

Semânticas de Linguagens de Programação

Exercícios 6

Máquina abstracta AM para a linguagem While

1. Determina o que calcula o código p seguinte executando $\langle p, \varepsilon, s \rangle$, com $s(x) = 10$ e $s(y) = 6$.

```
PUSH(0) : STORE(z) : FETCH(x) : STORE(r) :
LOOP(FETCH(r) : FETCH(y) : LE, FETCH(y) : FETCH(r) : SUB : STORE(r) :
PUSH(1) : FETCH(z) : ADD : STORE(z))
```

2. Verifica que com $s(x) = 3$,

$$\begin{aligned}
 & \mathcal{S}_{am} \llbracket y := 1; \text{while } \neg(x = 1) \text{ do } (y := y \times x; x := x - 1) \rrbracket s \\
 = & \mathcal{M} \llbracket \text{PUSH}(1) : \text{STORE}(y) : \text{LOOP}(\text{PUSH}(1) : \text{FETCH}(x) : \text{EQ} : \text{NEG}, \\
 & \text{FETCH}(x) : \text{FETCH}(y) : \text{MULT} : \text{STORE}(y) : \text{PUSH}(1) : \\
 & \text{FETCH}(x) : \text{SUB} : \text{STORE}(x)) \rrbracket s \\
 = & s[x \mapsto 1][y \mapsto 6]
 \end{aligned}$$

Em geral o que calcula?

3. Gera código **AM** para os seguintes programas:

- (a) $z := x; y := 0; \text{while } \neg z = 0 \text{ do } (y := y + 1; z := z - 1)$
- (b) $z := 0; \text{while } y \leq x \text{ do } (z := z + 1; x := x - y)$

Para cada um deles verifica que a semântica do código coincide com a semântica operacional natural dos programas, considerando o estado inicial s onde $s(x) = 2$ e $s(y) = 1$.

4. Prova que se $\langle p_1, e_1, s_1 \rangle \triangleright^k \langle p', e', s' \rangle$ então $\langle p_1 : p_2, e_1 : e_2, s_1 \rangle \triangleright^k \langle p' : p_2, e' : e_2, s' \rangle$.
5. Prova que se $\langle p_1 : p_2, e, s \rangle \triangleright^k \langle \varepsilon, e'', s'' \rangle$ então existem k_1 e k_2 com $k = k_1 + k_2$ tal que $\langle p_1, e, s \rangle \triangleright^{k_1} \langle \varepsilon, e', s' \rangle$ e $\langle p_2, e', s' \rangle \triangleright^{k_2} \langle \varepsilon, e'', s'' \rangle$.
6. Mostra que a semântica operacional de **AM** é determinística.
7. Implementa em Haskell a máquina abstracta **AM**, a sua semântica operacional e a tradução da linguagem **While** em código **AM**.
8. ((* **AM**₁) Modifica a máquina abstracta **AM** de modo a que não se usem nomes de variáveis mas sim posições de memória. As configurações são agora da forma

$$\langle p, e, m \rangle \in \mathbf{Code} \times \mathbf{Stack} \times \mathbb{Z}^*$$

onde $m \in \mathbb{Z}^*$ representa uma lista de valores (memória) e $len(m)$ representa o seu comprimento. Considera as instruções GET(n) e PUT(n) onde n é um endereço de memória com $1 \leq n \leq len(m)$ e $m[n]$ dá o valor na posição de memória de endereço n .

- (a) Especifica a semântica operacional da máquina abstracta **AM**₁
 - (b) Traduz a linguagem **While** para **AM**₁. Supõe um ambiente de variáveis $env : \mathbf{Var} \rightarrow \mathbb{N}$ que mapeia o nome de variáveis aos seus endereços.
9. ((* **AM**₂) Modifica a máquina **AM**₁ do exercício anterior, para usar instruções de salto (mais habituais) em vez de LOOP e BRANCH. Para tal temos de ter instruções que definem etiquetas e saltos para etiquetas.

- As configurações serão da forma $\langle pc, p, e, m \rangle$ onde pc é o contador de programa (um inteiro que aponta para uma instrução em p . Em geral o valor de pc é incrementado de 1 em cada passo.
- Em vez de LOOP e BRANCH definem-se as instruções:
 - LABEL- ℓ : que actualiza o contador de programa com ℓ (um inteiro que corresponde a endereço duma instrução)
 - JUMP- ℓ : mover o contador de programa para a instrução LABEL- ℓ (se existir tem de ser única).
 - JUMPFALSE- ℓ : como anterior mas só se no topo da pilha está o valor `false`; se for `true` o contador de programa é incrementado

Especifica a semântica operacional da máquina abstracta $\mathbf{AM}_2$. Podes referir por $p[pc]$ a instrução na posição pc .

Semântica operacional de AM e tradução da linguagem While

$$\begin{aligned}
\langle \text{PUSH}(n) : p, e, s \rangle &\triangleright \langle p, \mathcal{N}[n] : e, s \rangle \\
\langle \text{ADD} : p, z_1 : z_2 : e, s \rangle &\triangleright \langle p, (z_1 + z_2) : e, s \rangle \quad \text{se } z_1, z_2 \in \mathbb{Z} \\
\langle \text{MULT} : p, z_1 : z_2 : e, s \rangle &\triangleright \langle p, (z_1 * z_2) : e, s \rangle \quad \text{se } z_1, z_2 \in \mathbb{Z} \\
\langle \text{SUB} : p, z_1 : z_2 : e, s \rangle &\triangleright \langle p, (z_1 - z_2) : e, s \rangle \quad \text{se } z_1, z_2 \in \mathbb{Z} \\
\langle \text{TRUE} : p, e, s \rangle &\triangleright \langle p, \text{true} : e, s \rangle \\
\langle \text{FALSE} : p, e, s \rangle &\triangleright \langle p, \text{false} : e, s \rangle \\
\langle \text{EQ} : p, z_1 : z_2 : e, s \rangle &\triangleright \begin{cases} \langle p, \text{true} : e, s \rangle & \text{se } z_1 = z_2 \text{ e } z_i \in \mathbb{Z} \\ \langle p, \text{false} : e, s \rangle & \text{caso contrário e } z_i \in \mathbb{Z} \end{cases} \\
\langle \text{LE} : p, z_1 : z_2 : e, s \rangle &\triangleright \begin{cases} \langle p, \text{true} : e, s \rangle & \text{se } z_1 \leq z_2 \text{ e } z_i \in \mathbb{Z} \\ \langle p, \text{false} : e, s \rangle & \text{caso contrário e } z_i \in \mathbb{Z} \end{cases} \\
\langle \text{AND} : p, t_1 : t_2 : e, s \rangle &\triangleright \begin{cases} \langle p, \text{true} : e, s \rangle & \text{se } t_1 = t_2 = \text{true} \\ \langle p, \text{false} : e, s \rangle & \text{caso contrário} \end{cases} \\
\langle \text{NEG} : p, t_1 : e, s \rangle &\triangleright \begin{cases} \langle p, \text{true} : e, s \rangle & \text{se } t_1 = \text{false} \\ \langle p, \text{false} : e, s \rangle & \text{se } t_1 = \text{true} \end{cases} \\
\langle \text{FETCH}(x) : p, e, s \rangle &\triangleright \langle p, s(x) : e, s \rangle \\
\langle \text{STORE}(x) : p, z : e, s \rangle &\triangleright \langle p, e, s[x \mapsto z] \rangle \quad \text{se } z \in \mathbb{Z} \\
\langle \text{NOOP} : p, e, s \rangle &\triangleright \langle p, e, s \rangle \\
\langle \text{BRANCH}(p_1, p_2) : p, t : e, s \rangle &\triangleright \begin{cases} \langle p_1 : p, e, s \rangle & \text{se } t = \text{true} \\ \langle p_2 : p, e, s \rangle & \text{se } t = \text{false} \end{cases} \\
\langle \text{LOOP}(p_1, p_2) : p, e, s \rangle &\triangleright \langle p_1 : \text{BRANCH}(p_2 : \text{LOOP}(p_1, p_2), \text{NOOP}) : p, e, s \rangle
\end{aligned}$$

$$\begin{aligned}
\mathcal{CA}[n] &= \text{PUSH}(n) \\
\mathcal{CA}[x] &= \text{FETCH}(x) \\
\mathcal{CA}[a_1 + a_2] &= \mathcal{CA}[a_2] : \mathcal{CA}[a_1] : \text{ADD} \\
\mathcal{CA}[a_1 * a_2] &= \mathcal{CA}[a_2] : \mathcal{CA}[a_1] : \text{MULT} \\
\mathcal{CA}[a_1 - a_2] &= \mathcal{CA}[a_2] : \mathcal{CA}[a_1] : \text{SUB} \\
\mathcal{CB}[\text{true}] &= \text{TRUE} \\
\mathcal{CB}[\text{false}] &= \text{FALSE} \\
\mathcal{CB}[a_1 = a_2] &= \mathcal{CA}[a_2] : \mathcal{CA}[a_1] : \text{EQ} \\
\mathcal{CB}[a_1 \leq a_2] &= \mathcal{CA}[a_2] : \mathcal{CA}[a_1] : \text{LE} \\
\mathcal{CB}[\neg b] &= \mathcal{CB}[b] : \text{NEG} \\
\mathcal{CB}[b_1 \wedge b_2] &= \mathcal{CB}[b_2] : \mathcal{CB}[b_1] : \text{AND} \\
\mathcal{CC}[x := a] &= \mathcal{CA}[a] : \text{STORE}(x) \\
\mathcal{CC}[\text{skip}] &= \text{NOOP} \\
\mathcal{CC}[c_1; c_2] &= \mathcal{CC}[c_1] : \mathcal{CC}[c_2] \\
\mathcal{CC}[\text{if } b \text{ then } c_1 \text{ else } c_2] &= \mathcal{CB}[b] : \text{BRANCH}(\mathcal{CC}[c_1], \mathcal{CC}[c_2]) \\
\mathcal{CC}[\text{while } b \text{ do } c] &= \text{LOOP}(\mathcal{CB}[b], \mathcal{CC}[c])
\end{aligned}$$