* re-estruturar os k argumentos */
      ⋮
EndLabel            /* fim da função em C */
```

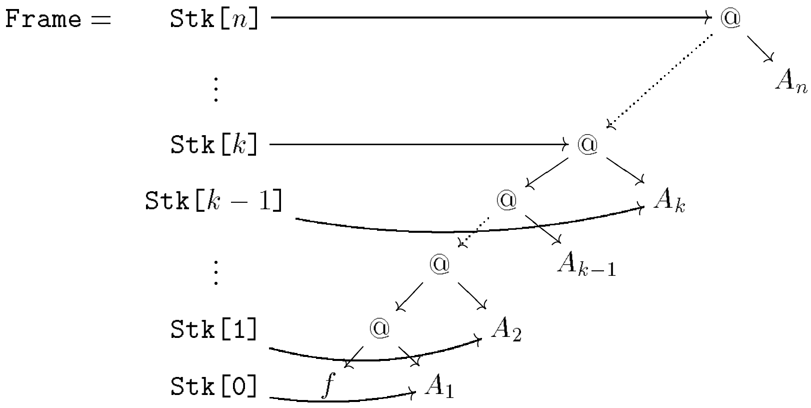

Figura 4.3: Configuração da pilha após execução de `Mkargs(`k`)`.

Em primeiro lugar `ArgsCheck(`k`)` verifica se `Frame` $-$ `Stk` $< k$: nesse caso o combinador não tem um número necessário de argumentos, logo o termo original já está em forma-normal fraca, e a redução termina (*i.e.*, a função `f` retorna imediatamente).

Caso o número de argumentos seja suficiente, `Mkargs(`k`)` vai re-estruturar k endereços a partir do topo da pilha de forma a referirem directamente os k argumentos do combinador. A figura 4.3 ilustra a configuração da pilha após a execução desta instrução.

Note-se que em geral podemos ter `Frame` $-$ `Stk` $> k$, *i.e.*, podemos ter *mais* argumentos na pilha do que a aridade do combinador f: a linguagem permite a definição de funções de ordem superior, pelo que o resultado de f pode ser uma função que utiliza os argumentos deixados na pilha.

Durante a execução do código do combinador os seus k argumentos são acedidos por `Stk[0]`, `Stk[1]`, ..., `Stk[`$k-1$`]`; o código termina escrevendo o resultado sobre o nó em `Stk[`k`]` que corresponde à raiz do termo $(f\ A_1\ A_2 \ldots A_k)$. Note-se que a redução do termo original pode ainda não estar concluída (se o resultado de f for uma função) pelo que a redução continua com um novo "*unwind*" do grafo a partir de `Frame`.

A opção por delegar o trabalho de verificação e re-estruturação da pilha no código de cada combinador tem algumas vantagens:

1. Permite não gerar esse código nas situações em que não é necessário (*i.e.*, combinadores sem argumentos);

2. Permite usar combinadores "especiais" para implementar funcionalidades extra: por exemplo, no combinador "indirecção" usado para representar indirecções no grafo (ver secção 4.2.3).

4.2.3 Actualização da raiz do *redex*

Após a redução de um combinador é necessário actualizar o grafo com o resultado: a redução de um grafo deve ser efectuada apenas uma vez e o resultado partilhado por todos os nós com referências ao grafo original, de forma a garantir que não há duplicação de reduções.

Para garantir que todas as referências ao nó original são re-direccionadas para o resultado devemos escreve-lo *sobre* o nó original. Como em geral isto pode não ser possível (se o resultado ocupa mais espaço que o nó original, ou se o resultado é um apontador e não um nó recém criado), torna-se necessário a colocação de uma *indirecção* para o resultado: para tal usamos uma instrução da máquina-ΦG `Update(`*dest*`,`*src*`)` que escreve sobre o nó apontado por *dest* uma indirecção para o nó apontado por *src*.

O código de um combinador f com k argumentos terminará a sua redução com uma instrução `Update(Stk[`k`],r`m`)` onde `r`m é o registo da máquina abstracta que contém o (apontador para) o resultado; note-se ainda que `Stk[`k`]` aponta sempre a raiz da redução num combinador com k argumentos.

Implementamos uma indirecção como um nó-`AP` em que o lado-esquerdo é um combinador especial `G__indirect` e o lado-direito é o resultado. O código que reduz o combinador-indirecção faz parte da biblioteca de primitivas do sistema de execução.

A redução de `G__indirect` consiste em colocar o seu (único) argumento para o topo da pilha e prosseguir reduzindo o resultado:

```
Label(C__indirect)
     Mkarg(0);     /* há sempre um só argumento */
     Slide(1);     /* desliza o argumento para o topo da pilha */
     Unwind();     /* continua a redução */
EndLabel
```

Para garantir o tratamento uniforme de indirecções usando este combinador `G__indirect` é necessário que qualquer nó que possa ser raiz de uma redução ocupe (pelo menos) o tamanho de um nó-`AP` (caso contrário não seria possível colocar um nó-`AP` sobre a raiz); isto é trivialmente verdade para um combinador com um ou mais argumentos (pois então a raiz é exactamente um nó-`AP`). A única outra possibilidade é um combinador sem argumentos (*i.e.*, um combinador-CAF: "*constant applicative form*"): nessa situação a raiz é o próprio nó-`CC`. A actualização correcta de CAFs justifica então a opção indicada na secção 4.1.4 de reservar duas palavras para cada nó-`CC`.

A secção 5.4 descreve algumas optimizações locais que evitam a utilização de `Update` quando o resultado da redução é um valor básico que pode ser directamente construído sobre a raiz da redução. A secção 4.3 descreve também uma optimização que remove indirecções no algoritmo de recolha de lixo.

4.2.4 "*Eval*": redução de um sub-grafo

Na redução não-estrita, os argumentos de uma função não são reduzidos antes de entrar no código de redução da função. Isto implica que o código de um combinador deve explicitamente invocar uma instrução `Eval(Stk[`i`])` para reduzir o seu i-ésimo argumento na pilha.

Mais geralmente, usamos `Eval(`l`)` para reduzir até à forma-normal fraca o grafo apontado por l (onde l é uma localização qualquer).

A implementação de `Eval` requer que preservemos o contexto actual (registo `Frame`) e inicializemos `Frame=Stk` (*i.e.*, um contexto vazio) antes de efectuar o "*unwind*" do grafo a reduzir. Após a redução do sub-grafo l devemos restaurar `Frame` anterior e continuar a execução de código.

Na implementação clássica da "*G-machine*" (Peyton Jones, 1987; Peyton Jones & Lestes, 1992), o registo `Frame` bem como o contador de programa são preservados num pilha auxiliar "*dump*". Na nossa implementação a pilha "*dump*" é representada implicitamente pela pilha de execução de C (cf. apêndice A, secção A.4.2):

- usamos uma variável local para guardar `Frame`;
- invocamos a função associada à redução do sub-grafo com uma chamada de função usual em C; o seu retorno continuará a execução do código do combinador original.

Notamos ainda que o apontador l pode ser modificado pela execução de `Eval(`l`)`; isto deve-se não à redução propriamente dita, mas a uma eventual execução duma recolha de lixo durante a redução do sub-grafo (ver a secção 4.3).

4.3 Gestão de memória "*heap*"

A área de livre de "*heap*" é gerida como uma lista ligada de blocos. Inicialmente toda a área de "*heap*" corresponde a um só bloco; à medida que a execução do programa requisita novos nós este bloco vai sendo fragmentado.

O código da máquina-ΦG não liberta explicitamente blocos de memória (como é usual num esquema de execução baseado em redução de grafos): quando a área de "*heap*" livre se esgota executa-se um algoritmo de *recolha de lixo* que recupera blocos que já não estão em uso (são "lixo").

O método de recolha de lixo faz parte de um conjunto de procedimentos auxiliares do sistema de execução da máquina-ΦG e que inclui rotinas de reserva de memória "*heap*". Na nossa implementação usamos um algoritmo de recolha de lixo de tipo "marcação/varrimento" clássico (Knuth, 1971) que "marca" todos os nós do grafo efectivamente em uso e recupera a área de "*heap*" complementar.

Em princípio, algoritmos diferentes (por exemplo: cópia "*two-space*" com compactação ou geracionais) poderiam facilmente ser implementados, pois o código da máquina-ΦG é independente do modelo de gestão de memória subjacente (não pressupõe, por exemplo, que os endereços de nós não-globais sejam preservados após uma recolha de lixo).

4.3.1 Como se efectua a recolha de lixo

Para efectuar uma recolha de lixo é necessário:

1. Percorrer todos os possíveis apontadores para nós do grafo;
2. Marcar todos os nós acessíveis a partir de cada um destes apontadores (fase de *marcação*);
3. Percorrer toda a área de "*heap*" re-construindo a lista de blocos livres com os nós não-marcados (fase de *varrimento*).

4.3.2 Marcação

Para marcar cada nó usamos alguns *bits* extra na "*tag*" de identificação[4].

O algoritmo de marcação que, dado um apontador, visita todos os nós acessíveis no grafo é baseado num algoritmo de "*pointer-reversal*" descrito em (Knuth, 1971) que usa a própria área de "*heap*" para guardar a pilha de descida recursiva, invertendo apontadores no grafo à medida que desce na estrutura e refazendo os apontadores quando regressa. Assim o algoritmo pode visitar um grafo arbitrariamente longo, eventualmente cíclico, sem consumir pilha de execução.

Durante o algoritmo de marcação podemos facilmente eliminar as indirecções no grafo conforme é descrito em (Peyton Jones, 1987; Peyton Jones & Lestes, 1992): para tal basta

[4] Na nossa implementação usamos 2-*bits* para "marcar" respectivamente uma primeira e segunda visita ao nó. Este esquema permite implementar um algoritmo de "*pointer-reversal*" com nós de tamanho variável como os nós-`PACK` (Knuth, 1971, secção 2.3.5, pág. 421, exercício 8).

não marcar um nó-`AP` cujo lado-esquerdo seja `G_indirect` e processeguir a marcação pelo lado-direito.

Quando o algoritmo de marcação retorna, refazendo o apontador original, a indirecção é eliminada, passando a um apontador directo para o resultado.

4.3.3 Varrimento

Após a marcação de todos os nós usados, a fase de varrimento percorre sequencialmente toda a área de "*heap*" (*i.e.*, do endereço mais baixo para o mais alto) agrupando as palavras não-marcadas numa lista ligada de blocos-livres. No processo todos os nós são desmarcados, o que repõe o estado inicial para uma eventual recolha de lixo posterior.

4.3.4 Localização dos apontadores

Em qualquer situação da execução da máquina-ΦG há apenas duas localizações para apontadores para nós do grafo: a *pilha de argumentos* e os *registos da máquina abstracta.*

A pilha de argumentos é implementada como um vector global, logo a sua localização é acessível no procedimento de recolha de lixo. Os registos da máquina abstracta, contudo, são implementados como variáveis locais ao código de cada combinador, pelo que a sua localização poderá ser a pilha de execução de C ou algum registo da máquina física: em qualquer caso, a sua localização precisa é só conhecida pelo compilador de C.

Uma vez que o procedimento de recolha de lixo deve seguir *todos* os apontadores "vivos" para o grafo, é necessário seguir também apontadores mantidos em variáveis locais. Uma solução discutida em (Peyton Jones, 1993; Peyton Jones et al., 1997) e também em (M. Jones, 1994) é optar por um *colector de lixo conservativo*, que segue a pilha de execução de C e marca todas as entradas que endereçam a área de "*heap*".

A necessidade de seguir apontadores em variáveis de C leva à distinção entre dois modelos de recolha de lixo: *conservativo* e *exacto* (Peyton Jones et al., 1997):

- Um modelo *conservativo* segue qualquer valor que "pareça" um apontador para um nó (*i.e.*, que pertença ao espaço de endereços de "*heap*"). Este modelo é necessário quando apontadores e valores básicos (inteiros, por exemplo) podem partilhar armazenamento, como sucede na pilha de execução em C.

- Um modelo *exacto* segue apenas verdadeiros apontadores para os nós em uso, nunca confundindo um valor básico com um apontador. Um modelo exacto requer *mais informação* do que um modelo conservativo: necessita de distinguir completamente apontadores de valores básicos.

Instrução	Acção
`Eval(`loc`)`	redução recursiva de uma sub-expressão
`Jump(`$label$`)`	transferência incondicional
`Switch(`loc`)`	selecção de tipo estruturado
`Unwind()`	continuação da redução
`Return()`	fim da redução

Tabela 4.1: Instruções de quebra de fluxo.

Um colector de lixo conservativo não só impede a recuperação da totalidade de "*heap*" livre, mas mais importante, impede a modificação do grafo (por exemplo: eliminação indirecções ou compactação), pois correria o risco de modificar valores básicos tomados como apontadores.

Esta solução é também extremamente dependente dum conhecimento implícito da estrutura do contextos na pilha de execução de C, que pode mudar não só entre compiladores mas mesmo no mesmo compilador[5].

Por último, não resolve a situação de apontadores em registos físicos da máquina, o que limita a utilização de declarações `register` e/ou opções de compilação. É uma solução muito dependente duma arquitectura específica e responsável por erros de execução de difícil detecção.

Na máquina-ΦG optamos por um colector de lixo exacto seguindo o seguinte princípio: *todos os apontadores que devem ser seguidos pelo colector de lixo devem estar colocados na pilha de argumentos em qualquer situação em que possa ocorrer a recolha de lixo.*

Implementar este princípio acarreta o custo óbvio de colocar e retirar da pilha de argumentos alguns registos da máquina abstracta. Contudo, e como veremos, podemos limitar em tempo de compilação quer quais os registos que necessitam ser salvaguardados, quer as situações em que pode ocorrer a recolha de lixo.

Quando se efectua a recolha de lixo

Podemos limitar a ocorrência de uma recolha de lixo às transferências entre *blocos-básicos* de instruções da máquina abstracta.

Definição 19 Definimos um *bloco-básico* como uma sequência de instruções cujo fluxo é sequencial, terminando numa instrução de quebra de fluxo (ver tabela 4.1).

[5] Com diferentes opções de optimização, por exemplo.

Para garantir que dentro de um bloco-básico não ocorrem recolhas de lixo basta reservar o espaço de "*heap*" necessário no início do bloco-básico. Usamos uma nova instrução da máquina abstracta `HeapGet(`n`)` que reserva um bloco de "*heap*" contíguo de n palavras e coloca o seu endereço no (novo) registo `Heap`.

O compilador pode introduzir instruções `HeapGet(`n`)` no início de cada bloco básico, onde n é o número de células de "*heap*" usadas nesse bloco. O valor de n é facilmente sintetizado pelo compilador após a geração de código dos bloco-básicos sem as anotações `HeapGet`.

Esta solução permite não só centralizar a execução do colector de lixo na instrução `HeapGet` mas também simplifica a reserva de memória das instruções `Mkint`, `Mkap`, etc. que passam a usar directamente o registo `Heap` (*i.e.*, não necessitam de percorrer a lista de blocos livres).

Exemplo 13 O segmento de código do exemplo 12 anotado com uma instrução `HeapGet` no início do bloco-básico:

```
HeapGet(6);                  /* 6 = 1+2+1+2 */
Mkint(r0, 1);                /* int:  1 célula */
Mkap(r0, G_f, r0);           /* ap.:  2 células */
Mkint(r1, 2);                /* int:  1 célula */
Mkap(r0, r0, r1);            /* ap.:  2 células */
```

Quando salvaguardar registos

Há apenas duas instruções que podem despoletar a recolha de lixo, a saber:

`HeapGet(`n`)`: é a (única) instrução que reserva memória e pode, portanto, executar uma recolha de lixo;

`Eval(`l`)`: reduz um sub-grafo à forma-norma fraca, pelo que pode executar código arbitrário (e, em particular, efectuar recolhas de lixo).

Antes de alguma destas duas instruções devemos salvaguardar um registo "vivo" r na pilha usando um instrução `Push(`r`)`. Após a instrução `HeapGet/Eval` introduzimos um `Pop(`r`)` retira o elemento no topo da pilha e coloca-o de volta no registo[6].

[6] O valor do registo pode ser modificado pois o colector de lixo pode alterar endereços do grafo (por exemplo, eliminando indirecções).

Quais os registos a salvaguardar

Nem sempre será necessário salvaguardar todos os registos usados numa função: basta salvaguardar os registos que contêm apontadores "vivos", *i.e.*, que serão mais tarde usados no lado-direito de alguma instrução de construção do grafo.

Exemplo 14 Considerando a função `g x y = (f x) + (f y)`; o código gerado será:

```
Label(g)
      ...
      HeapGet(2);
      Mkap(r0,G_f,Stk[1]);
      Eval(r0);                  /* reduzir (f y); resultado em r0 */
      HeapGet(2);
      Mkap(r1,G_f,Stk[0]);
      Push(r0);                  /* salvaguardar r0 */
      Eval(r1);                  /* reduzir (f x) */
      Pop(r0);                   /* recuperar r0 */
      Stadd(Stk[2],r0,r1);       /* somar r0+r1 */
      ...
EndLabel
```

Antes da instrução `Eval(r0)` não é necessário salvaguardar outros registos (pois `r1` não contém um apontador válido). Por outro lado, antes de `Eval(r1)` é necessário salvaguardar `r0` pois este contém um apontador "vivo" que será usado mais tarde na instrução de soma `Stadd`.

O compilador pode sintetizar o conjunto dos registos a salvaguardar por meio de uma *análise de dependências* clássica em teoria de compiladores (Aho et al., 1986).

Análise de dependências

Para o compilador salvaguardar registos é suficiente inferir o conjunto D de localizações dependentes[7] para uma sequência de instruções. Este conjunto é sintetizado indutivamente por duas funções mutuamente recursivas:

- a função `dep` propaga as dependências duma só instrução;

[7] Para simplificar a apresentação sintetizamos o conjunto de *todas* as localizações dependentes e não apenas de registos, como seria suficiente e como é efectivamente feito na nossa implementação.

- a função Deps obtém as dependências de uma lista de instruções.

As equações 4.1 ilustram alguns casos representativos da inferência de dependências. Salientamos que o tratamento de `Switch` toma uma aproximação conservativa fazendo a união das dependências entre alternativas disjuntas.

$$
\begin{array}{ll}
\mathsf{dep}\ [\![\mathtt{Mkap(}l_0\mathtt{,}l_1\mathtt{,}l_2\mathtt{)}]\!]\ D & = (D\backslash\{l_0\}) \cup \{l_1, l_2\} \\
\mathsf{dep}\ [\![\mathtt{Mkint(}l_0\mathtt{,}n\mathtt{)}]\!]\ D & = D\backslash\{l_0\} \\
\vdots & \\
\mathsf{dep}\ [\![\mathtt{Switch(}l\mathtt{,}alts\mathtt{)}]\!]\ D & = \{l\} \cup \left(\cup_{(\sigma,c)\in alts}\,(\mathsf{Deps}\ c\ D)\right) \\
\mathsf{dep}\ [\![\ldots]\!]\ D & = D \quad \text{(instruções que não introduzem dependências)}
\end{array}
\tag{4.1}
$$

$$
\begin{array}{ll}
\mathsf{Deps}\ \mathtt{[}i_1, i_2, \ldots, i_n\mathtt{]}\ D & = \mathsf{dep}\ i_1\ (\mathsf{dep}\ i_2\ (\ldots(\mathsf{dep}\ i_n\ D)\ldots)) \\
 & = \mathtt{foldr}\ \mathsf{dep}\ D\ \mathtt{[}i_1, i_2, \ldots, i_n\mathtt{]}
\end{array}
$$

Capítulo 5

Compilação para a máquina-ΦG

Neste capítulo apresentamos o processo de compilação de um programa-fonte para código da máquina-ΦG como uma série de *esquemas de tradução* que transpõem expressões da linguagem-fonte para sequências de instruções da máquina abstracta.

5.1 Convenções de notação

5.1.1 Sequências de instruções

Sequências de instruções da máquina-ΦG são representadas como listas. Note-se que as listas de instruções podem ser imbricadas: um dos operandos da instrução de transferência condicional `Switch` é uma lista de instruções associada a cada alternativa. Esta convenção não só torna mais simples a geração de código (que não necessita de criar etiquetas locais) como se adequa directamente à geração de código C.

5.1.2 Ambientes

A informação mantida em tempo de compilação sobre associações entre identificadores e localizações (globais, pilha ou registos) é representada por um *ambiente*: uma função finita $\rho : Id \longrightarrow Loc$ entre o conjunto Id de identificadores e Loc de localizações:

$$Loc = \{\mathtt{r}n,\ n \in \mathbb{N}\} \cup \{\mathtt{Stk[}n\mathtt{]},\ n \in \mathbb{Z}\} \cup \{\mathtt{G\_}g,\ g \text{ é global}\} \cup \{\mathtt{NULL}\} \qquad (5.1)$$

(onde `NULL` é usado para representar uma localização inválida, distinta de qualquer das outras.)

Representamos um ambiente por uma lista de associações

$$\rho = [x_1 \mapsto l_1, \ldots, x_n \mapsto l_n]$$

A extensão ou re-definição ρ' de um ambiente ρ para os identificadores y_i é representada por $\rho' = \rho[y_1 \mapsto q_1, \ldots, y_n \mapsto q_n]$.

5.1.3 Registos

Os esquemas de tradução necessitam de reservar registos temporários. A informação sobre os registos disponíveis deve ser simultaneamente passada como argumento herdado e sintetizada como parâmetro de saída. Para simplificar a exposição omitimos essa informação, recorrendo apenas a uma anotação como "r é um novo registo".

Claro está, na implementação destes esquemas incluímos a informação extra que permite re-utilizar registos temporários.

5.2 Esquemas de compilação

Os esquemas seguintes fazem a compilação de expressões da linguagem-fonte em sequências de código da máquina abstracta, baseados na apresentação feita em (Peyton Jones & Lestes, 1992).

Note-se que o código básico gerado não é ainda directamente transponível para linguagem C, necessitando ainda da introdução de algumas instruções de anotação referentes à gestão de memória (ver secção 4.3) e uso de registos. Além disso, o código pode ainda passar por algumas optimizações.

Para tornar mais clara a exposição, apresentamos aqui apenas a geração do código básico, remetendo para a secção 5.4 a discussão de optimizações seguintes.

5.2.1 Esquema $\mathcal{C}$

O esquema $\mathcal{C}$ é usado para gerar código não-estrito, isto é, código que constrói o grafo para uma expressão que será eventualmente reduzida mais tarde.

Além da expressão a compilar delimitada por parêntesis rectos duplos $[\![\ldots]\!]$, o esquema $\mathcal{C}$ recebe ainda o ambiente ρ que associa identificadores a localizações. O resultado da compilação é um par ordenado (l, c) em que c é o código da máquina abstracta (uma lista de instruções) e l é a localização final da raiz do grafo.

Expressões básicas

As expressões básicas são inteiros, identificadores (locais ou globais) e a aplicação. Note-se que usamos o ambiente ρ para obter localização de identificadores locais e globais.

$$
\begin{array}{lll}
\mathcal{C}[\![n]\!]\,\rho & = (r, \texttt{[Mkint(}r\texttt{,}n\texttt{)]}) & (\text{se } n \text{ é inteiro}) \\
 & \quad \text{onde } r \text{ é um novo registo} & \\
\mathcal{C}[\![x]\!]\,\rho & = (\rho\, x, \texttt{[]}) & (\text{se } x \text{ é um identificador}) \\
\mathcal{C}[\![g]\!]\,\rho & = (\rho\, g, \texttt{[]}) & (\text{se } g \text{ é um global}) \\
\mathcal{C}[\![e_1\; e_2]\!]\,\rho & = (r, c_1 +\!\!+ c_2 +\!\!+ \texttt{[Mkap(}r\texttt{,}l_1\texttt{,}l_2\texttt{)]}) & \\
 & \quad \begin{array}{lll} \text{onde} & r & \text{é um novo registo} \\ & (l_1, c_1) & = \mathcal{C}[\![e_1]\!]\,\rho \\ & (l_2, c_2) & = \mathcal{C}[\![e_2]\!]\,\rho \end{array} &
\end{array}
\tag{5.2}
$$

Definições locais

Na compilação de uma expressão-`let` é necessário associar os identificadores ligados às localizações das sub-expressões. O código resultado é apenas a concatenação do código de cada uma das sub-expressões:

$$
\begin{array}{l}
\mathcal{C}\left[\!\!\left[\begin{array}{ll} \texttt{let} & x_1 = e_1 \\ & \quad\vdots \\ & x_n = e_n \\ & \texttt{in}\ E \end{array}\right]\!\!\right]\rho \;= (l, c_1 +\!\!+ \cdots +\!\!+ c_n +\!\!+ c) \\
\qquad\qquad \begin{array}{ll} \text{onde} & \\ (l_i, c_i) & = \mathcal{C}[\![e_i]\!]\,\rho, \quad i = 1, \ldots, n \\ \rho' & = \rho[x_1 \mapsto l_1, \ldots, x_n \mapsto l_n] \\ (l, c) & = \mathcal{C}[\![E]\!]\,\rho' \end{array}
\end{array}
\tag{5.3}
$$

No caso de uma definição recursiva é necessário reservar nós-`AP` "fictícios" para associar aos identificadores durante a compilação das sub-expressões. No final do código de cada sub-expressão acrescentamos uma instrução `Update` que escreve sobre o nó fictício a indirecção para o resultado real; todas as referências recursivas nas sub-expressões passam a referências cíclicas no grafo.

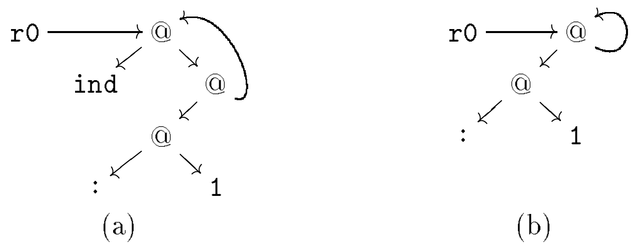

Figura 5.1: Grafos construídos por "`letrec xs=1:xs in xs`"

$$
\mathcal{C}\left[\!\!\left[\begin{array}{ll} \texttt{letrec} & x_1 = e_1 \\ & \vdots \\ & x_n = e_n \\ & \texttt{in}\ E \end{array}\right]\!\!\right] \rho \;=\; \begin{array}{l} (l,\ \ [\texttt{Mkap(}r_i\texttt{,NULL,NULL)} \mid i = 1,\ldots,n] +\!\!+ \\ \qquad +\!\!+ c'_1 +\!\!+ \cdots +\!\!+ c'_n +\!\!+ c) \\[1em] \text{onde} \\ \begin{array}{ll} r_1,\ldots,r_n & \text{são novos registos} \\ \rho' & = \rho[x_1 \mapsto r_1,\ldots,x_n \mapsto r_n] \\ (l,c) & = \mathcal{C}[\![E]\!]\ \rho' \\ (l_i,c_i) & = \mathcal{C}[\![e_i]\!]\ \rho', \quad i = 1,\ldots,n \\ c'_i & = c_i +\!\!+ [\texttt{Update(}r_i\texttt{,}l_i\texttt{)}], \quad i = 1,\ldots,n \end{array} \end{array} \tag{5.4}
$$

Exemplo 15 Consideremos a definição recursiva "`letrec xs=1:xs in xs`", que representa uma lista infinita de 1s; o código gerado para `xs` será:

```
Mkap(r0,NULL,NULL);
Mkint(r1,1);
Mkap(r1,G_Cons,r1);
Mkap(r1,r1,r0);
Update(r0,r1);
```

Esta sequência constrói o grafo representado na figura 5.1 (a), onde "`ind`" representa o combinador-indirecção exposto na secção 4.2.3 e "`:`" é o construtor de listas.

Na secção 5.4.1 introduziremos uma optimização local que substitui as duas últimas instruções por uma só, evitando a indirecção e construído directamente o grafo da figura 5.1 (b).

5.2.2 Esquema $\mathcal{E}$

O esquema $\mathcal{E}$ é usado para gerar código estrito para uma expressão, *i.e.*, código que reduz uma expressão à forma-normal fraca. A semântica do código compilador por $\mathcal{E}$ é equivalente ao código de $\mathcal{C}$ seguido de um `Eval`. Contudo, em muitas situações podemos compilar directamente código que reduz o resultado sem necessitar do `Eval` final.

Sempre que possível o esquema $\mathcal{E}$ propaga a geração de código estrito usando $\mathcal{E}$ recursivamente para compilar sub-expressões; quando tal não é possível recorremos ao esquema geral $\mathcal{C}$ seguido de uma instrução `Eval`.

Os parâmetros passados a $\mathcal{E}$ e o resultado são idênticos aos usados em $\mathcal{C}$.

Expressões básicas

Apenas necessitamos de distinguir o caso dos inteiros e de constantes globais: nestes casos podemos obter o resultado directamente, sem necessitar de um `Eval` posterior.

$$\begin{aligned} \mathcal{E}[\![n]\!]\,\rho \quad &= (r, \texttt{[Mkint(}r\texttt{,}n\texttt{)]}) && \text{(se } n \text{ é inteiro)} \\ &\quad \text{onde } r \text{ é um novo registo} \\ \mathcal{E}[\![g]\!]\,\rho \quad &= (\rho\, g, \texttt{[]}) && \text{(se } g \text{ é primitiva global)} \end{aligned} \tag{5.5}$$

Não incluímos aqui nem identificadores locais nem os identificadores globais definidos pelo utilizador: estes serão tratados no caso final, pois em geral necessitam de uma instrução `Eval` para os reduzir à forma-normal fraca.

Operações estritas

Na compilação de operadores primitivos estritos (como os operadores aritméticos) podemos propagar o uso de $\mathcal{E}$ para os argumentos; ilustramos com o exemplo do operador `+`:

$$\begin{aligned} \mathcal{E}[\![+\ e_1\ e_2]\!]\,\rho \quad &= (r, c_1 \mathbin{+\!\!+} c_2 \mathbin{+\!\!+} \texttt{[Add(}r\texttt{,}l_1\texttt{,}l_2\texttt{)]}) \\ \text{onde} \quad r \quad &\text{é um novo registo} \\ (l_1, c_1) \quad &= \mathcal{E}[\![e_1]\!]\,\rho \\ (l_2, c_2) \quad &= \mathcal{E}[\![e_2]\!]\,\rho \end{aligned} \tag{5.6}$$

A instrução da máquina abstracta `Add` soma dois valores inteiros; o resultado é colocado num novo nó-`INTG` (cf. apêndice A).

Omitimos as definições análogas a esta para todos os operadores aritméticos e de comparação.

Operações não-estritas

Por outro lado, ao compilar construtores de tipos estruturados como o construtor de listas "`:`" não podemos garantir que as sub-componentes sejam reduzidas, pelo que recorremos ao esquema $\mathcal{C}$:

$$\begin{array}{rl} \mathcal{E}[\![: \ e_1 \ e_2]\!] \ \rho & = (r, c_1 +\!\!+ c_2 +\!\!+ \texttt{[Pack}_2\texttt{(CTAG\_Cons,}r\texttt{,}l_1\texttt{,}l_2\texttt{)]}) \\ & \quad \text{onde} \quad \begin{array}{ll} r & \text{é um novo registo} \\ (l_1, c_1) & = \mathcal{C}[\![e_1]\!] \ \rho \\ (l_2, c_2) & = \mathcal{C}[\![e_2]\!] \ \rho \end{array} \end{array} \tag{5.7}$$

Note-se ainda que construtores sem argumentos (como `True`, `False`, `[]`) não necessitam de um tratamento especializado: são constantes globais com nós-`PACK` reservados estaticamente. A geração de código fica tratada nas equações 5.5.

Definições locais

As definições locais `let` e `letrec` são tratadas no esquema $\mathcal{E}$ de forma análoga ao esquema $\mathcal{C}$. Contudo, note-se que podemos propagar o uso de $\mathcal{E}$ para a expressão final E (embora não para as sub-expressões e_i):

$$\begin{array}{l} \mathcal{E}\left[\!\!\left[\begin{array}{ll} \texttt{let} & x_1 = e_1 \\ & \vdots \\ & x_n = e_n \\ & \texttt{in}\ E \end{array} \right]\!\!\right] \rho \ = (l, c_1 +\!\!+ \cdots +\!\!+ c_n +\!\!+ c) \\ \qquad \text{onde} \\ \qquad \begin{array}{ll} (l_i, c_i) & = \mathcal{C}[\![e_i]\!] \ \rho, \quad i = 1, \ldots, n \\ \rho' & = \rho[x_1 \mapsto l_1, \ldots, x_n \mapsto l_n] \\ (l, c) & = \mathcal{E}[\![E]\!] \ \rho' \end{array} \end{array} \tag{5.8}$$

$$\begin{array}{l} \mathcal{E}\left[\!\!\left[\begin{array}{ll} \texttt{letrec} & x_1 = e_1 \\ & \vdots \\ & x_n = e_n \\ & \texttt{in}\ E \end{array} \right]\!\!\right] \rho \ = \begin{array}{ll} (l, & \texttt{[Mkap(}r_i\texttt{,NULL,NULL)} \mid i = 1, \ldots, n\texttt{]} +\!\!+ \\ & +\!\!+ c'_1 +\!\!+ \cdots +\!\!+ c'_n +\!\!+ c) \end{array} \\ \qquad \text{onde } r_1, \ldots, r_n \text{ são novos registos} \\ \qquad \begin{array}{ll} \rho' & = \rho[x_1 \mapsto r_1, \ldots, x_n \mapsto r_n] \\ (l, c) & = \mathcal{E}[\![E]\!] \ \rho' \\ (l_i, c_i) & = \mathcal{C}[\![e_i]\!] \ \rho', \quad i = 1, \ldots, n \\ c'_i & = c_i +\!\!+ \texttt{[Update(}r_i\texttt{,}l_i\texttt{)]}, \quad i = 1 \ldots, n \end{array} \end{array} \tag{5.9}$$

Expressões-`case`

Para compilar uma expressão de selecção sobre um objecto estruturado é necessário gerar código para reduzir o objecto estruturado e para cada alternativa.

Expressão-`case`:	**Código em C:**
	`switch(*`l`) {`
`case` E `of`	
σ_1 `...` `->` e_1	`case CTAG_`σ_1`:`
	`/* código para` e_1 `*/`
	`break;`
	`case CTAG_`σ_2`:`
$\mid$ σ_2 `...` `->` e_2	`/* código para` e_2 `*/`
	`break;`
$\vdots$	$\vdots$
	`case CTAG_`σ_n`:`
$\mid$ σ_n `...` `->` e_n	`/* código para` e_n `*/`
	`break;`
	`}`

Figura 5.2: Geração de código para uma expressão-`case`

A instrução de transferência condicional da máquina-ΦG é `Switch(`l`,`$alts$`)` que testa o construtor no nó-`PACK` em l (que deve, portanto, ter sido já reduzido) e transfere a execução para a alternativa correspondente em $alts$.

Ao nível da máquina abstracta, o código das alternativas é imbricado dentro do `Switch` e não marcado com etiquetas locais; isto permite-nos transpor para linguagem C como uma construção-`switch` (ver figura 5.2).

Definimos então o esquema $\mathcal{E}$ para uma expressão-`case`:

$$
\begin{array}{llll}
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ alt_1|\ldots|alt_n]\!]\ \rho & = (r, c +\!\!+ \texttt{[Switch(}l\texttt{,[}(\sigma_1, c_1), \ldots, (\sigma_n, c_n)\texttt{])]}) & & \\
& \quad \text{onde}\ \ r & \text{é um novo registo} & \\
& \quad\quad\quad\ \ (l, c) & = \mathcal{E}[\![e]\!]\ \rho & \\
& \quad\quad\quad\ \ (\sigma_i, c_i) & = \mathcal{E}_{alt}[\![alt_i]\!]\ \rho\ l\ r &
\end{array}
\tag{5.10}
$$

O esquema auxiliar $\mathcal{E}_{alt}$ é usado para gerar o código de cada alternativa que vem etiquetada com o construtor associado. Note-se que $\mathcal{E}_{alt}$ recebe como parâmetros não só o ambiente ρ mas também a localização da raiz do objecto estruturado l e do registo r onde deve ficar o resultado. Ao gerar para uma alternativa que liga n variáveis usamos a instrução $\texttt{Split}_n$ para decompor as n componentes do tipo estruturado (cf. apêndice A).

$$
\begin{array}{lll}
\mathcal{E}_{alt}[\![\sigma \to e]\!]\ \rho\ l\ r & = (\sigma,\ c_2 +\!\!+ \texttt{[Set(}r\texttt{,}l_2\texttt{)]}) & \\
& \quad \text{onde} \quad (l_2, c_2) \quad = \mathcal{E}[\![e]\!]\ \rho & \\
\mathcal{E}_{alt}[\![\sigma\ x_1 \ldots x_n \to e]\!]\ \rho\ l\ r & = (\sigma,\ \texttt{[Split}_n\texttt{(}l\texttt{,}r_1,\ldots,r_n\texttt{)]} +\!\!+ c_2 +\!\!+ \texttt{[Set(}r\texttt{,}l_2\texttt{)]}) & \\
& \quad \text{onde} \quad r_1,\ldots,r_n \quad \text{são novos registos} & \\
& \qquad\qquad \rho' \qquad\quad = \rho[x_1 \mapsto r_1, \ldots, x_n \mapsto r_n] & \\
& \qquad\qquad (l_2, c_2) \quad = \mathcal{E}[\![e]\!]\ \rho' & (5.11) \\
\mathcal{E}_{alt}[\![\texttt{var}\ v \to e]\!]\ \rho\ l\ r & = (\texttt{default},\ c_2 +\!\!+ \texttt{[Set(}r\texttt{,}l_2\texttt{)]}) & \\
& \quad \text{onde} \quad \rho' \qquad = \rho[v \mapsto l] & \\
& \qquad\qquad (l_2, c_2) \quad = \mathcal{E}[\![e]\!]\ \rho' & \\
\mathcal{E}_{alt}[\![\texttt{default} \to e]\!]\ \rho\ l\ r & = (\texttt{default},\ c_2 +\!\!+ \texttt{[Set(}r\texttt{,}l_2\texttt{)]}) & \\
& \quad \text{onde} \quad (l_2, c_2) \quad = \mathcal{E}[\![e]\!]\ \rho &
\end{array}
$$

Caso terminal

Caso a expressão a compilar não seja uma das anteriores, o esquema $\mathcal{E}$ recorre ao esquema $\mathcal{C}$ acrescentando uma instrução `Eval` ao código gerado:

$$
\begin{array}{ll}
\mathcal{E}[\![e]\!]\ \rho & = (l, c +\!\!+ \texttt{[Eval(}l\texttt{)]}) \quad \text{(se } e \text{ não é nenhum dos anteriores)} \\
& \text{onde } (l, c) = \mathcal{C}[\![e]\!]\ \rho
\end{array} \qquad (5.12)
$$

5.2.3 Esquema $\mathcal{R}$

O esquema $\mathcal{R}$ gera o código de redução para um super-combinador definido no programa-fonte.

Cada equação de definição "$f\ x_1 \ldots x_n\ =\ E$" dum super-combinador f no programa-fonte origina uma função em C a que corresponde uma sequência de instruções da máquina abstracta `[Label(`f`),`$\ldots$`,EndLabel]`. Note-se ainda que o esquema $\mathcal{R}$ recebe como argumento um ambiente inicial ρ_0 que associa identificadores globais às suas localizações.

$$
\begin{array}{ll}
\mathcal{R}[\![f\ x_1 \ldots x_n\ =\ E]\!]\ \rho_0 & = \texttt{[Label(}f\texttt{),ArgsCheck(}n\texttt{),Mkargs(}n\texttt{)]} +\!\!+ c +\!\!+ \\
& \quad +\!\!+ \texttt{[Update(Stk[}n\texttt{],}l\texttt{),Unwind(),EndLabel]} \\
& \text{onde} \quad \rho \qquad = \rho_0[x_1 \mapsto \texttt{Stk[0]}, \ldots, x_n \mapsto \texttt{Stk[}n-1\texttt{]]} \\
& \qquad\quad (l, c) \quad = \mathcal{E}[\![E]\!]\ \rho
\end{array} \qquad (5.13)
$$

5.3 Compilação de combinadores $\mathcal{C}\ell(\mathcal{K})$

O abstracção combinatória substitui abstracções-λ no programa-fonte original por combinadores $\mathcal{C}\ell(\mathcal{K})$. Como foi mencionado, a "eliminação de λs" um programa-fonte concreto necessita de um número finito de combinadores Φ_α (mais precisamente: quando muito um por cada abstracção). O compilador necessita então de gerar sequências de código para a redução de cada um destes combinadores Φ_α.

Observamos que os combinadores primitivos K e $\Phi_\epsilon(\equiv \mathsf{I})$ são os únicos cujo *contractum* é uma "projecção" de um dos seus argumentos; o código para estes dois combinadores é pré-definido em C na biblioteca de rotinas de execução da linguagem. Todos os outros Φ_α são definidos por aplicações (cf. equações 2.9, pág. 26).

O código de construção do *contractum* dos combinadores Φ_α é então sintetizado pelo esquema $\mathcal{K}$ a partir do índice α seguindo a definição nas equações 2.9.

O esquema $\mathcal{K}$ tem como argumentos o índice α do combinador e uma lista $\texttt{[}x_1, \ldots, x_n\texttt{]}$ de índices dos argumentos na pilha.

$$
\begin{array}{ll}
\mathcal{K}\,[\![\epsilon]\!]\,\texttt{[}x\texttt{]} & = (\texttt{Stk[}x\texttt{]},\texttt{[]}) \\
\mathcal{K}\,[\![c \cdot \alpha]\!]\,\texttt{[}x_1,\ldots,x_n\texttt{]} & = (r, c_1 \mathbin{+\!\!+} \texttt{[Mkap(}r\texttt{,}l_1\texttt{,Stk[}x_1\texttt{])]}) \\
 & \quad \begin{array}{lll} \text{onde} & r & \text{é um novo registo} \\ & (l_1,c_1) & = \mathcal{K}\,[\![\alpha]\!]\,\texttt{[}x_2,\ldots,x_n\texttt{]} \end{array} \\
\mathcal{K}\,[\![b \cdot \alpha]\!]\,\texttt{[}x_1,\ldots,x_n\texttt{]} & = (r, c_1 \mathbin{+\!\!+} \texttt{[Mkap(}r\texttt{,Stk[}x_1\texttt{],}l_1\texttt{)]}) \\
 & \quad \begin{array}{lll} \text{onde} & r & \text{é um novo registo} \\ & (l_1,c_1) & = \mathcal{K}\,[\![\alpha]\!]\,\texttt{[}x_2,\ldots,x_n\texttt{]} \end{array} \\
\mathcal{K}\,[\![(\alpha_1,\alpha_2)]\!]\,\texttt{[}x_1,\ldots,x_n\texttt{]} & = (r, c_1 \mathbin{+\!\!+} c_2 \mathbin{+\!\!+} \texttt{[Mkap(}r\texttt{,}l_1\texttt{,}l_2\texttt{)]}) \\
 & \quad \begin{array}{lll} \text{onde} & r & \text{é um novo registo} \\ & k & = \#\alpha_1 \\ & (l_1,c_1) & = \mathcal{K}\,[\![\alpha_1]\!]\,\texttt{[}x_1,\ldots,x_k,x_n\texttt{]} \\ & (l_2,c_2) & = \mathcal{K}\,[\![\alpha_2]\!]\,\texttt{[}x_{k+1},\ldots,x_n\texttt{]} \end{array}
\end{array}
\tag{5.14}
$$

Finalmente, definimos o esquema $\mathcal{R}_\Phi$ que gera o código de redução de um combinador Φ_α:

$$
\begin{array}{ll}
\mathcal{R}_\Phi[\![\Phi_\alpha]\!] & = \texttt{[Label(}\Phi_\alpha\texttt{),ArgsCheck(}n\texttt{),Mkargs(}n\texttt{)]} \mathbin{+\!\!+} c \mathbin{+\!\!+} \\
 & \quad \mathbin{+\!\!+} \texttt{[Update(Stk[}n\texttt{],}l\texttt{),Unwind(),EndLabel]} \\
 & \quad \begin{array}{lll} \text{onde} & n & = \#\alpha + 1 \\ & (l,c) & = \mathcal{K}\,[\![\alpha]\!]\,\texttt{[}0,1,\ldots,n-1\texttt{]} \end{array}
\end{array}
\tag{5.15}
$$

5.4 Optimizações

Uma das possibilidades abertas pela compilação para código de máquina abstracta é a de efectuar optimizações sobre este código intermédio. Estas optimizações permitem tornar a execução do código gerado mais eficiente (em termos de tempo, espaço ou ambos) e são independentes do gerador de código máquina concreto (o que as torna completamente portáteis). Além disso, há uma convicção fundamentada na comunidade de compiladores de que o potencial para melhoria da execução de programas com optimizações a este nível é grande.

No caso de linguagens simbólicas (*i.e.*, funcionais e lógicas) o potencial para optimização em tempo de compilação parece ainda maior: a "distância semântica" entre o "*hardware*" genérico e o modelo de execução destas linguagens (que foi em tempos um argumento justificativo para o desenvolvimento de "*hardware*" com arquitecturas especiais) parece agora motivar o desenvolvimento de analises de programa em tempo de compilação de forma a potenciar optimizações globais (Peyton Jones, 1996; Kogge, 1991).

Desde da introdução da "*G-machine*" que foram desenvolvidas várias optimizações ao nível do código intermédio (Peyton Jones, 1987). Nesta secção apresentamos algumas dessas optimizações transpostas para a nossa variante, a máquina-ΦG. Não se trata, contudo, de uma descrição exaustiva, mas sim da exemplificação de como algumas optimizações simples se integram de forma transparente com o nosso modelo de geração e execução de código.

5.4.1 Optimizações locais

As *optimizações locais* consistem na substituição de sequências de instruções da máquina abstracta por uma instrução ou sequência equivalente mais eficiente.

Apresentamos as optimizações locais por meio de um esquema de re-escrita Opt que opera entre listas de instruções. Definimos apenas as transformações "locais" de Opt, subentendendo-se que estas devem ser propagadas recursivamente para as sub-listas de instruções.

Eliminação de indirecções

Podemos evitar introduzir uma indirecção para um nó recém-criado simplesmente construindo o nó directamente sobre o resultado (Peyton Jones, 1987). Para tal basta substituir a sequência "construção/`Update`" por uma instrução especializada (cf. apêndice A).

Note-se contudo que não podemos prescindir completamente de indirecções para as situações em que o objecto referido não é um nó mas sim um apontador.

$$
\begin{aligned}
\mathsf{Opt}\ [\texttt{Mkap}(l_1,l_2,l_3),\texttt{Update}(l_4,l_1)] &= [\texttt{Stap}(l_4,l_2,l_3)] \\
\mathsf{Opt}\ [\texttt{Mkint}(l_1,n),\texttt{Update}(l_2,l_1)] &= [\texttt{Stint}(l_2,n)] \\
\mathsf{Opt}\ [\texttt{Add}(l_1,l_2,l_3),\texttt{Update}(l_4,l_1)] &= [\texttt{Stadd}(l_4,l_2,l_3)] \\
\mathsf{Opt}\ [\texttt{Mul}(l_1,l_2,l_3),\texttt{Update}(l_4,l_1)] &= [\texttt{Stmul}(l_4,l_2,l_3)] \\
\mathsf{Opt}\ [\texttt{Sub}(l_1,l_2,l_3),\texttt{Update}(l_4,l_1)] &= [\texttt{Stsub}(l_4,l_2,l_3)] \\
\mathsf{Opt}\ [\texttt{Div}(l_1,l_2,l_3),\texttt{Update}(l_4,l_1)] &= [\texttt{Stdiv}(l_4,l_2,l_3)] \\
\mathsf{Opt}\ [\texttt{Mod}(l_1,l_2,l_3),\texttt{Update}(l_4,l_1)] &= [\texttt{Stmod}(l_4,l_2,l_3)]
\end{aligned}
\tag{5.16}
$$

Eliminação de instruções redundantes

Em alguns casos as instruções de máquina abstracta geradas são redundantes, *i.e.*, são operações nulas. Em vez de verificar esses casos ao nível da geração de código preferimos fazer essas simplificações como um conjunto de optimizações locais:

$$
\begin{aligned}
\mathsf{Opt}\ [\texttt{ArgsCheck(0)}] &= [] \\
\mathsf{Opt}\ [\texttt{Mkargs(0)}] &= [] \\
\mathsf{Opt}\ [\texttt{HeapGet(0)}] &= [] \\
\mathsf{Opt}\ [\texttt{Set}(l,l)] &= [] \\
\mathsf{Opt}\ [\texttt{Drop(0)}] &= [] \\
\mathsf{Opt}\ [\texttt{Trim}(n,\texttt{0})] &= []
\end{aligned}
\tag{5.17}
$$

5.4.2 Eliminação de reduções redundantes

Na redução de grafos duma linguagem funcional pura, um grafo deve ser reduzido quando muito uma vez, pois o resultado dessa redução é imutável. Isto implica que o uso repetido de `Eval` de uma mesma localização é redundante e deve ser evitado, uma vez que levaria à duplicação de trabalho (cf. secção 3.4.2).

O compilador pode detectar `Eval`s redundantes fazendo uma *análise de reduções* sobre o código gerado, *i.e.*, sintetizando o conjunto E com as localizações que foram garantidamente já reduzidas; uma instrução `Eval`(l) onde $l \in E$ é então redundante e pode ser eliminada.

Descrevemos sucintamente o esquema eval que sintetiza as reduções de uma instrução e Evals que sintetiza as reduções de uma lista de instruções:

- o tratamento de instruções "`Push/Pop`" desloca os índices de localizações da forma `Stk`[n], deixando inalteradas as outras localizações em E;
- o tratamento de `Switch` toma uma aproximação conservativa: a intersecção dos conjuntos sintetizados por cada umas das alternativas.

Nas equações 5.18 apresentamos apenas os casos mais representativos; os esquemas podem ser facilmente estendidos para os restantes.

$$
\begin{aligned}
\textsf{eval}\ E\ [\![\texttt{Mkint(}l_0\texttt{,}n\texttt{)}]\!] &= E \cup \{l_0\} \\
\textsf{eval}\ E\ [\![\texttt{Mkap(}l_0\texttt{,}l_1\texttt{,}l_2\texttt{)}]\!] &= E \backslash \{l_0\} \\
&\ \vdots \\
\textsf{eval}\ E\ [\![\texttt{Eval(}l\texttt{)}]\!] &= E \cup \{l\} \\
\textsf{eval}\ E\ [\![\texttt{Push(}l\texttt{)}]\!] &= E^{+1} \\
\textsf{eval}\ E\ [\![\texttt{Pop(}l\texttt{)}]\!] &= E^{-1} \\
\textsf{eval}\ E\ [\![\texttt{Switch(}l\texttt{,}alts\texttt{)}]\!] &= \cap_{(\sigma,c)\in alts}\ (\textsf{Evals}\ c\ E) \\
\\
\textsf{Evals}\ \texttt{[}i_1, i_2, \ldots, i_n\texttt{]}\ E &= \textsf{eval}\ (\ldots(\textsf{eval}\ (\textsf{eval}\ E\ i_1)\ i_2)\ldots)\ i_n \\
&= \texttt{foldl}\ \textsf{eval}\ E\ \texttt{[}i_1, i_2, \ldots, i_n\texttt{]}
\end{aligned}
\tag{5.18}
$$

onde

$$E^{\pm k} = \{\texttt{Stk[}n \pm k\texttt{]} \mid \texttt{Stk[}n\texttt{]} \in E\} \ \cup\ \{l \mid l \in E \wedge l \neq \texttt{Stk[}n\texttt{]}\}$$

5.4.3 Optimização de última chamada

Quando a definição de um super-combinador termina com uma a aplicação saturada a um outro combinador (*i.e.*, aplicação com o número correcto de argumentos) dizemos que o super-combinador termina com uma *última chamada*. No caso em que a função invocada é a mesma que está a ser definida, a última chamada corresponde à *recursão na cauda*.

Exemplo 16 A função `fact` ilustra uma definição de factorial com recursão na cauda. O 2º argumento de `fact` é usado para "acumular" os produtos intermédios, de forma que $n! = $ (`fact` n 1):

```
fact n f = if n>0 then fact (n-1) (n*f) else 1 ;
```

Podemos gerar código mais eficiente para a última chamada: uma vez que o próximo *redex* será `fact (n-1) (n*f)` e sabemos que `fact` usa dois argumentos, não é necessário construir um grafo para imediatamente o decompor com um "*unwind*". Em vez disso, podemos colocar os novos argumentos na pilha e saltar directamente para o código de redução de `fact`, efectivamente transformando a recursão em iteração.

A optimização de última chamada é importante em linguagens funcionais estritas (Kogge, 1991) e não-estritas (Peyton Jones, 1987) e linguagens lógicas (Aït-Kaci, 1991). Nestas linguagens a recursão é a única forma de exprimir iteração (como no exemplo `fact`) e a optimização de última chamada permite executar essa recursão sem consumir espaço de pilha.

Numa implementação baseada na "*G-machine*", a optimização de última chamada é importante não só pela recuperação de espaço da pilha, mas também para reduzir a utilização de "*heap*" associada ao grafo da última chamada. Além disso, a execução é transferida directamente de volta para o corpo da função, sem o peso adicional de efectuar um "*unwind*" e de verificar os argumentos.

A nossa apresentação para a máquina-ΦG é baseada na descrição dada em (Peyton Jones, 1987) com uma simplificação: consideramos apenas o caso em que a última chamada é *saturada*, *i.e.*, com o número correcto de argumentos. Nesta situação a última chamada pode ser efectivamente substituída por um salto incondicional sem testes adicionais durante a execução (Peyton Jones, 1987, cap. 21). Além disso, a principal vantagem desta optimização parece-nos estar na geração de código iterativo para certas funções recursivas, o que corresponde sempre a últimas chamadas saturadas.

Alterações à geração de código

Para possibilitar a optimização de última chamada necessitamos de separar o código de um super-combinador em duas partes: a *inicialização* (que verifica e re-estrutura argumentos na pilha) e a *execução* (que implementa a redução de grafos propriamente dita). É necessário poder executar o código a partir destes dois pontos de entrada, o que implica que cada uma destas sequências seja traduzida numa função de C separada (pois não podemos saltar para o meio de uma função).

O código gerado para um combinador f com n argumentos passará então a ter a estrutura geral:

```
Label(f_init)
    ArgsCheck(n);
    Mkargs(n);
    Jump(f_exec);
EndLabel
Label(f_exec)
    ⋮
EndLabel
```

Note-se que esta decomposição é desnecessária se f não tem argumentos (*i.e.*, se $n = 0$) ou se f não é usado numa última chamada por outro combinador.

Suponhamos então que um outro combinador g de m argumentos termina com uma última chamada $(f\ A_1 \ldots A_n)$; então o código para g deve:

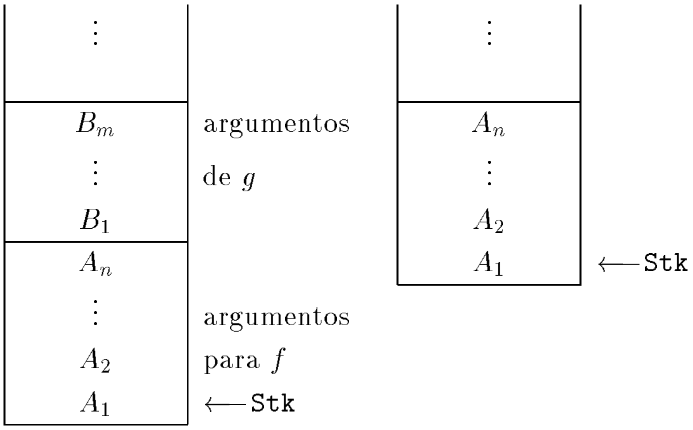

Figura 5.3: Execução de `Trim(`n`,`m`)`

1. Construir os novos n argumentos para f;
2. Eliminar os seus m argumentos;
3. Saltar para f`_exec`.

Utilizamos uma nova instrução `Trim(`n`,`m`)` para eliminar os argumentos de g, deslocando os n apontadores no topo da pilha m posições para baixo (ver figura 5.3 e ainda o apêndice A). O código para um combinador que termina numa última chamada terá então a forma geral:

```
Label(g)
     ArgCheck(m);   /* g tem m argumentos */
     Mkargs(m);
     ⋮
     Push(&An);     /* n-ésimo argumento de f */
     ⋮
     Push(&A2);     /* 2º argumento */
     Push(&A1);     /* 1º argumento */
     Trim(n,m);     /* elimina argumentos de g */
     Jump(f_exec);  /* executa redução de f */
EndLabel
```

Note-se que cada "`Push(&`A_i`)`" corresponderá efectivamente a uma sequência de instruções que constrói o grafo do argumento A_i e que efectua o `Push` dum apontador.

O exemplo seguinte ilustra o resultado da optimização de última chamada de uma função definida com recursão na cauda.

Exemplo 17 O código com optimização de última chamada para a definição de `fact` do exemplo 16 poderá ser:

```
Label(fact_init)                    /* código de inicialização */
     ArgsCheck(2);
     Mkargs(2);
     Jump(fact_exec);
EndLabel

Label(fact_exec)                    /* código de redução */
     Reg(r0);
     Reg(r1);
     Eval(Stk[0]);
     HeapGet(1);
     Mkint(r0,0);
     Lt(r0,r0,Stk[0]);
     Switch(r0) {                   /* n > 0?  */
     case CTAG_True:
          HeapGet(9);
          Mkap(r0,G_mul,Stk[0]);    /* 2º argumento */
          Mkap(r0,r0,Stk[1]);
          Push(r0);                 /* coloca (n * f) na pilha */
          Mkint(r0,1);              /* 1º argumento */
          Mkap(r1,G_sub,Stk[1]);
          Mkap(r0,r1,r0);
          Push(r0);                 /* coloca (n - 1) na pilha */
          Trim(2,2);
          Jump(fact_exec);          /* iteração na cauda */
     break;
     case CTAG_False:
          Eval(Stk[1]);             /* caso terminal */
          Update(Stk[2],Stk[1]);
          Drop(2);
          Unwind();
     break;
     }
EndLabel
```

O exemplo acima ilustra que a optimização de última chamada deve ser propagada para as alternativas de expressões condicionais. Isto implica efectuar algumas modificações sobre os nossos esquemas de compilação; descrevemo-las aqui de forma abreviada, uma vez tratarem-se de modificações extensas mas simples e semelhantes ao que é tratado na literatura, em particular (Peyton Jones, 1987, cap. 21):

- usamos um novo esquema $\mathcal{RS}$ baseado no esquema $\mathcal{E}$ para gerar código do corpo de um supercombinador; $\mathcal{RS}$ deve detectar uma última chamada e gerar o código adequado;
- $\mathcal{RS}$ tem um argumento extra, o número de argumentos m do supercombinado original, de forma a poder gerar a instrução `Trim(`n`,`m`)`;
- devemos propagar $\mathcal{RS}$ para alternativas de expressões-`case`.

Notamos ainda que numa linguagem não-estrita a optimização de última chamada por si só não permite a execução de uma função como factorial (exemplo 17) em espaço constante: os argumentos são ainda construídos como grafos, apesar de serem efectivamente usados para a redução do factorial. Para obviar isto seria possível empregar técnicas de "*strictness analysis*" (Peyton Jones, 1987, cap. 22) para inferir em tempo de compilação quais os argumentos que seriam sempre reduzidos.

Capítulo 6

Conclusão

6.1 Reflexões sobre o trabalho efectuado

Nesta dissertação apresentamos a implementação de uma linguagem puramente funcional reduzida com as características nucleares das linguagens funcionais modernas (incluindo semântica não-estrita e tipos estruturados).

A nossa implementação toma partido de uma tradução que conjuga a simplicidade da redução em combinadores com a complexidade em número de reduções do programa original:

- Introduzimos, no pior caso, um combinador por cada abstracção-λ no programa original; consequentemente efectuamos uma redução em combinadores por cada β-*redex* no programa original (cf. secção 3.4.1).
- Obtemos combinadores Φ de *granularidade alta*, *i.e.*, cuja redução corresponde a sequências de instruções e não a acções elementares. Isto é importante na implementação de linguagens de programação, pois aumenta a componente sequencial e reduz o peso relativo das transferências de execução.
- O programa em combinadores preserva propriedades do programa original nomeadamente não duplica reduções comuns (cf. "*full-laziness*", secção 3.4.2).

Por outro lado, o nosso modelo de execução — a máquina-ΦG — foi concebido de forma a simultaneamente facilitar a a geração de código de redução de grafos e a transposição para linguagem C:

- Usamos uma arquitectura mais flexível do que a "*G-machine*" original, com algumas características tipo-RISC: um conjunto ilimitado de registos de uso geral e instruções que operam de forma ortogonal com vários tipos de operandos.

- Esta arquitectura é adequada à geração de código C: o código da máquina abstracta é um ficheiro de linguagem C em que cada instrução corresponde a uma "*macro*". Um programa da máquina-ΦG pode ainda ser compilado separadamente e ligado como um programa em C convencional.

- Obtemos código em C com uma granularidade alta: uma função por (super)combinador no programa. As instruções "`Jump(...)`" podem ser optimizadas com o pós-processamento do texto de assemblador (cf. apêndice B).

- O modelo de gestão de memória é *modular* e *robusto*: o código gerado é independente do tipo de colector de lixo empregue e a recolha de lixo pode identificar correctamente todo o espaço livre disponível (cf. secção 4.3).

- É possivel integrar optimizações clássicas para a "*G-machine*" ao nível do procedimento (como a optimização de última chamada, cf. secção 5.4.3).

6.2 Comparação com super-combinadores

A implementação por tradução para combinadores compactos apresenta algumas vantagens sobre a tradução em super-combinadores por "*lambda-lifting*":

- O algoritmo de abstracção combinatória é conceptualmente mais simples que o algoritmo de "*lambda-lifting*". No caso geral, com expressões-`let(rec)` o "*lambda-lifting*" é um processo complexo, particularmente se pretender preservar reduções do programa original ("*full lazy lambda-lifting*") (Peyton Jones, 1995).

- A tradução para combinadores compactos preserva as reduções presentes no programa original, *i.e.*, preserva "*laziness*" (ver secção 3.4.2).

Por outro lado:

- O algoritmo de abstracção não trata directamente expressões-`case` (ver secção 3.3.4). A tradução para combinadores compactos é, portanto, uma alternativa parcial ao "*lambda-lifting*".

- A abstracção de múltiplas variáveis efectua várias passagens sobre um mesmo termo, gerando termos intermédios que não são necessários.

- Muitas vezes o resultado da abstracção de uma variável é um termo combinatório com várias "constantes de ordem superior" (por exemplo: primitivas associadas às operações aritméticas). Desta forma separam-se as operações primitivas das suas aplicações, o que diminui as possibilidades do compilador as optimizar.

O último ponto parece-nos particularmente relevante: idealmente a execução de um programa funcional totalmente estrito não deveria incorrer peso adicional em relação à sua execução numa linguagem imperativa. Apenas os programas que manipulem funções de nível superior deveriam incorrer o custo da redução de grafos.

Como as optimizações de uma expressão são limitadas pelo conhecimento do compilador do seu uso, o "*lambda-lifting*" tem a vantagem em relação à tradução para combinadores de manter a proximidade entre funções e os seus usos.

6.3 Trabalho futuro

A implementação de uma linguagem de programação funcional real necessita de muito mais do que um modelo de redução básico. Um compilador real deve usar várias técnicas de optimização de forma a implementar eficientemente um maior número de programas: nenhuma estratégia de redução por si só é ideal para a implementação de largo espectro de programas (Peyton Jones, 1996).

Neste sentido, um aspecto importante que não foi introduzido é a informação sobre tipos associados às expressões. As anotações de tipos introduzidas pelo programador, bem como a inferência automática de tipos, permitem especializar o código gerado pelo compilador. Esta especialização pode chegar à geração de versões "internas" de super-combinadores recursivos que operam directamente sobre objectos primitivos (por exemplo: inteiros) sem o peso adicional de construir e decompor o grafo. O código resultante torna-se muito próximo do código correspondente implementado numa linguagem imperativa (Peyton Jones, 1996).

Outro aspecto relevante na execução de um programa funcional é a obtenção de *métricas de complexidade* para a redução de grafos, quer em termos de número de reduções efectuadas, quer em termos de espaço do grafo consumido. Um dos obstáculos à maior divulgação da programação funcional é a dificuldade em obter um modelo intuitivo para o programador do espaço e tempo de execução dum programa (Peyton Jones, 1987, cap. 23). Isto torna-se particularmente importante dada a "falha semântica" entre o programa-fonte que o programador concebe e o programa-transformado que é executado. A tradução para combinadores compactos poderá ter alguma vantagem relativa pois preserva mais estrutura do programa original do que o "*lambda-lifting*", nomeadamente em número de reduções.

Apêndice A

Instruções da máquina abstracta

Este apêndice contém uma descrição completa das instruções da máquina-ΦG. Descrevemos a semântica de cada instrução numa linguagem "pseudo-C++":

- usamos construtores (como `INTG(...)`, `AP(...,...)`, etc.) para representar nós do grafo (ver secção 4.3);
- quando uma instrução reserva um novo nó usamos uma declaração "**`new`**";
- quando uma instrução re-escreve um nó usamos apenas o construtor e `*` no lado-esquerdo da atribuição;
- variáveis r e l representam *localizações*, *i.e.*, apontadores para células do grafo;
- usamos $*l$ para desreferenciar um apontador; no caso de apontadores para nós-`INTG` usamos $*l$ para obter o seu valor inteiro.

A.1 Instruções de construção do grafo

Mnemónica	Pseudo-C++	Comentário
`Mkint(`l`,`n`)`	l `= new INTG(`n`);`	"*make*-`INTG`"
`Mkap(`l_1`,`l_2`,`l_3`)`	l_1 `= new AP(`l_2`,`l_3`);`	"*make*-`AP`"
`Pack`$_n$`(`t`,`r`,`$l_1,\ldots,l_n$`)`	r `= new PACK(`n`,` t`,` $l_1,\ldots,l_n$`);`	
`Split`$_n$`(`l`,`$r_1,\ldots,r_n$`)`	r_1 `=` l`[1];...;`r_n `=` l`[`n`];`	
`Update(`l_1`,`l_2`)`	`*`l_1 `= AP(G_indirect,` l_2`);`	
`Stap(`l_1`,`l_2`,`l_3`)`	`*`l_1 `= AP(`l_2`,`l_3`);`	"*store*-`AP`"
`Stint(`l`,`n`)`	`*`l `= INTG(`n`);`	"*store*-`INTG`"
`Add(`l_1`,`l_2`,`l_3`)`	l_1 `= new INTG(*`l_2 `+ *`l_3`);`	"*add*"
`Sub(`l_1`,`l_2`,`l_3`)`	l_1 `= new INTG(*`l_2 `- *`l_3`);`	"*subtract*"
`Mul(`l_1`,`l_2`,`l_3`)`	l_1 `= new INTG(*`l_2 `* (*`l_3`));`	"*multiply*"
`Div(`l_1`,`l_2`,`l_3`)`	l_1 `= new INTG(*`l_2 `/ *`l_3`);`	"*divide*"
`Mod(`l_1`,`l_2`,`l_3`)`	l_1 `= new INTG(*`l_2 `% *`l_3`);`	"*modulo*"
`Neg(`l_1`,`l_2`)`	l_1 `= new INTG(-*`l_2`);`	"*negate*"
`Stadd(`l_1`,`l_2`,`l_3`)`	`*`l_1 `= INTG(*`l_2 `+ *`l_3`);`	"*store-add*"
`Stsub(`l_1`,`l_2`,`l_3`)`	`*`l_1 `= INTG(*`l_2 `- *`l_3`);`	"*store-subtract*"
`Stmul(`l_1`,`l_2`,`l_3`)`	`*`l_1 `= INTG(*`l_2 `* (*`l_3`));`	"*store-multiply*"
`Stdiv(`l_1`,`l_2`,`l_3`)`	`*`l_1 `= INTG(*`l_2 `/ *`l_3`);`	"*store-divide*"
`Stmod(`l_1`,`l_2`,`l_3`)`	`*`l_1 `= INTG(*`l_2 `% *`l_3`);`	"*store-modulo*"
`Stneg(`l_1`,`l_2`)`	`*`l_1 `= INTG(-*`l_2`);`	"*store-negate*"
`Lt(`l_1`,`l_2`,`l_3`)`	l_1 `= (*`l_2`<*`l_3 `? G_True : G_False);`	"*less-than*"
`Lte(`l_1`,`l_2`,`l_3`)`	l_1 `= (*`l_2`<=*`l_3 `? G_True : G_False);`	"*less-than-or-equal*"
`Eq(`l_1`,`l_2`,`l_3`)`	l_1 `= (*`l_2`==*`l_3 `? G_True : G_False);`	"*equal*"
`Neq(`l_1`,`l_2`,`l_3`)`	l_1 `= (*`l_2`!=*`l_3 `? G_True : G_False);`	"*not-equal*"

A.2 Instruções de manipulação da pilha

Mnemónica	Pseudo-C++	Comentário
`Push(`l`)`	`*(--Stk) =` l`;`	colocar no topo da pilha
`Pop(`l`)`	l `= *(Stk++);`	retirar do topo da pilha
`Drop(`n`)`	`Stk+=`n`;`	eliminar n elementos
`Slide(`n`)`	`Stk+=`n`; *Stk=Stk[-`n`];`	desliza o elemento do topo n posições

`Trim(`n`,`m`)`	`for(i=`$n-1$`; i>=0; i--)`
	`Stk[i+`m`] = Stk[i];`
	`Stk +=` m`;`
		desliza n elementos m posições

A.3 Instruções de transferência de execução

Mnemónica	Pseudo-C++	Comentário
`Switch(`l`)` $\{\cdots\}$	`switch(*`l`)` $\{\cdots\}$	escolha entre alternativas
`Jump(`f`)`	f`(); return;`	salto incondicional (ver apêndice B)
`Return()`	`return;`	terminar redução
`Eval(`l`)`	l `= doEval(`l`);`	reduzir o grafo l à forma-normal fraca (ver secção 4.2.4).
`Unwind()`	`if(UNWINDABLE(Frame)) { cont = doUnwind(); (*cont)(); } return;`	continuar a redução a partir de `Frame` (ver secção 4.2 e o apêndice B)

A.4 Miscelânea

Mnemónica	Pseudo-C++	Comentário
`Set(`l_1`,`l_2`)`	l_1 `=` l_2`;`	atribuição de apontadores
`ArgsCheck(`n`)`	`if(Frame-Stk<`n`) return;`	verificar se existem n argumentos
`Mkargs(`n`)`	—	re-estruturar os n argumentos
`HeapGet(`n`)`	—	reservar n células de "*heap*"
`Label(`f`)`	`void` f`(void) {...`	declaração de função
`EndLabel`	`...}`	fim de função
`Reg(`r_n`)`	`register void *`r_n`;`	declaração de registo

A.4.1 Função `doUnwind`

```
code_ptr doUnwind(void) {
    register cell_ptr p;
    p = *Stk;
    if(TAG(*p) == TAG_AP) {
        do {                /* enquanto p é um nó-AP */
            p = LINK(*p);   /* descer pelo nó esquerdo */
            Push(p);        /* colocar apontador na pilha */
        } while(TAG(*p) == TAG_AP);
    }
    return CODE(p[1]);      /* retorna endereço no nó-CC */
}
```

A.4.2 Função doEval

```
cell_ptr doEval(cell_ptr l) {
        stack_ptr saved_frame;
        if(UNWINDABLE(l)) {
          saved_frame = Frame;
          Push(l);
          Frame = Stk;
          cont = doUnwind();
          (*cont)();
          Stk = Frame;
          Pop(l);
          Frame = saved_frame;
        }
        return l;
}
```

Apêndice B

Optimização de última chamada em C

B.1 Objectivo

Este apêndice descreve uma optimização ao código gerado pelo compilador de C que transforma a última chamada duma função num salto incondicional. A optimização de última chamada é essencial, pois o código de máquina abstracta em C usa uma última chamada para efectuar um salto incondicional; sem esta optimização a pilha de execução de C seria consumida desnecessariamente.

Esta optimização não é, em geral, efectuada pelos compiladores de C ou de outras linguagens imperativas (Bacon et al., 1994), pois não é uma situação comum em código escrito directamente por um programador (este escreveria o código de forma iterativa).

No artigo (Peyton Jones et al., 1997) foi proposta uma variante da linguagem C (com o nome sugestivo de "`C--`") concebida com o intuito de ser mais adequada à geração de código intermédio dum compilador. A possibilidade de explicitamente invocar uma função com optimização de última chamada é uma das modificações propostas.

Entretanto, e enquanto uma tal linguagem não se encontra difundida, optamos por uma solução de compromisso:

- geramos código C de uma forma correcta mas ineficiente, *i.e.*, usando chamada seguida de retorno em vez de salto;

- pós-processamos o texto de assemblador gerado pelo compilador de C de forma a substituir chamadas terminais por saltos incondicionais.

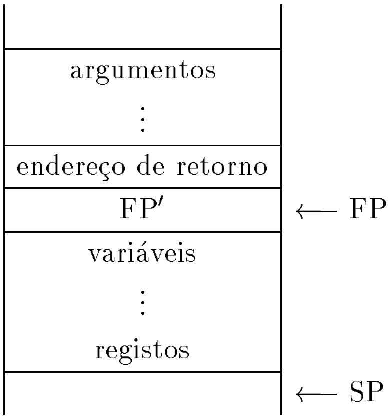

Figura B.1: Estrutura de um contexto de activação

Esta solução pressupõe, claro está, que o compilador de C permite gerar texto de assemblador e não só o código-objecto; isto é possível no compilador de C GNU usado no nosso desenvolvimento (Stallman et al., 1997) e largamente difundido na comunidade académica.

Contudo, esta solução fica obviamente dependente da arquitectura do processador físico (ou pelo menos da sua linguagem assemblador). Esperamos, no entanto, que o trabalho necessário para transpor o pós-processamento para outras linguagens de assemblador seja bastante menor que o trabalho da geração de código propriamente dito.

B.2 Implementação na arquitectura i386

A nossa implementação foi efectuada para a família i386 usando o compilador de C GNU no sistema operativo Linux.

O pós-processador consiste num programa em C, parte escrito usando o utilitário `flex` de construção de analisadores léxicos. O programa funciona como um filtro que recebe pela entrada-padrão um texto de assemblador e escreve na saída-padrão o texto já processado.

A sintaxe do assemblador GNU para i386 segue as convenções da *AT&T/System V* que diferem ligeiramente da sintaxe Intel descrita em (Intel, 1996): por exemplo, em instruções do tipo `mov A,B` a atribuição é de `A` para `B`, que é a ordem inversa da sintaxe Intel. Para uma descrição completa das diferenças, remetemos para a documentação do assemblador GNU `as` (Stallman et al., 1997).

B.2.1 Estrutura dum contexto de activação

A execução de uma função em C é implementada associando-lhe uma área da pilha de execução designada *contexto de activação*. O contexto de activação contém toda a informação que é necessário preservar entre invocações de funções e que inclui (ver figura B.1):

- valores de argumentos passados à função;
- endereço de retorno da execução;
- valores de registos temporários;
- variáveis locais à função (que não estão em registos);
- espaço de trabalho temporário (por exemplo: argumentos para outras funções invocadas).

Nem todos estes campos estão necessariamente presentes: uma função pode não ter argumentos ou não usar variáveis locais.

O contexto de activação é criado quando uma função (designada "*caller*") invoca outra (designada "*callee*"). Parte do novo contexto de activação é construído pelo código da função "*caller*": a colocação de argumentos e a execução da instrução `call` (que coloca o endereço de retorno na pilha). Os restantes campos são inicializados pela função "*callee*".

É comum os compiladores de C usarem um registo auxiliar como apontador do contexto de activação, designado "*frame-pointer*", que serve de base para endereçar os argumentos da função.

A função "*callee*" coloca o seu valor de retorno num registo fixo: no caso de valores inteiros ou apontadores, o registo `%eax`.

B.2.2 Estrutura do código

O código associado à função "*callee*" é executado quando os seus argumentos e o endereço de retorno já foram colocados na pilha (pela função "*caller*"). O código divide-se então em três partes distintas:

inicialização: código que guarda registos temporários na pilha e inicializa o "*frame-pointer*";

execução: código de execução da função propriamente dita;

retorno: código de retorno da função que recupera os registos temporários, o "*frame-pointer*" anterior e retorna à função que a invocou.

Note-se que a sequência de retorno pode ocorrer replicada em diferentes alternativas do código da função (se o compilador empregar técnicas de optimização como "*code-motion*").

B.2.3 Eliminação da última chamada

Para eliminar uma última chamada não é suficiente transformar uma instrução `call` numa instrução `jmp`: é necessário executar primeiro a sequência de retorno associada à função (com excepção da instrução `ret` propriamente dita) de forma a eliminar a parte do contexto de activação construído pela função "*callee*".

No caso do código da máquina-ΦG todas as funções associadas a combinadores não têm argumentos nem valor de retorno (*i.e.*, são de tipo "`void ...(void)`"). Assim, os contextos de activação não incluem a zona de argumentos representada na figura B.1. Isto torna possível eliminar o contexto de activação (à excepção do endereço de retorno) e saltar para o código de outra função mantendo a semântica de "*call*" seguido de "*return*" mas evitando consumir a pilha de execução.

Exemplo 18 Considerando a função `foo` abaixo:

```
extern int a;
void foo(void) {
    if(a) { bar1(); return; } else { bar2(); return; }
}
```

O código gerado pelo compilador de C seria:

```
foo:  pushl %ebp          /* guarda ‘frame-pointer’ */
      movl %esp,%ebp      /* novo ‘frame-pointer’ */
      cmpl $0,a
      je .L2
      call bar1
      movl %ebp,%esp      /* sequência de retorno #1 */
      popl %ebp
      ret
.L2:  call bar2
      movl %ebp,%esp      /* sequência de retorno #2 */
      popl %ebp
      ret
```

Após eliminação de última chamada o código deverá ser:

```
foo:  pushl %ebp          /* guarda 'frame-pointer' */
      movl %esp,%ebp      /* novo 'frame-pointer' */
      cmpl $0,a
      je .L2
      movl %ebp,%esp      /* sequência de retorno #1 */
      popl %ebp
      jmp bar1
.L2:  movl %ebp,%esp      /* sequência de retorno #2 */
      popl %ebp
      jmp bar2
```

B.2.4 Pós-processamento

O pós-processador opera sobre o texto de assemblador gerado pelo compilador de C. Durante a geração de código C acrescentamos algumas "anotações" que permitem identificar as chamadas a optimizar e a sequência de retorno.

A função `foo` do exemplo 18 será escrita em C forma ligeiramente diferente:

```
extern int a;
void foo(void) {
    if(a) { asm("#JUMP bar1"); goto _ret; }
    else { asm("#JUMP bar2"); goto _ret; }
_ret:  asm("#RETURN");
}
```

As anotações `asm(...)` são introduzidas re-definindo as "*macros*" do pré-processador que correspondem às instruções da máquina abstracta. O texto dentro de `asm(...)` é passado sem alteração para o código de assemblador (note-se que o código resultante não é directamente assemblável nesta fase):

```
foo:  pushl %ebp          /* inicializar 'frame-pointer' */
      movl %esp,%ebp
      cmpl $0,a
      je .L2
      #JUMP bar1
      jmp .L3
.L2:
      #JUMP bar2
.L3:
```

```
      #RETURN
      movl %ebp,%esp      /* sequência de retorno */
      popl %ebp
      ret
```

O pós-processador pode agora identificar a sequência de retorno da função pela anotação "`#RETURN`" e quais as chamadas a optimizar pela anotação "`#JUMP`".

É necessário replicar a sequência de retorno em cada anotação "`#JUMP`", seguida pela instrução "`jmp`" correspondente. As anotações são removidas do código final:

```
foo:  pushl %ebp          /* inicializar 'frame-pointer' */
      movl %esp,%ebp
      cmpl $0,a
      je .L2
      movl %ebp,%esp      /* sequência de retorno #1 */
      popl %ebp
      jmp bar1
      jmp .L3
.L2:  movl %ebp,%esp      /* sequência de retorno #2 */
      popl %ebp
      jmp bar2
.L3:  movl %ebp,%esp
      popl %ebp
      ret
```

Note-se que ainda resulta algum "código morto" deste processamento que poderia ser eliminado usando técnicas clássicas de optimização em compiladores.

B.2.5 Geração de código

Para possibilitar o pós-processamento são necessárias algumas definições alternativas de algumas "*macros*" de instruções da máquina abstracta (que na implementação são seleccionadas por uma opção de compilação). A geração de código abstracto propriamente dito não é alterada, bem como a semântica de "*call/return*" entre funções. Em particular, isto permite que funções compiladas sem optimização de última chamada invoquem funções compiladas sem essa optimização e vice-versa.

Execução de uma função

O código de cada função termina com uma anotação que marca a sequência de retorno:

```
void ...(void) {
    ⋮
    _ret:  asm("#RETURN");
}
```

Note-se que a anotação também impede o compilador de C de replicar a sequência de retorno em outros pontos da função (uma vez que o compilador não pode prever o efeito de anotações `asm`).

Execução de um `Jump`

Um salto incondicional `Jump` é implementado usando uma anotação "`#JUMP`":[1]

```
#define Jump(l)        asm("#JUMP "#l);
```

Execução de `Return`

O retorno directo (*i.e.*, sem mais reduções, quando o resultado já está em whnf) é efectuado transferindo a execução para a sequência `_ret`:

```
#define Return()       goto _ret;
```

Execução de `Unwind`

A execução de um `Unwind` é decomposta em três passos:

1. Se o grafo apontado por `*Frame` não é uma aplicação ou um combinador, então encontra-se já em forma-normal fraca e devemos retornar imediatamente.

2. Caso contrário, encontramos o endereço de código a executar usando a função `doUnwind()` da biblioteca de execução, que efectua a descida pelos nós-esquerdos das aplicações, colocando apontadores para nós intermédios na pilha (cf. secção 4.2).

[1] Note-se que a "*macro*" `Jump` usa uma extensão do pré-processador cpp da GNU que permite tomar o valor de um argumento como uma "*string*".

3. Efectuamos um salto indirecto por meio do registo `%eax` com o valor retornado por `doUnwind`.

```
#define Unwind()      if(UNWINDABLE(*Frame)) goto _ret; \
                      asm("call doUnwind"); asm("#JUMP *%eax");
```

Bibliografia

Aho, A. V., Sethi, R. & Ullman, J. D. (1986). "*Compilers: Principles, Techiques and Tools*", Addison-Wesley.

Aït-Kaci, Hassan. (1991). "*Warren's Abstract Machine: A Tutorial Reconstruction*", The MIT Press.

Bacon, D. F., Graham, S. L. & Sharp, O. J. (1994). "*Compiler Transformations for High-Performance Computing*", *ACM Computing Surveys*, 26, 345–420.

Bird, R. & Wadler, P., (1988). "*Introduction to Functional Programming*", Prentice-Hall International.

Broda, S. (1996). "Sobre Bases de Combinadores para Sistemas de λ-calculus". Tese de Doutoramento submitida ao Departamento de Ciência de Computadores da Faculdade de Ciências da Universidade do Porto.

Broda, S. & Damas, L. (1995). "*A New Translation Algorithm from Lambda Calculus into Combinatory Logic*". Pág. 359–370 em: Mamede & Pinto-Ferreira (eds), *7th Portuguese Conference in Artificial Intelligence*. LNAI, vol. 990, Springer Verlag.

Broda, S. & Damas, L. (1997). "*Compact Bracket Abstraction in Combinatory Logic*", *Journal of Symbolic Logic*, vol. 62, n. 3.

Jones, Mark P. (1994). "*The Implementation of the Gofer functional programming system*", Research report, Department of Computer Science, Yale University.

Hankin, C. (1994). "*Lambda Calculi: a guide for computer scientists*", Oxford University Press.

Hennessy, J. L., Patterson, D. A. (1996). "*Computer Architecture: A quantitative approach*" (2ª ed.), Morgan Kaufmann.

Intel (1996). "*Pentium Pro Family Developer's Manual, Vol. 2: Programmer Reference Manual*", Order Number 242691.

Knuth, D. E. (1971). "*The Art of Computer Programming, vol.1: Fundamental Algorithms*" ($2^{\underline{a}}$ ed.), Addison-Wesley.

Kogge, P. M. (1991). "*The Architecture of Symbolic Computers*", McGraw-Hill.

Peyton Jones, S. L. (1987). "*The Implementation of Functional Programming Languages*", Prentice-Hall International.

Peyton Jones, S. L. (1993). "*Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*", Technical report, Department of Computer Science, University of Glasgow.

Peyton Jones, S. L. (1995). "*A modular fully-lazy lambda lifter in Haskell*", Technical report, Department of Computer Science, University of Glasgow.

Peyton Jones, S. L. (1996). "*Compiling Haskell by program transformation: a report from the trenches*", Technical report, Department of Computer Science, University of Glasgow.

Peyton Jones, S. L. & Lestes, D. (1992). "*Implementing Functional Programming Languages: A Tutorial*", Prentice-Hall International.

Peyton Jones, S. L. et al.. (1992). "*The Glasgow Haskell compiler: a technical overview*", em *Proceedings of UK Joint Framework for Information Technology*, Department of Computing Science, University of Glasgow.

Peyton Jones, S. L., Nordin, T., Oliva, D. (1997). "`C--`*: A Portable Assembly Language*". To appear in the *Proceedings of the IFL'97*, Department of Computer Science, University of Glasgow.

Peterson, J. & Hammond, K. (eds) (1996). "*Report on the Programming Language Haskell, version 1.3*", Technical report, Department of Computer Science, Yale University.

Stallman, R. et al.. (1997). Documentação do compilador GNU C e utilitários associados, versão 2.7.2. "*Software*" de distribuição livre; "*site*" oficial: `http://www.fsf.org`); "*mirror*" na Universidade do Porto: `ftp://ftp.ncc.up.pt/pub/gnu/`.

Turner, D. A. (1979a). "*Another algorithm for bracket abstraction*", em *The Journal of Symbolic Logic*, vol. 44, n. 2.

Turner, D. A. (1979b). "*A New Implementation Technique for Applicative Languages*", em *Software Practice and Experience*, vol. 9, 31–49.