Redução de grafos

Pedro Vasconcelos

8 de Maio de 2020

Semântica estrita vs. não-estrita

Como implementar $((\lambda x.\, M)\ N)$?

Semântica estrita

Call-by-value avaliar N uma vez (mesmo quando não usado)

Semântica não-estrita

Call-by-name avaliar N zero ou múltiplas vezes (sempre que for usado)

Call-by-need, lazy evaluation avaliar N zero ou uma vez (apenas se for usado)

Semântica estrita vs. não-estrita

Semântica estrita

- Usual em todas as linguagens imperativas
- Também em Scheme, Standard ML, Caml e F#
- Avaliação não-estrita é usada nalguns casos:
  - operadores lógicos `&&`, `||`
  - parametros declarados *call-by-name*
  - estruturas de dados especiais (e.g. *streams*)
  - *macros*

Semântica não-estrita

- *Call-by-name* é ineficiente para uso geral
- *Call-by-need/lazy evaluation* em linguagens funcionais puras (Miranda, Clean, Haskell)

Dificuldades de implementação

- *Call-by-name* é ineficiente: leva à duplicação de computações
- *Call-by-need/lazy evaluation*:
  - necessitamos de representar *computações suspensas* (*thunks*)
  - actualizar um *thunk* com o resultado após a avaliação (*update-in-place*)
  - ter o cuidado de *partilhar thunks*

Exemplo

$$\begin{aligned} \text{sqr} &= \lambda x.\, \text{mult}\ x\ x \\ \text{main} &= \text{sqr}\ (\text{sqr}\ 5) \end{aligned}$$

Vamos calcular a expressão “main”.

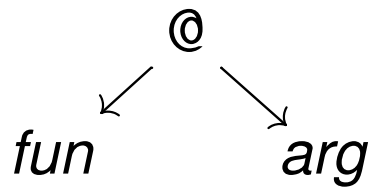
Redução *call-by-name*

Usando ordem normal

$$
\begin{aligned}
\underline{\text{sqr}}\ (\text{sqr}\ 5) &\to \underline{\text{mult}}\ \underbrace{(\text{sqr}\ 5)\ (\text{sqr}\ 5)}_{\text{duplicação}} \\
&\to \text{mult}\ (\underline{\text{sqr}}\ 5)\ (\text{sqr}\ 5) \\
&\to \text{mult}\ (\underline{\text{mult}}\ 5\ 5)\ (\text{sqr}\ 5) \\
&\to \text{mult}\ 25\ (\underline{\text{sqr}}\ 5) \\
&\to \text{mult}\ 25\ (\underline{\text{mult}}\ 5\ 5) \\
&\to \underline{\text{mult}}\ 25\ 25 \\
&\to 625
\end{aligned}
$$

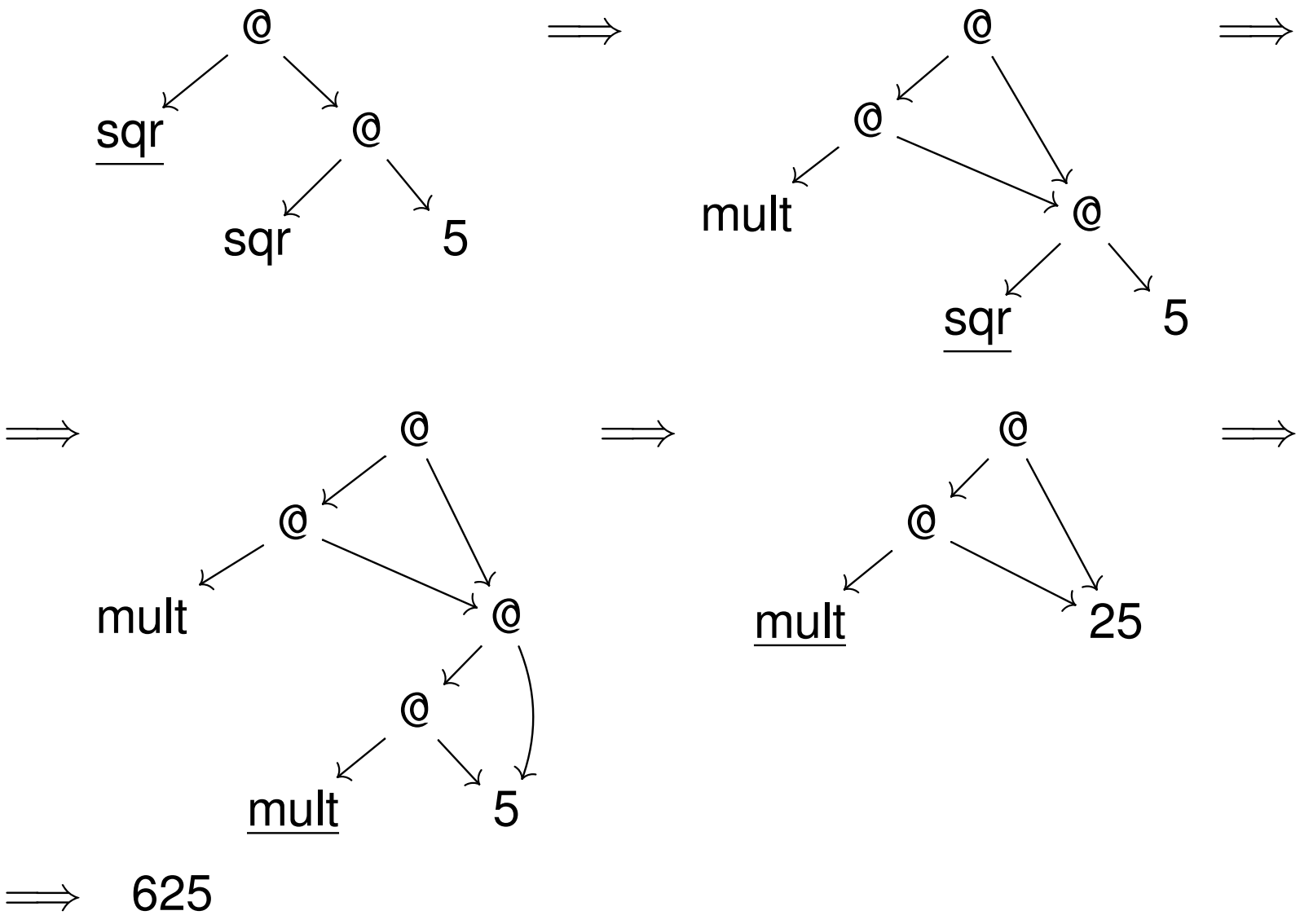
Redução de grafos

- Representamos o termo como um grafo em que as aplicações são nós binários.



- Procuramos o *redex* mais exterior e mais à esquerda (ordem normal)
- Após reduzir um sub-termo actualizamos o grafo com o resultado

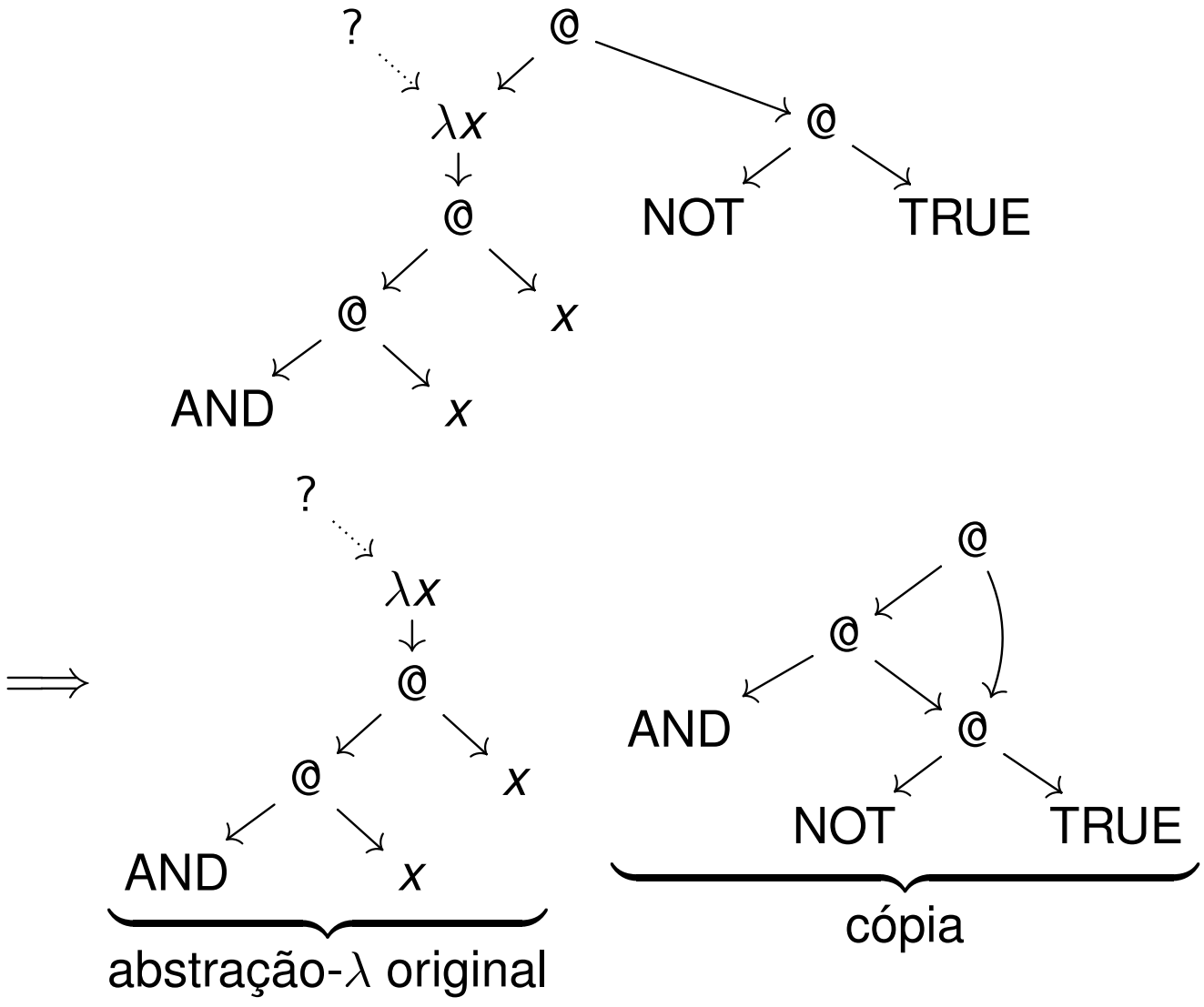
Exemplo



Observações

- A partilha de sub-grafos evita a duplicação de *redexes*
- O grafo pode ser implementado em memória como uma estrutura ligada
- Cada passo de re-escrita apenas modifica apontadores da estrutura
- Problema: necessitamos de copiar o corpo da abstração-λ ao efectuar uma redução-β

Exemplo


?
@
λx
@
@
NOT
TRUE
@
x
AND
x
⟹
?
λx
@
@
x
AND
x
abstração-λ original
@
@
AND
@
NOT
TRUE
cópia

Evitar duplicações

Podemos evitar a necessadade de cópia se efecutarmos redução de combinadores em vez de λ-termos.

Combinadores

Termos da lógica combinatória são constituidos por:

1. variáveis $x, y, z \ldots$;
2. constantes $\mathbf{S}$, $\mathbf{K}$, $\mathbf{I}$;
3. aplicações $(M\,N)$ onde M e N são termos.

Exemplos:

$$((\mathbf{S}\,x)\,\mathbf{I}) \qquad ((\mathbf{S}\,\mathbf{K})\,\mathbf{K}) \qquad ((\mathbf{S}\,(\mathbf{K}\,\mathbf{S}))\,\mathbf{K})$$

Como no cálculo-λ, omitimos parêntesis convencionando que a aplicação associa à esquerda, e.g.

$$\mathbf{S}\,\mathbf{K}\,\mathbf{K} \equiv ((\mathbf{S}\,\mathbf{K})\,\mathbf{K})$$

Combinadores (cont.)

Regras de redução $\rightarrow_w$ (*weak*):

$$\mathbf{I}\,P \rightarrow_w P$$
$$\mathbf{K}\,P\,Q \rightarrow_w P$$
$$\mathbf{S}\,P\,Q\,R \rightarrow_w P\,R\,(Q\,R)$$

Exemplo:

$$\mathbf{S}\,\mathbf{K}\,\mathbf{K}\,x \rightarrow_w \mathbf{K}\,x\,(\mathbf{K}\,x) \rightarrow_w x$$

Um termo que não admite mais reduções diz-se em forma normal fraca. Exemplos: x, $\mathbf{I}$, $\mathbf{S}\,\mathbf{K}$, $\mathbf{S}\,\mathbf{K}\,\mathbf{K}$.

Tradução do cálculo-λ em combinadores

Podemos traduzir qualquer λ-termo em combinadores usando duas transformações sintáticas:

$(-)_{CL}$ converte um λ-termo em combinadores;

$\lambda^* x$ abstrai uma variável num termo de combinadores.

$$
\begin{aligned}
(x)_{CL} &\equiv x \\
(M\,N)_{CL} &\equiv (M)_{CL}(N)_{CL} \\
(\lambda x.\, M)_{CL} &\equiv \lambda^* x.(M)_{CL}
\end{aligned}
$$

$$
\begin{aligned}
\lambda^* x.\, x &\equiv \mathbf{I} \\
\lambda^* x.\, P &\equiv \mathbf{K}\,P, \qquad \text{se } x \notin fv(P) \\
\lambda^* x.\, P\,Q &\equiv \mathbf{S}\,(\lambda^* x.\, P)\,(\lambda^* x.\, Q)
\end{aligned}
$$

Exemplo de tradução

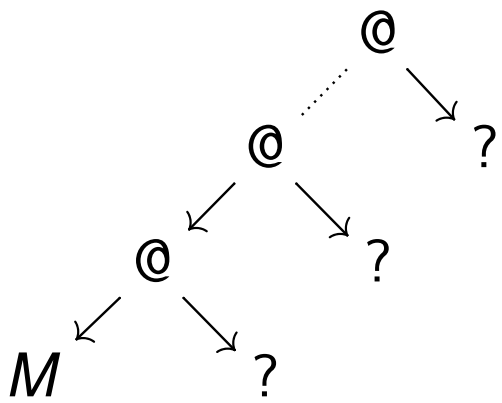
$$
\begin{aligned}
(\lambda xy.\, y\, x)_{CL} &\equiv \lambda^* x.\, \lambda^* y.\, (y\, x)_{CL} \\
&\equiv \lambda^* x.\, \lambda^* y.\, (y\, x) \\
&\equiv \lambda^* x.\, \mathbf{S}\, (\lambda^* y.\, y)\, (\lambda^* y.\, x) \\
&\equiv \lambda^* x.\, (\mathbf{S}\, \mathbf{I})\, (\mathbf{K}\, x) \\
&\equiv \mathbf{S}\, (\lambda^* x.\, \mathbf{S}\, \mathbf{I})\, (\lambda^* x.\, \mathbf{K}\, x) \\
&\equiv \mathbf{S}\, (\mathbf{K}\, (\mathbf{S}\, \mathbf{I}))\, (\mathbf{S}\, (\lambda^* x.\, \mathbf{K})\, (\lambda^* x.\, x)) \\
&\equiv \mathbf{S}\, (\mathbf{K}\, (\mathbf{S}\, \mathbf{I}))\, (\mathbf{S}\, (\mathbf{K}\, \mathbf{K})\, \mathbf{I})
\end{aligned}
$$

Propriedades da tradução

- $(M)_{CL}$ é "equivalente" a M (ver bibliografia — *Foundations of FP*)
- A tradução *não* preserva formas normais (não é importante na prática)
- Cada λ^* pode duplicar o número de aplicações
- O termo em combinadores pode ser exponencialmente maior do que o λ-termo original
- A tradução pode ser melhorada para complexidade quadrática usando mais combinadores (Turner, 1979)
- Podemos ainda acrescentar constantes e operações primitivas e um combinador ponto-fixo para recursão

Redução de combinadores usando grafos

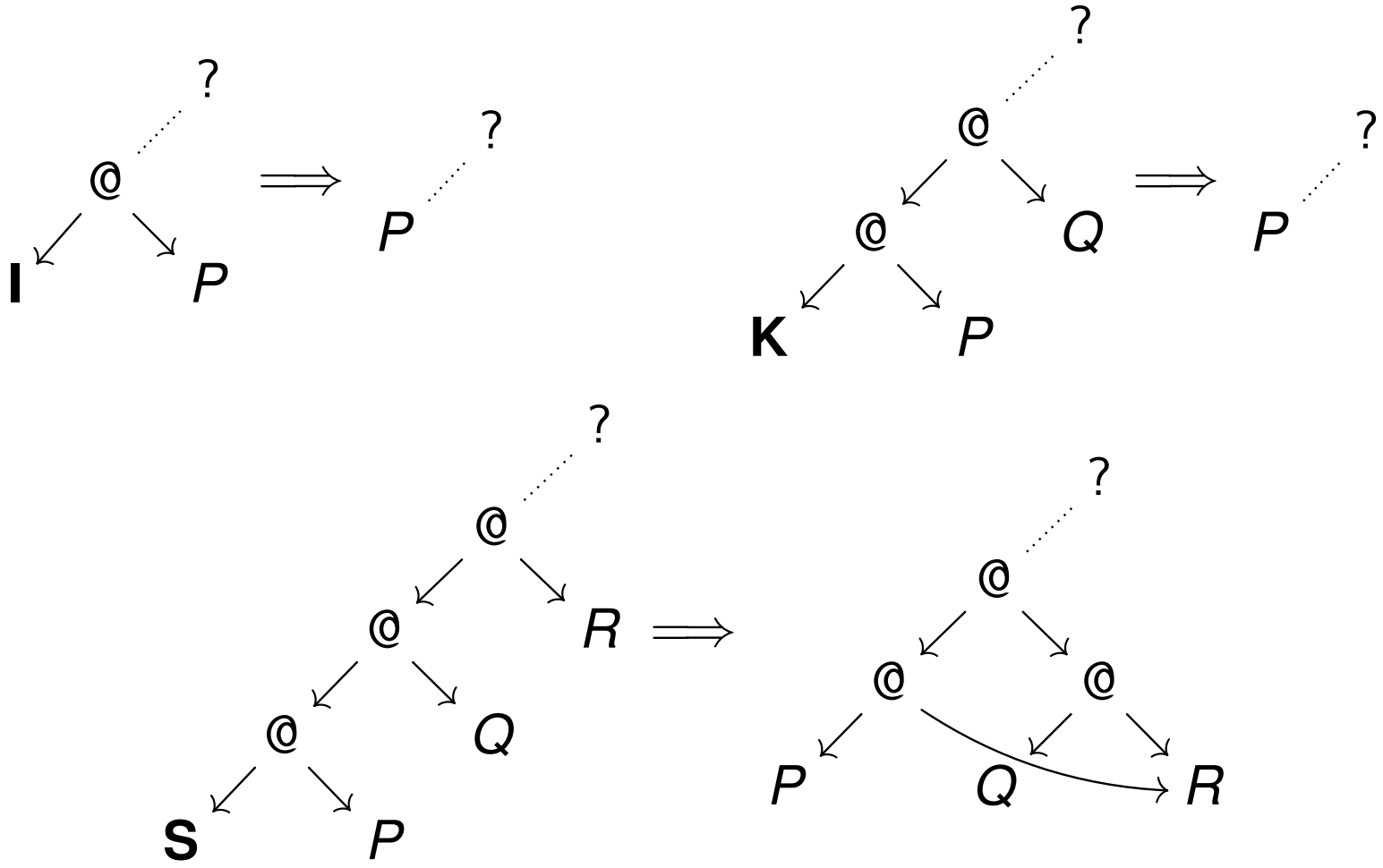
Unwind: procurar o *redex* mais exterior e mais à esquerda descendo pela esquerda dos nós-@.



tal que *M* não é um nó-@

Redução de combinadores usando grafos (cont.)

Se encontrou o *redex*, efectua uma re-escrita local e repete o processo de *unwind*.



Redução de combinadores usando grafos (cont.)

Se não encontrou um *redex* a redução termina: o grafo está em forma normal fraca (*weak head normal form*).

Supercombinadores

Em vez dum conjunto pré-determinado de combinadores podemos traduzir um programa em combinadores especializados.

Um supercombinador é um termo-λ sem variáveis livres da forma

$$\lambda x_1 . \lambda x_2 . \ldots \lambda x_n . M$$

tal que M não é uma abstração e todas as abstrações em M são também supercombinadores.

Note ainda que $n \geq 0$, i.e. podemos não ter λs.

Exemplos

São supercombinadores:

$$\lambda x.\, x$$
$$\lambda f.\lambda x.\, f\,(f\,x)$$
$$\lambda x.\, +\ x\ 1$$
$$(+\ 2\ 5)$$
$$42$$

Não são supercombinadores:

$\lambda x.\, y$	y ocorre livre
$\lambda y.\, +\ y\ x$	x ocorre livre
$\lambda f.\, f\,(\lambda x.\, f\,x\,2)$	a abstração λx não é supercombinador (f ocorre livre)

Lambda-lifting

A transformação de um λ-termo noutro em que todas as abstrações são supercombinadores chama-se lambda-lifting.

Exemplo: transformar o λ-termo

$$(\lambda x.\,(\lambda y.\; +\;\; y\; x)\; x)\; 4$$

em que nenhuma das abstrações é um supercombinador.

Lambda-lifting (cont.)

Começamos pela abstração interior, substituimos a variável livre x por um argumento extra z e uma aplicação:

$$
\begin{aligned}
&(\lambda x.\, \underline{(\lambda y.\, +\ y\ x)}\ x)\ 4 \\
=&(\lambda x.\, (\underbrace{(\lambda z.\, \lambda y.\, +\ y\ z)}_{\$F}\ x)\ x)\ 4 \\
=&\underbrace{(\lambda x.\, \$F\ x\ x)}_{\$G}\ 4 \\
=&\$G\ 4
\end{aligned}
$$

Lambda-lifting (cont.)

Obtemos a definição de dois supercombinadores:

$$\$F = \lambda z.\, \lambda y.\, +\ y\ z$$
$$\$G = \lambda x.\, \$F\ x\ x$$

A expressão original é transformada em

$$\$G\ 4$$

Lambda-lifting (cont.)

Podemos interpretar as definições dos supercombinadores como regras de re-escrita:

$$\$F\ z\ y \rightarrow_w +\ y\ z$$
$$\$G\ x \rightarrow_w \$F\ x\ x$$

Efectuando a redução da expressão:

$$\begin{aligned}\$G\ 4 &\rightarrow_w \$F\ 4\ 4\\ &\rightarrow_w +\ 4\ 4\\ &\rightarrow_w 8\end{aligned}$$

Algoritmo de *lambda-lifting*

Enquanto existem abstrações-λ:

1. escolher uma abstração-λ mais interior (i.e. sem subtermos abstrações-λ);
2. abstrair todas as suas variáveis livres como novos parametros;
3. acrescentar a definição de um novo supercombinador (e.g. $\$F_{n+1}$ onde n é o número actual de supercombinadores);
4. substituir a ocorrência da abstração-λ pelo supercombinador aplicado às variáveis livres.

Mais detalhes: *The Implementation of Functional Programming Languages*, Simon L. Peyton Jones, 1987, capítulos 13 & 14.

Redução de supercombinadores

Para fazer redução de grafos necessitamos de implementar as regras de re-escrita do grafo para cada um dos supercombinadores usados.

Ao contrário dos combinadores SKI, o conjunto de supercombinadores não é fixo—i.e. depende de cada programa concreto.

Redução de supercombinadores (cont.)

Duas alternativas:

1. efectuar cópia do corpo do supercombinador (abstrações-λ);
2. compilar código para efectuar a re-escrita do grafo na definição.

A segunda alternativa possibilita muitas optimizações durante a compilação e a permite implementação muito mais eficientes.

A G-machine é uma máquina virtual para compilação de supercombinadores em código intermédio que efectua redução do grafos.