Computabilidade e Teoria da Recursão

Armando Matos
Nelma Moreira

Departamento de Ciência de Computadores
Faculdade de Ciências, Universidade do Porto
email: {acm,nam}@ncc.up.pt

Conteúdo

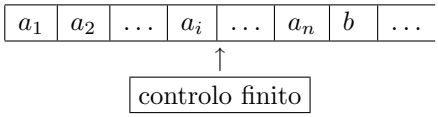
Capítulo 1

Máquinas de Turing

Bibliografia [HU79, Cap. 7], [Pap94, Cap. 2],[LP81, Cap. p4][Dew89, Cap. 28]

Começamos por definir um tipo básico de máquina de Turing. Como veremos, existem muitos outros modelos, todos equivalentes a este.

Uma máquina de Turing é constituída por um controlo finito (conjunto finito de estados), uma fita dividida em células, e uma cabeça de leitura/escrita (RW) que actua sobre uma célula da fita de cada vez. Supomos que a fita tem uma célula mais à esquerda mas é infinita para a direita. No início supomos que as n células mais à esquerda contém a sequência de entrada ("dados") e as restantes o caracter de fita "branco".



Num movimento, dependendo do símbolo lido na fita pela cabeça e do estado do controlo finito, a máquina

1. muda de estado
2. escreve um símbolo na célula que está debaixo da cabeça
3. move a cabeça para a esquerda ou para a direita

Formalmente, uma máquina de Turing (TM) é um tuplo

$$M = (S, \Sigma, \Gamma, \delta, s_0, b, F)$$

onde

S é um conjunto finito de estados

Γ é o conjunto finito de *símbolos da fita*

Σ é um subconjunto de Γ que não incluí b, é o conjunto dos *símbolos de entrada*

δ é a *função movimento seguinte*, função parcial de $S \times \Gamma$ em $S \times \Gamma \times \{l, r\}$

s_0 é o estado inicial

b é um símbolo de Γ, designado por *branco*

$F \subseteq S$ é o conjunto de estados finais

Uma *descrição instantânea* (ID) ou *configuração* é representada por xsy, onde s é o estado corrente, $x \in \Gamma^\star$ é a sequência de símbolos à esquerda da cabeça, sendo o último símbolo de x o que está imediatamente à esquerda da cabeça, e $y \in \Gamma^\star$ é a sequência à direita da cabeça (o primeiro símbolo de y é o que está debaixo da cabeça). Supomos que x e y não contêm "brancos" desnecessários, mas y pode ser b.

A relação $@ >> M >$, um *movimento* (passo), entre duas configurações é definida da seguinte forma. Seja $X_1 \cdots X_{i-1} s X_i \cdots X_n$ uma configuração. Se $\delta(s, X_i) = (s', Y, D)$ e $i = 1$ então $D = r$ obrigatoriamente. Se $i > 1$ e $D = l$ então,

$$X_1 \cdots X_{i-1} s X_i \cdots X_n @ >> M > X_1 \cdots X_{i-2} s' X_{i-1} Y X_{i+1} \cdots X_n$$

Analogamente se $D = r$ tem-se que

$$X_1 \cdots X_{i-1} s X_i \cdots X_n @ >> M > X_1 \cdots X_{i-1} Y s' X_{i+1} \cdots X_n$$

Diz-se que $xsy@ >> M >^k x's'y'$ se $x's'y'$ resulta de xsy em k movimentos e $xsy@ >> M >^\star x's'y'$ se existe $k > 0$ tal que $xsy@ >> M >^k x's'y'$.

A *linguagem aceite por* M, é

$$L(M) = \{x \mid \ x \in \Sigma^\star \text{ e } s_0 x @ >> M >^\star x_1 s x_2 \text{ para algum } s \in F, \text{ e } x_1, x_2 \in \Gamma^\star\}$$

Exemplo 1 *Sendo* $L = \{0^n 1^n \mid \ n \geq 1\}$ *definimos a seguinte TM,*

$$M = (\{s_0, s_1, s_2, s_3, s_4\}, \{0, 1\}, \{0, 1, X, Y, b\}, \delta, s_0, b, \{s_4\})$$

onde,

$$\begin{array}{ll} \delta(s_0, 0) = (s_1, X, r) & \delta(s_0, Y) = (s_3, Y, r) \\ \delta(s_1, 0) = (s_1, 0, r) & \delta(s_1, 1) = (s_2, Y, l) \\ \delta(s_1, Y) = (s_1, Y, r) & \delta(s_2, 0) = (s_2, 0, l) \\ \delta(s_2, X) = (s_0, X, r) & \delta(s_2, Y) = (s_2, Y, l) \\ \delta(s_3, Y) = (s_3, Y, r) & \delta(s_3, b) = (s_4, b, r) \end{array}$$

Exercício 1.0.1 Descreva máquinas de Turing que aceitem cada uma das seguintes linguagens:

a) $\{0^n 1 0^n \mid \ n \geq 1\}$

b) o conjunto de sequências com igual número de 1s e de 0s

c) $L = \{ww^r \mid w \in \{0, 1\}^\star\}$ onde w^r corresponde ao inverso de w

Se $x \in L(M)$ então M pára quando atinge um estado final. Caso contrário, M pode não parar.

Uma linguagem diz-se *recursivamente enumerável* (r.e) se é aceite por uma máquina de Turing. Neste caso, consideramos TMs como *aceitadoras* de linguagens.

Note-se que sendo M uma TM, o problema $x \in L(M)$? não é computável. Se for verdade, sendo x a sequência de entrada (dados) de M, esta pára num número finito de movimentos (passos). Mas se $x \notin L(M)$, M pode não parar e portanto não se obtém resposta.

Designam-se por *recursivas* as linguagens para as quais existe pelo menos uma TM que pára para todos os dados.

Podemos, também, considerar TMs *reconhecedoras* de linguagens, isto é, máquinas

$$M = (S, \Sigma, \Gamma, b, \delta, s_0, b, s_a, s_r)$$

onde há só dois estados finais, um estado de aceitação, s_a e um estado de rejeição, s_r. Neste caso, as máquinas param para quaisquer dados. Como veremos, isto equivale a TM calcular uma função total de $\Sigma^\star$ em $\{0,1\}$.

Exercício 1.0.2 Descreva uma máquina de Turing que determine se um $x \in \{0,1\}^\star$ é um palíndromo.

1.1 Máquinas de Turing calculadoras de funções parciais

Uma máquina de Turing pode ser vista como uma calculadora de funções parciais dos inteiros nos inteiros:

$$f : \mathbb{N}^k \longrightarrow_p \mathbb{N}$$

Suponhamos que os inteiros estão codificados em *unário*. Então, um inteiro $i \geq 0$ é representado por 1^i. Se uma função tiver k argumentos $i_1, \ldots, i_k$, a sequência de entrada pode ser $1^{i_1}0 \ldots 01^{i_k}$. Se a TM pára com a sequência 1^m na fita, então $f(i_1, \ldots, i_k) = m$. Se a TM pára para quaisquer argumentos, então f é uma função recursiva total. Note-se que as funções recursivas totais correspondem a linguagens recursivas.

Exemplo 2 *Seja*

$$m \dot{-} n = \begin{cases} m - n & \text{se } m \geq n \\ 0 & \text{caso contrário} \end{cases}$$

A TM $M = (\{s_0, s_1, \ldots, s_6\}, \{0,1\}, \{0,1,b\}, \delta, s_0, b, \emptyset)$, *onde*

$$\begin{array}{lll} \delta(s_0,1) = (s_1,b,r) & \delta(s_1,1) = (s_1,1,r) & \delta(s_1,0) = (s_2,0,r) \\ \delta(s_2,0) = (s_2,0,r) & \delta(s_2,1) = (s_3,0,l) & \delta(s_3,1) = (s_3,1,l) \\ \delta(s_3,0) = (s_3,0,l) & \delta(s_3,b) = (s_0,b,r) & \delta(s_2,b) = (s_4,b,l) \\ \delta(s_4,0) = (s_4,b,l) & \delta(s_4,1) = (s_4,1,l) & \delta(s_4,b) = (s_6,0,r) \\ \delta(s_0,0) = (s_5,b,r) & \delta(s_5,0) = (s_5,b,r) & \delta(s_5,1) = (s_5,b,r) \\ \delta(s_5,b) = (s_6,b,r) & & \end{array}$$

calcula $1^m01^n@ >> M >^\star 1^{m\dot{-}n}$. A máquina vai sucessivamente apagando um 1 da representação de m e substituindo um 1 por 0 na representação de n, até que uma das seguintes condições ocorre:

- quando procura um 1 na representação de n, não o encontra. Neste caso, $m > n$. Então substitui $n+1$ $0's$ por um 1 e por n símbolos b, deixando $m\dot{-}n$ 1s na fita.
- quando começa um ciclo não encontra nenhum 1. Neste caso, $m < n$. Então apaga toda a fita.

Exercício 1.1.1 *Escreva máquinas de Turing que permitam calcular as seguintes funções, supondo os inteiros codificados em "unário":*

(i) $f(n) = n + 1$

(ii) $f(n) = n^2$

(iii) $f(n, m) = n + m$

Dum modo geral uma TM M calcula uma função parcial f de $\Sigma^\star$ em $\Sigma^\star$ se dada como sequência de entrada $x \in \Sigma^\star$ a configuração final de M é $f(x)sb$, para um estado final s.

1.2 Variações sobre as Máquinas de Turing

Várias modificações da TM básica são possíveis sem que o modelo de computação obtido seja nem mais nem menos poderoso.

Entre elas considerem-se as seguintes restrições/extensões:

1. Alfabeto.
   (a) Codificação de um alfabeto noutro com pelo menos dois símbolos. Dê exemplos e conclua que o alfabeto não reduz a potência.
2. Aridade das funções
   (a) Codificação de pares de inteiros em inteiros. Considere a função bijectiva p de $\mathbf{N}_0 \times \mathbf{N}_0$ em $\mathbf{N}$
   $$p(x,y) = \frac{(x+y)(x+y+1)}{2} + x$$
   (b) Codificações bijectivas de $\mathbf{N}^n$ em $\mathbf{N}$ usando a função p definida atrás.
   (c) Conclusão: As funções de aridade maior que 1 podem ser codificadas em funções de aridade 1.
3. Movimentos da cabeça
   (a) Máquinas com movimentos $\{l, r\}$ (esquerda/direita)
   (b) Máquinas com movimentos $\{l, n, r\}$ (esquerda/não mexe/direita)
4. Tamanho da fita
   (a) Máquinas com uma fita semi-infinita
   (b) Máquinas com uma fita duplamente infinita
5. Número e tipo de fitas
   (a) Máquinas com 1 fita RW
   (b) Máquinas com k fitas RW
   (c) Máquinas com fita de escrita de saída (W) e uma fita RW de trabalho.

Vamos apenas analisar a equivalência dos seguintes modelos: 3a e 3b;4a e4b; 5a e 5b.

Máquinas com movimentos $\{l, n, r\}$ Exceptuando a terceira componente da definição de δ estas máquinas são definidas como as TM básicas.

Se M_1 é uma TM com movimentos $\{l, r\}$ também é uma TM com movimentos $\{l, n, r\}$. Então, suponhamos que $M_2 = (S_2, \Sigma_2, \Gamma_2, \delta_2, s_2, b, F_2)$ é uma máquina com movimentos $\{l, n, r\}$. Vamos construir uma TM apenas com movimentos $\{l, r\}$, M_1, tal que se L é aceite por M_2 então, L é aceite por M_1. Quando a cabeça de M_2 fica parada, M_1 deve entrar num estado especial e movimentar-se uma célula para direita e outra para a esquerda, voltando à mesma célula de tal modo que simule o facto de M_2 não se mexer. Para tal, os estados de M_1 vão ser da forma $[s, X]$ ou $[s, Y]$ para cada estado $s \in S_2$, sendo os estados $[s, X]$ usados sempre que a cabeça de M_2 se movimente e os estados $[s, Y]$ usados no caso da cabeça de M_2 não se mova. Formalmente, tem-se $M_1 = (S_1, \Sigma_1, \Gamma_1, \delta_1, [s_2, X], b, F_1)$ onde

$$S_1 = \{[s, X], [s, Y] \mid s \in S_2\},\ F_1 = F_2 \times \{X, Y\}$$

$\Sigma_1 = \Sigma_2$, $\Gamma_1 = \Gamma_2$

δ_1 define-se por:

1. $\forall \alpha \in \Gamma_1$,

$$\delta_1([s, X], \alpha) = ([s', X], \alpha', r) \qquad \text{se } \delta_2(s, \alpha) = (s', \alpha', r)$$

2. $\forall \alpha \in \Gamma_1$,

$$\delta_1([s, X], \alpha) = ([s', X], \alpha', l) \qquad \text{se } \delta_2(s, \alpha) = (s', \alpha', l)$$

3. $\forall \alpha \in \Gamma_1$,

$$\delta_1([s, X], \alpha) = ([s', Y], \alpha', r) \qquad \text{se } \delta_2(s, \alpha) = (s', \alpha', n)$$

4. $\forall \alpha \in \Gamma_1, \forall s \in S_2$,

$$\delta_1([s, Y], \alpha) = ([s, X], \alpha, l)$$

Máquinas com fita duplamente infinita Uma TM com uma fita duplamente infinita é definida como a TM básica com a excepção de que não existe nenhuma célula mais à esquerda e portanto δ não está limitada nos seus movimentos para a esquerda. A noção de configuração e da relação $@ >> M >$ são definidas análogamente, mas neste caso também se supõe a existência de uma infinidade de "brancos" à esquerda, que podem ser omitidos. No início a sequência de entrada pode começar em qualquer célula.

Exercício 1.2.1 Formalize as noções associadas a uma TM com fita duplamente infinita.

Teorema 1.2.1 *L é uma linguagem aceite por uma TM com uma fita duplamente infinita se e só se é aceite por uma TM com uma fita semi-infinita.*

Dem. ($\Leftarrow$) Uma TM M_2 com uma fita duplamente infinita pode simular facilmente uma TM com uma fita semi-infinita, M_1. Basta que M_2 marque a célula à esquerda da cabeça no início e depois simule a TM M_1.

($\Rightarrow$) Seja $M_2 = (S_2, \Sigma_2, \Gamma_2, \delta_2, s_2, b, F_2)$ uma TM com fita duplamente infinita. Vamos construir M_1 com uma fita semi-infinita para a direita. Podemos imaginar que a fita de M_1 é dividida ao meio em duas pistas, de modo a que cada célula pode conter dois símbolos. Uma pista (S) corresponde às células da fita de M_2 à direita da célula onde inicialmente se encontra a cabeça e a outra (I) às células à esquerda dessa célula. Por exemplo:

Fita de M_2:

$\ldots$	a_{-2}	a_{-1}	a_0	a_1	a_2	$\ldots$

Fita de M_1:

a_0	a_1	a_2	$\ldots$
\$	a_{-1}	$a_{-2},$	$\ldots$

onde a_0 é a célula que inicialmente está debaixo da cabeça de M_2. A primeira célula da fita de M_1 contém o símbolo \$, na "pista" inferior, para indicar que é a célula mais à esquerda. O controlo finito de M_1 indicará se a cabeça de M_2 está debaixo duma célula correspondente à parte superior ou inferior da fita de M_1. Enquanto a cabeça de M_2 estiver à direita da célula inicial, a cabeça de M_1 "trabalhará" na parte superior da sua fita e movimentar-se-á como

a de M_2. Se a cabeça de M_2 estiver à esquerda da célula inicial, a cabeça de M_1 actuará na parte infeior da fita e movimentar-se-á em sentido contrário ao da cabeça de M_2. Note-se que a noção de pistas é apenas conceptual, não havendo nenhuma alteração na definição de TM básica.

Formalmente seja $M_1 = (S_1, \Sigma_1, \Gamma_1, \delta_1, s_1, [b,b], F_1)$ onde,

$$S_1 = \{[s,S],[s,I] \mid s \in S_2\} \cup \{s_1\}$$
$$\Sigma_1 = \{[\alpha, b] \mid \alpha \in \Sigma_2\}$$
$$\Gamma_1 = \{[X,Y] \mid X \in \Gamma_2, Y \in \Gamma_2 \cup \{\$\}\}$$
$$F_1 = \{[s,S],[s,I] \mid s \in F_2\}$$

A função $\delta_1 : S_1 \times \Gamma_1 \longrightarrow_p S_1 \times \Gamma_1 \times \{l, r\}$ é definida por:

1. Para cada $\alpha \in \Sigma_1 \cup \{b\}$,
$$\delta_1(s_1, [\alpha, b]) = ([s,S],[X,\$], r) \qquad \text{se } \delta_2(s_2, \alpha) = (s, X, r)$$
2. Para cada $\alpha \in \Sigma_1 \cup \{b\}$,
$$\delta_1(s_1, [\alpha, b]) = ([s,I],[X,\$], r) \qquad \text{se } \delta_2(s_2, \alpha) = (s, X, l)$$
3. Para cada $[X,Y] \in \Gamma_1$, com $Y \neq \$$ e $m = r$ ou $m = l$,
$$\delta_1([s,S],[X,Y]) = ([s',S],[Z,Y], m) \qquad \text{se } \delta_2(s, X) = (s', Z, m)$$
4. Para cada $[X,Y] \in \Gamma_1$, com $Y \neq \$$ e se $m = r$ então $\bar{m} = l$ e se $m = l$ então $\bar{m} = r$,
$$\delta_1([s,I],[X,Y]) = ([s',I],[X,Z], m) \qquad \text{se } \delta_2(s, Y) = (s', Z, \bar{m})$$
5. $$\delta_1([s,S],[X,\$]) = \delta_1([s,I],[X,\$]) \quad = \quad ([s',P],[Y,\$], r) \quad \text{se } \delta_2(s, X) = (s', Y, m)$$

   onde $P = S$ se $m = r$ e $P = I$ se $m = l$

□

Máquinas com k fitas RW Uma máquina de Turing com k fitas, $k > 1$, consiste num controlo finito com k cabeças, uma para cada uma das k fitas. Vamos considerar que cada fita é duplamente infinita. Num movimento, dependendo do estado do controlo finito e dos símbolos lidos em cada uma das k fitas pelas k cabeças, a máquina

1. muda de estado
2. escreve um símbolo em cada uma das células que estão debaixo de cada cabeça
3. move cada cabeça, independentemente, para a esquerda ou para a direita ou não mexe

Inicialmente a sequência de entrada encontra-se na primeira fita e todas as outras estão em branco. Os restantes conceitos são análogos aos duma TM com uma só fita.

Exercício 1.2.2 Descreva formalmente uma TM com k fitas e as noções de configuração, relação $@ >> M >$ e de linguagem aceite por uma TM de k fitas.

Teorema 1.2.2 *Se uma linguagem L é aceite por uma TM com k fitas, então L é aceite por uma TM com uma única fita.*

Dem. Suponhamos que L é aceite por uma TM com k fitas M_1. Podemos construir uma TM, M_2 com uma só fita e com $2k$ pistas, duas pistas para cada fita. A primeira pista contém o conteúdo da fita correspondente de M_1 e a segunda está em branco excepto para a célula actualmente lida pela cabeça correspondente de M_1. Esta célula contém um indicador, que supomos ser o símbolo X. Por exemplo se $k = 2$, a fita de M_2 podia ser

	b	X	$\ldots$	b	
	a_1	a_2	$\ldots$	a_m	
	b	b	$\ldots$	X	
	b_1	b_2	$\ldots$	b_m	

Formalmente, ter-se-á $\Sigma_2 \subseteq (\{X, b\} \times \Sigma_1)^k$.

O controlo finito de M_2 guardará para além do estado de M_1 o número de indicadores à direita da cabeça de M_2 (n_i). Cada movimento de M_1 é simulado percorrendo a fita de M_2 da esquerda para a direita, para saber quais os símbolos actualmente lidos pelas k cabeças de M_1 e depois da direita para a esquerda, para actualizar esses valores e movimentar convenientemente os "indicadores". Inicialmente supõe-se que a cabeça de M_2 está na célula mais à esquerda que contém um indicador. Então, a cabeça de M_2 percorre a fita para a direita, e para cada célula com indicador "memoriza" o símbolo lido e diminui n_i. Esta memorização pode ser feita acrescentado a M_2 novos estados, cada um correspondendo a uma combinação de um estado de M_1 e a k-símbolos de Γ_1. Quando $n_i = 0$ então M_2 tem informação suficiente para determinar o movimento de M_1. Então, a cabeça de M_2 percorre a fita da esquerda para a direita, e sempre que encontra um indicador modifica o símbolo lido e movimenta o indicador de acordo com o movimento da cabeça correspondente em M_1. Quando já actualizou todos os indicadores, M_2 muda o estado correspondente a M_1. Se o novo estado de M_1 é um estado final, o de M_2 também o é. □

Exercício 1.2.3 Para $k = 2$ formaliza a máquina M_2 construída na demonstração anterior.

Exercício 1.2.4 Estima o número de movimentos efectuados por M_2 para cada movimento de M_1. Analisa a eficiência do uso de TMs com k fitas, $k > 1$, em relação a TMs só com uma fita.

Exercício 1.2.5 Descreve uma TM com 2 fitas que aceite a seguinte linguagem:

$$L = \{ww^r \mid w \in \{0, 1\}^\star\}$$

e compara-a com a descrita no exercício 1.0.1.c)

1.3 Máquinas de Turing não determinísticas

Uma máquina de Turing é *não determinística* se dado um estado e um símbolo lido, existirem várias hipóteses para o movimento seguinte, isto é,

$$\delta \subseteq (S \times \Gamma) \times (S \times \Gamma \times \{l, r\})$$

Uma TM não determinística aceita a sequência de entrada se existe uma sequência de escolhas de movimentos que conduza a um estado de aceitação (análogamente se definem os conceitos de função calculada ou linguagem reconhecida). Em contraste, as TM definidas anteriormente denominam-se *determinísticas*.

Contudo a adição de não determinismo não torna estas TMs um modelo de computação mais poderoso que as TM determinísticas.

Teorema 1.3.1 *Se uma linguagem é aceite por uma máquina de Turing não determinística, M_1, então L é aceite por alguma máquina de Turing determinística, M_2.*

Dem. Para cada estado e símbolo de fita de M_1, existe um número finito de escolhas para o movimento seguinte. Estas escolhas podem ser numeradas 1, 2, ... Seja r o número máximo de escolhas de um par estado-símbolo. Então qualquer sequência finita de escolhas pode ser representado por uma sequência de dígitos de 1 a r.

Vamos construir M_2 com 3 fitas. A primeira contém a sequência de entrada. A segunda, gera sequências de dígitos de 1 a r dum modo sistemático. Por exemplo, por ordem crescente de comprimento e por ordem lexicográfica se de igual comprimento. Para cada sequência gerada na segunda fita, M_2 copia a sequência de entrada para a terceira fita e simula M_1 na terceira fita, usando a sequência da segunda fita para decidir os movimentos de M_1. Se M_1 atinge um estado de aceitação, então M_2 aceita. Se nenhuma sequência de escolhas conduz a M_1 aceitar, M_2 também não aceita. □

Exercício 1.3.1 Formaliza a geração de sequências de dígitos entre 1 e r, usada na demonstração do teorema anterior. Mostra como é que esta sequência pode codificar os movimentos de M_1.

Capítulo 2

Linguagem FOR(ℕ)

Bibliografia [SvW88, Cap. 3], [Wei87, Cap. 1]

Considere a linguagem FOR(ℕ) cuja sintaxe é definida pela seguinte gramática:

$$\begin{array}{rcl} < \texttt{Prog} > & \rightarrow & \{(n, p, < \texttt{Insts} >) \mid \texttt{n} \geq \texttt{0},\ \texttt{p} \geq \texttt{n} + \texttt{1}\} \\ < \texttt{Insts} > & \rightarrow & < \texttt{Inst} >^{\star} \\ < \texttt{Inst} > & \rightarrow & < \texttt{Var} > ++ \\ & \rightarrow & \mid\ < \texttt{Var} > -- \\ & \rightarrow & \mid \texttt{for}\ < \texttt{Var} >\ \{ < \texttt{Insts} > \} \end{array}$$

Onde,

- considera-se que $\texttt{0} - - = \texttt{0}$.
- supõe-se que os valores das variáveis são inteiros não negativos.
- uma <Var> é uma letra eventualmente seguida de 1 ou mais letras ou símbolos.
- inicialmente todas as variáveis contêm 0 excepto as que correspondem aos valores dos dados. Estas são designadas por x1, x2,... xn. O número total de variáveis dum programa é p.
- a variável que define o resultado é y.
- o "<Insts>" dentro de uma instrução "for" é executado um número de vezes que é igual ao valor *inicialmente* contido na variável.

O parâmetro ℕ pode ser omitido supondo-se que o valor das variáveis pertence a ℕ.
Um programa em FOR com n variáveis de entrada, calcula uma função $f : \mathbb{N}^n \longrightarrow \mathbb{N}$.

Exercício 2.0.2 Mostre que as seguintes funções podem ser implementadas em linguagem FOR.

1. Para cada valor k a função constante $f(x) = k$
2. A função $x1 + x2$.
3. A função $x1 - x2$ definindo-se $x1 - x2 = 0$ se $x2 > x1$.
4. A função $x1 * x2$.
5. A função $x1^{x2}$.

Exercício 2.0.3 Escreva um programa que implemente a instrução condicional genérica

`if` $<$ `Var` $>$ `then` $<$ `Insts` $>$ `else` $<$ `Insts` $>$

supondo que o primeira sequência de instruções é executada se o valor da variável é diferente de 0.

Para formalizar a semântica operacional dum programa em `FOR` considere-se que cada programa é formado por um conjunto de linhas e cada linha contém:

- uma instrução de incremento ou decremento
- `for` $<$ `Var` $>$ {
- }

Supõe-se ainda que uma linha com o início duma instrução `for` tem à direita o número da linha da instrução que *fecha* o ciclo e que essa instrução tem ao número da linha do início do ciclo. Por exemplo:

```
1:  for x1{   :3
2:  y++
3:  }         :1
```

Definição 2.0.1 *Seja $Q = (n, p, P)$ um programa em* `FOR`*, com linhas numeradas de 1 a t. Um* estado *de Q é um tuplo (l, x), $1 \leq l \leq t+1$ e $x \in \mathbb{N}^p$. Para cada $1 \leq i \leq p$ x_i representa o valor da variável xi. O* estado *é inicial se $l = 1$ e $x_{n+1} = \ldots = x_p = 0$. O estado é final se $l = t+1$.*

Definição 2.0.2 *Seja $Q = (n, p, P)$ um programa em* `FOR`*, com linhas numeradas de 1 a t. Seja S_Q o conjunto de todos os seus estados. Um estado $s_1 = (l, x)$ é transformado num passo de computação num estado $s_2 = (l', x')$, e denota-se $s_1 @ >> Q > s_2$ se e só se:*

1. *l : `xj --` ou l : `xj ++`, $l' = l + 1$, $x'_i = x_i$ para todos $1 \leq i \leq p$ e $i \neq j$ sendo $x'_j = x_j - 1$ ou $x'_j = x_j + 1$, respectivamente*
2. *l : `for xj {` $: l_1$, então $l' = l + 1$ se $x_j \neq 0$ e $l' = l_1 + 1$ caso contrário. E $x' = x$.*
3. *l :} : l_1 então $l' = l_1$ e $x'_j = x_j - 1$ sendo x_j a variável de ciclo da linha l_1*

Definem-se como habitualmente $@ >> Q >^+$ e $@ >> Q >^\star$.

Definição 2.0.3 *Seja $Q = (n, p, P)$ um programa em* `FOR`*, com linhas numeradas de 1 a t. Seja S_Q o conjunto de todos os seus estados. A função* estado seguinte *$F_Q : \mathbb{N}^{p+1} \longrightarrow \mathbb{N}^{p+1}$ é definida por:*

$$F_Q(s) = \begin{cases} s_1 & \text{se } s@ >> Q > s_1 \\ s & \text{se } s \notin S_Q \text{ ou não existe tal } s_1 \end{cases}$$

Finalmente,

Definição 2.0.4 *Seja $Q = (n, p, P)$ um programa em* `FOR`*, com linhas numeradas de 1 a t. A semântica de Q é uma função $M_Q : \mathbb{N}^p \longrightarrow \mathbb{N}^p$ tal que $M_Q(x) = y$ se existe $(1, x)@ >> Q >^\star (l, y)$ e $l = t + 1$, senão M_Q não está definida.*

Exercício 2.0.4 Defina uma expressão para a função calculada por um programa Q, usando a função M_Q.

Exercício 2.0.5 Mostre que qualquer programa Q em `FOR` termina sempre a sua execução, isto é que M_Q é uma função total.

Exercício 2.0.6 Mostre que, se $f(x)$ e $g(x)$ são implementáveis em `FOR` também o são as seguintes funções.

1. $f(x) + g(x)$, $f(x) * g(x)$ e $f(x)^{g(x)}$.
2. $g(f(x))$, a função composta.

Exercício 2.0.7 Considere de novo o programa que escreveu para o cálculo de $x1^{x2}$. Supondo que cada incremento ou decremento (isto é, instruções da forma `x++` ou `x--`) demora 1 nanosegundo a ser executado e, não contabilizando os restantes testes, etc., calcule o tempo (em unidades apropriadas ao seu tamanho) que demoraria a calcular 100^{100}.

Capítulo 3

Funções Recursivas

Bibliografia [Rog67], [Phi92], [LP81, Cap. 5], [Yas71, Cap. 4, 6]

Vamos estudar as funções de $\mathbb{N}^n$ em $\mathbb{N}$.

3.1 Funções Primitivas Recursivas

Tentativa de englobar todas as funções totais num mesmo método de definição.

Definição 3.1.1 *1. Funções de base:*

- Função zero: $0 : \mathbb{N} \to \mathbb{N}$ *definida por*

$$0(x) = 0 \text{ para todo o } x \in \mathbb{N}$$

- Função sucessor: $s : \mathbb{N} \to \mathbb{N}$ *definida por*

$$s(x) = x + 1 \text{ para todo o } x \in \mathbb{N}$$

- Funções projecção: $\pi^n_m : \mathbb{N}^n \to \mathbb{N}$ *com* $n \geq m \geq 1$*) definida por*

$$\pi^n_m(x_1, \cdots, x_n) = x_m$$

2. Regras para formação de funções primitivas recursivas.

- Composição: *Dadas as* $n + 1$ *funções* $f : \mathbb{N}^n \to \mathbb{N}$ *e*

$$g_i : \mathbb{N}^m \to \mathbb{N} \text{ para } i = 1, \cdots, n$$

definimos a função composta $h : \mathbb{N}^m \to \mathbb{N}$ *por*

$$h(\mathbf{x}) = f(g_1(\mathbf{x}), \cdots, g_n(\mathbf{x}))$$

onde $\mathbf{x}$ *designa o "vector" de variáveis* $x_1, \cdots, x_m$*. A função* h *pode ser mais explicitamente designada por* $comp^n_m(f, g_1, \cdots, g_n)$*.*

- Recursão primitiva: *Dadas as duas funções com* $n \geq 1$

$$f : \mathbb{N}^n \to \mathbb{N}$$
$$g : \mathbb{N}^{n+2} \to \mathbb{N}$$

definimos por recursão primitiva a função $h : \mathbb{N}^{n+1} \to \mathbb{N}$ *por (definição indutiva)*

$$\begin{cases} h(\mathbf{x}, 0) = f(\mathbf{x}) \\ h(\mathbf{x}, y+1) = g(\mathbf{x}, y, h(\mathbf{x}, y)) \end{cases}$$

A função h *pode ser mais explicitamente designada por* $prim^n(f, g)$.

A classe das funções primitivas recursivas (PR) é o menor conjunto de funções de $\mathbb{N}^n$ *em* $\mathbb{N}$ *que inclui as funções de base e que é fechado relativamente à composição e à recursão primitiva.*

É fácil verificar que todas as funções desta classse são totais.

Exercício 3.1.1 Mostra que as seguintes funções são primitivas recursivas. Dá uma justificação detalhada.

1. $f(x) = x$, a função identidade.
2. $f(x) = s(s(x))$.
3. $d(x) = 2$, função constante 2.
4. $f(x, y) = g(y, x)$ supondo que g é PR.
   Sugestão: Utiliza a composição.
5. $f(x, y) = x + y$.
6. $f(x, y) = x \times y$.
7. $f(x, y) = x^y$.
8. A função *não zero*, $sg(0) = 0$, $sg(x) = 1$ para $x \geq 1$.
   Sugestão: $sg(x + 1) = 1$.
9. A função *teste zero*, $\overline{sg}(0) = 1$, $\overline{sg}(x) = 0$ para $x \geq 1$.
10. A função $f(x, y)$ cujo valor é igual a $g(x)$

    $$f(x, y) = g(x) \text{ para todo o } x \text{ e } y$$

    supondo que g é PR.
11. A função assim definida

    $$f(x, y) = \begin{cases} f(x) \text{ se } y = 0 \\ g(x) \text{ se } y \neq 0 \end{cases}$$

    supondo que f e g são PR.
    Sugestão: Utiliza o resultado da alínea anterior e a recursão primitiva.
12. A função assim definida

    $$f(x, y) = \begin{cases} g(x) \text{ se } x \geq y \\ h(x) \text{ se } x < y \end{cases}$$

    supondo que f e g são PR.
    Sugestão: Utiliza o resultado da alínea anterior e a função $f'(x, y) = f(x, x - y + 1)$

13. A função assim definida
$$f(x,y) = \begin{cases} g(x) \text{ se } x = y \\ h(x) \text{ se } x \neq y \end{cases}$$
supondo que f e g são PR.
Sugestão: Utiliza a função $f'(x,y) = f(x,(x-y)+(y-x))$

14. $f(x,y) : \mathbf{N}^2 \to \mathbf{N}$ sendo $f(0,y) = g(y)$ onde g é primitiva recursiva e $f(1,y) = h(y)$ onde h é primitiva recursiva. O valor de $f(x,y)$ para os restantes valores de x deve ser escolhido por forma que f seja primitiva recursiva. Sugestão: considere a função $x\,h(x) + (1-x)f(x)$

15. $f(x,y) = y - 1$, antecessor de y com um argumento extra x. Define-se $f(x,0) = 0$ para todo o x.

16. $f(x) = x - 1$, antecessor de y, definindo-se $f(0) = 0$.
Sugestão: Utiliza a alínea anterior e a composição de funções.

17. $f(x,y) = x - y$, definindo-se $x - y = 0$ para $y \geq x$.
Sugestão: Utiliza a alínea anterior.

18. $f(x,y) = x \,\mathtt{mod}\, y$ (resto da divisão inteira, definindo $a\mathbf{mod}0 = 0$ para todo o a).
Sugestão: Utiliza o resultado da alínea anterior a recursão primitiva e a composição.

19. $f(x,y) = x/y$ (divisão inteira, definindo $a/0 = 0$ para todo o a).
Sugestão: Utiliza o resultado da alínea anterior.

Exercício 3.1.2 Sendo $f : \mathbb{N} \longrightarrow \mathbb{N}$ define-se $f^\star : \mathbb{N}^2 \longrightarrow \mathbb{N}$, *exponenciação* de f, por:
$$f^\star(\mathbf{x}, y) = \underbrace{f(f(\cdots f(\mathbf{x})\cdots))}_{y\text{vezes}}$$
Mostra que se f for PR então $f^\star$ é PR.

Exercício 3.1.3 (i) Mostra que a classe `FOR` de funções de $\mathbf{N}^n$ em $\mathbf{N}$ contém a classe PR. Sugestão: Usa indução no número de regras utilizadas para a definição de uma função PR.

(ii) [$\star$] Mostra que para qualquer programa em `FOR` $Q = (n,p,P)$, M_Q corresponde a uma função primitiva recursiva. Concluí que todas as funções calculadas por um programa em *FOR* são PR. Sugestão: Define $T_Q(x)$ como o número de estados necessários para terminar uma computação que se inicie no estado $(1,x)$.

Exercício 3.1.4 Seja `P` um programa `FOR` que implementa uma função primitiva recursiva. Mostra que o programa `FOR`
$$\mathtt{for}\ v\ \{\mathtt{P}\}$$
onde v é uma variável que não ocorre em `P` (esta restrição não é importante) também implementa uma função primitiva recursiva. Sugestão: Considera uma função com mais um argumento (v) definida por recursão primitiva em v.

3.1.1 Função de Ackermann

[eAP95]

A função de Ackermann é uma função computável que não é primitiva recursiva.

Definição 3.1.2

$$\begin{aligned} A(0,y) &= y+1 \\ A(x+1,0) &= A(x,1) \\ A(x+1,y+1) &= A(x,A(x+1,y)) \end{aligned}$$

Exercício 3.1.5 (i) Mostra que $A(x,y)$ está bem definida.

(ii) Calcula $A(1,2)$, $A(1,3)$, $A(5,3)$. Podes fazer um programa para este fim.

(iii) Mostra que $A(n,m)$ é uma função total. Sugestão: Considera o seguinte predicado

$$P(x,y) \quad sse \quad \begin{array}{l} A(x',y') \text{ termina para todo } (x',y') \text{ tal que} \\ x' < x \text{ ou } x' = x \text{ e } y' < y \end{array}$$

e prova $\forall\, x,\, y \in \mathbb{N},\ P(x,y)$ por dupla indução.

A classe das funções primitivas recursivas é muito vasta e inclui quase todas as funções de interesse prático. Mas, ao contrário daquilo que se pretendia não inclui todas as funções totais.

Pode demonstrar-se, por exemplo, que a função de Ackermann não é primitiva recursiva. Mais concretamente, existe um n tal que para todo o x é $f(x) < A(n,x)$. Nenhuma função primitiva recursiva cresce tão rapidamente como a função de Ackermann!

Exercício 3.1.6 Mostra que existem funções computáveis totais que não pertencem à classe PR.

1. Demonstrando que dada uma qualquer classe enumerável de funções totais computáveis é sempre possível definir uma função total computável não pertencente a essa classe.
2. Demonstrando que a função de Ackermann não é primitiva recursiva.

3.2 Funções Recursivas

O que falta às funções primitivas recursivas? Todas as funções são totais (análogo aos programas `FOR`). Vamos introduzir a *parcialidade* das funções. Se f é uma função parcial, designámos $f(x)\downarrow$ se f está definida para x e por $f(x)\uparrow$ caso contrário.

Definição 3.2.1 (Operador de minimização) *Seja* $f : \mathbb{N}^{n+1} \longrightarrow_p \mathbb{N}$. *Define-se* $g : \mathbb{N}^n \longrightarrow_p \mathbb{N}$ *por* minimização *como:*

$$g(\mathbf{x}) = \textit{menor } y \textit{ tal que } f(\mathbf{x},y) = 0$$

Designa-se $g = \mu^n(f)$.

Dado que f pode não estar definido para alguns valores, tem-se que:

$$\begin{array}{rl} \mu^n(f)(\mathbf{x}) = y & \text{se } f(\mathbf{x},y) = 0 \text{ e} \\ & \forall z,\, 0 \le z < y,\, f(\mathbf{x},z)\downarrow \text{ e } f(\mathbf{x},z) \neq 0 \\ \mu^n(f)(\mathbf{x})\uparrow & \text{caso tal } y \text{ não exista} \end{array}$$

Definição 3.2.2 *A classe das funções (parciais) recursivas (R) é o menor conjunto de funções de $\mathbb{N}^n$ em $\mathbb{N}$ que inclui as funções de base e que é fechado relativamente à composição, à recursão primitiva e à minimização. Uma função é total recursiva se pertence a R e é total.*

Exercício 3.2.1 Mostra que as seguintes funções são recursivas (R). Podes usar os resultados dos problemas anteriores.

1. $f(x, y) =$ menor y (se existir) tal que $f(x, y) = 2$ onde f é R.
2. $f(x, y) =$ menor y (se existir) tal que $f(x, y) = 0$ ou $g(x, y) = 0$ onde f e g são PR. Sugestão: Considera $\mu^1(f(x, y) \times g(x, y))$

Exercício 3.2.2 Mostra que as seguintes funções são recursivas.

(i) Qualquer função primitiva recursiva.

(ii) A função não definida em nenhum ponto, $\uparrow: \mathbf{N} \to \mathbf{N}$.

(iii) A função $f(x)$ não definida para $x \leq 4$ e com valor 0 nos restantes pontos.

Exercício 3.2.3 Considera uma operação de minimização cuja única diferença para μ^n é a seguinte: em vez de se testar $f(\overline{x}, y) = 0$ testa-se $f(\overline{x}, y) = 2$. Mostra que esta operação é equivalente a μ.

Capítulo 4

Linguagem WHILE(ℕ)

Bibliografia [SvW88, Cap. 3, 4], [Wei87, Cap. 1]

Considere a linguagem WHILE(ℕ) cuja sintaxe é definida pela seguinte gramática:

$$
\begin{array}{lcl}
< \texttt{Prog} > & \rightarrow & \{(n, p, < \texttt{Insts} >) \mid \texttt{n} \geq \texttt{0},\ \texttt{p} \geq \texttt{n} + \texttt{1}\} \\
< \texttt{Insts} > & \rightarrow & < \texttt{Inst} >^{\star} \\
< \texttt{Inst} > & \rightarrow & < \texttt{Var} > \texttt{++} \\
 & & \mid < \texttt{Var} > \texttt{--} \\
 & & \mid \texttt{if } < \texttt{Var} > \texttt{ then } < \texttt{Insts} > \texttt{ else } < \texttt{Insts} > \\
 & & \mid \texttt{while } < \texttt{Var} > \texttt{ \{ } < \texttt{Insts} > \texttt{ \}}
\end{array}
$$

Onde,

- considera-se que $\texttt{0} - - = \texttt{0}$.
- supõe-se que os valores das variáveis (registos) são inteiros não negativos.
- uma $<\texttt{Var}>$ é da forma $\texttt{x0}, \texttt{x1}, \texttt{x2}, \ldots$
- inicialmente todas as variáveis (registos) contêm 0 excepto as que correspondem aos valores dos dados. Estas são designadas por x1, x2,... xn. O número total de variáveis do programa é p.
- a variável (registo) que define o resultado é $\texttt{x}_0$.
- o "$<\texttt{Insts}>$" dentro de uma instrução "while" é executado enquanto o valor da variável $<\texttt{Var}>$ for diferente de 0.

O parâmetro ℕ pode ser omitido supondo-se que o valor das variáveis pertence a ℕ.

Um programa em WHILE com n variáveis de entrada, calcula uma função $f : \mathbb{N}^n \longrightarrow \mathbb{N}$. Designámos essa classe de funções como a classe WHILE.

Para formalizar a semântica operacional dum programa em WHILE considere-se que cada programa é formado por um conjunto de linhas. A instrução condicional vai ser omitida, pois pode ser implementada com as restantes (Verifica!). Cada linha contém:

- uma instrução de incremento ou decremento
- while $<$ Var $>$ {
- }

Supõe-se ainda que uma linha com o início duma instrução `while` tem à direita o número da linha da instrução que *fecha* o ciclo e que essa instrução, por sua vez, tem à direita o número da linha do início do ciclo. Por exemplo:

```
1:  while x1{   :3
2:  x1--
3:  }           :1
```

Definição 4.0.3 *Seja $Q = (n, p, P)$ um programa em* `WHILE`*, com linhas numeradas de* 1 *a* t*. Um* estado *de Q é um tuplo (l, x), $1 \leq l \leq t + 1$ e $x \in \mathbb{N}^p$. Para cada $0 \leq i \leq p - 1$ x_i representa o valor da variável xi. O* estado *é inicial se $l = 1$ e $x_0 = x_{n+1} = \ldots = x_{p-1} = 0$. O estado é final se $l = t + 1$.*

Definição 4.0.4 *Seja $Q = (n, p, P)$ um programa em* `WHILE`*, com linhas numeradas de* 1 *a* t*. Seja S_Q o conjunto de todos os seus estados. Um estado $s_1 = (l, x)$ é transformado num passo de computação num estado $s_2 = (l', x')$, e denota-se $s_1@ >> Q > s_2$ se e só se:*

1. *l : `xj --` ou l : `xj ++`, $l' = l + 1$, $x'_i = x_i$ para todos $1 \leq i \leq p$ e $i \neq j$ sendo $x'_j = x_j - 1$ ou $x'_j = x_j + 1$, respectivamente*
2. *nos restantes casos $x' = x$ e se*
   (a) *l : `while xj {` : l_1, então $l' = l + 1$ se $x_j \neq 0$ e $l' = l_1 + 1$ caso contrário.*
   (b) *l :`}` : l_1 e l_1 : `while xj {` : l ocorre em Q (e $l_1 < l$), então $l' = l_1 + 1$ se $x_j \neq 0$ e $l' = l + 1$ caso contrário.*

Definem-se como habitualmente $@ >> Q >^+$ e $@ >> Q >^\star$.

Definição 4.0.5 *Seja $Q = (n, p, P)$ um programa em* `WHILE`*, com linhas numeradas de* 1 *a* t*. Seja S_Q o conjunto de todos os seus estados. A função* estado seguinte *$F_Q : \mathbb{N}^{p+1} \longrightarrow \mathbb{N}^{p+1}$ é definida por:*

$$F_Q(s) = \begin{cases} s_1 & \text{se } s@ >> Q > s_1 \\ s & \text{se } s \notin S_Q \text{ ou não existe tal } s_1 \end{cases}$$

Uma computação dum programa Q pode ser identificada com a a sequência de estados $s_0, s_1, \ldots$ tal que $F_Q(s_i) = s_{i+1}$ $\forall\, i \geq 0$. A computação termina no estado s_f, se $f = min\{j \mid F_Q(s_j) = s_j\}$. O comprimento da computação é definido por $T_Q(x) = f$, e é designado por complexidade temporal.

Definição 4.0.6 *Seja $Q = (n, p, P)$ um programa em* `WHILE`*, com linhas numeradas de* 1 *a* t*. A semântica de Q é uma função $M_Q : \mathbb{N}^p \longrightarrow \mathbb{N}^p$ tal que $M_Q(x) = y$ se existe $(1, x)@ >> Q >^\star (t+1, y)$, senão $M_Q(x) \uparrow$ (não está definida). A função calculada por Q é $f_Q : \mathbb{N}^n \longrightarrow \mathbb{N}$ definida por $f_Q(x) = \pi_1^p(M_Q(x))$.*

Teorema 4.0.1 *A classe de funções* `WHILE` *é equivalente à classe R.*

Dem. ($\Leftarrow$) Prova-se por indução na estrutura de R.

($\Rightarrow$) Seja $Q = (n, p, P)$ um programa `WHILE`. Pretende-se provar que $f_Q \in R$. Em primeiro lugar é necessário codificar os uplos de inteiros (estados) em inteiros, de tal modo que uma função recursiva nos números codificados corresponda à função F_Q. Iremos considerar a codificação de "Cantor" de $\mathbb{N}^k$ em $\mathbb{N}$, p^k e as correspondentes funções de projecção (que são PR).

Para facilitar a exposição, considerem-se as funções:

$$\begin{array}{rll} \iota : \mathbb{N} \longrightarrow \mathbb{N}^{p+1} & \text{tal que} & \iota < l, x_0, x_1, \ldots, x_{p-1} >= (l, x_0, x_1, \ldots, x_{p-1} \\ \lambda : \mathbb{N} \longrightarrow \mathbb{N} & \text{tal que} & \lambda < l, x_0, x_1, \ldots, x_{p-1} >=< x_0, x_1, \ldots, x_{p-1} > \\ \tau : \mathbb{N} \longrightarrow \mathbb{N}^{p} & \text{tal que} & \tau < x_0, x_1, \ldots, x_{p-1} >= (x_0, x_1, \ldots, x_{p-1}) \\ \sigma : \mathbb{N}^{n} \longrightarrow \mathbb{N} & \text{tal que} & \sigma(x_1, \ldots, x_n) =< 0, 0, x_1, \ldots, x_n, 0, \ldots, 0 > \\ & & \text{e } (0, 0, x_1, \ldots, x_n, 0, \ldots, 0) \in \mathbb{N}^{p+1} \end{array}$$

Vamos definir uma função $sf : \mathbb{N} \longrightarrow \mathbb{N}$ tal que se $\iota(z) = s$ então

$$\iota(sf(z)) = F_Q(\iota(z)) \tag{4.1}$$

Para Q, com linhas numeradas de 1 a t, a função F_Q pode ser escrita:

$$F_Q(l, x_0, \ldots, x_{p-1}) = \begin{cases} (l, x_0, \ldots, x_{p-1}) & \text{se } l = 0 \\ F_1(l, x_0, \ldots, x_{p-1}) & \text{se } l = 1 \\ \vdots & \\ F_t(l, x_0, \ldots, x_{p-1}) & \text{se } l = t \\ (l, x_0, \ldots, x_{p-1}) & \text{se } l > t \end{cases}$$

onde, F_i é a transformação (passo de computação) correspondente à linha i do programa, de acordo com a definição de $@ >> Q >$. Sejam $f_i < l, x_0, \ldots, x_{p-1} >=< l', x'_0, \ldots, x'_{p-1} >$ as funções corresponentes a cada F_i. Define-se para $z =< l, x_0, \ldots, x_{p-1} >$

$$sf(z) = z.\overline{sg}(l) + f_1(z).\overline{sg}(|l-1|) + \ldots + f_t(z).\overline{sg}(|l-t|) + z.\overline{sg}(t+1-l)$$

onde $\overline{sg}(0) = 1$, $\overline{sg}(x) = 0$ para $x \geq 1$. Se $l = 1$ então $sf(z) = f_1$, ..., se $l > t$ então $sf(z) = z$. É fácil verificar que sf é PR e satisfaz (4.1). E, também sf^y é PR , $\forall y \geq 0$. Seja, $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$ tal que $g(z, 0) = z$ e $g(z, y+1) = sf(g(z, y))$. Então, g é PR e $g(z, y) = sf^y(z)$. Defina-se $h : \mathbb{N}^2 \longrightarrow \mathbb{N}$ por:

$$h(z, y) = t + 1 - \pi_1^{p+1}(g(z, y))$$

Então, $T_Q(\tau(\lambda(z))) = \mu^2(h)(z)$, se tal valor existe. Finalmente,

$$M_Q(\tau(\lambda(z)) = \tau(g(z, \mu^2(h)(z)))$$

se z corresponder ao estado inicial isto é $z =< 0, 0, x_1, \ldots, x_n, 0, \ldots, 0 >$ e se $x \in \mathbb{N}^n$,

$$f_Q(x) = \pi_1^p(\tau(g(\sigma(x), \mu^2(h)(\sigma(x)))))$$

e portanto $f_Q \in R$. □

Exercício 4.0.4 Mostra cuidadosamente que sf é PR.

Exercício 4.0.5 Supõe que o programa `P` implementa uma função recursiva com resultado em `x0` e dados em `x1` e `x2`. Mostra que o seguinte programa também implementa uma função recursiva (com resultado em `x0` e dados em `x1`)

```
P
x2 <- 0;
while x0 {
   P
   x2 <- x2+1;
}
x0 <- x2;
```

Exercício 4.0.6 (i) Mostra que a classe `WHILE` de funções de $\mathbf{N}^n$ em $\mathbf{N}$ contém a classe R. Sugestão: Usa indução no número de regras utilizadas para a definição de uma função R.

(ii) [$\star$] Mostra que a classe R contém a classe `WHILE`. Define uma semântica M_Q para um programa Q em `WHILE` análoga à definida para programas `FOR`. Define uma função $T_Q(x)$ que seja igual a l, o menor número de estados s_l tal que $F_Q(s_l) = s_l$ se tal valor existir sendo $s_0 = (1, x)$

Exercício 4.0.7 [$\star$] Mostra que para qualquer programa P escrito em `WHILE` existe um programa P1 equivalente a P que tem no máximo um ciclo `while`. Sugestão: Utiliza a indução na estrutura de P. É possível guardar em variáveis adicionais de P1 informação sobre o estado de execução dos "whiles" internos em P (que, pela hipótese indutiva não têm outros "whiles" dentro deles) e executar (em P1) um só ciclo externo.

Nota Este resultado pode traduzir-se em termos das funções recursivas no facto de só ser preciso aplicar uma vez a minimização. Está relacionado com o teorema da forma normal de Kleene.

Capítulo 5

Predicados recursivos

Bibliografia [Min74], [LP81, Cap. 5], [Hof79], [Yas71, Cap. 3,7], [HU79, Cap. 8]

Definição 5.0.7 *Dado um predicado (relação) $P \subseteq \mathbb{N}^k$, a função característica de P define-se por*[1]*:*

$$ch_P(\mathbf{x}) = \begin{cases} 1 & \text{se } P(\mathbf{x}), \text{ isto é, se } P \text{ é verdade para } \mathbf{x} \\ 0 & \text{se não } P(\mathbf{x}) \end{cases}$$

Se ch_P é recursiva (primitiva recursiva) P diz-se recursivo (primitivo recursivo). Se ch_P é total, P diz-se decidível.

Exercício 5.0.8 Mostre que as seguintes funções são PR:

(i) A função *não zero*, $sg(0) = 0$, $sg(x) = 1$ para $x \geq 1$.
Sugestão: $sg(x+1) = 1$.

(ii) A função *zero*, $\overline{sg}(0) = 1$, $\overline{sg}(x) = 0$ para $x \geq 1$. Sugestão: $\overline{sg}(0) = s(0); \overline{sg}(x+1) = 0(\cdots)$.

Exemplo 3 *A relação $=$ é PR, dado que a sua função característica é*

$$ch_=(x,y) = 1 - sg((x-y) + (y-x))$$

Análogamente, a relação $<$ é PR porque $ch_<(x,y) = \overline{sg}(y-x)$.

Lema 5.0.1 *Se Q e P são predicados PR também são PR os predicados $\neg P$, $Q \wedge P$ e $Q \vee P$.*

Exercício 5.0.9 Prova o lema anterior, mostrando que dados ch_P e ch_Q então $ch_{\neg P} = 1 - ch_P$, $ch_{Q \wedge P} = ch_Q \cdot ch_P$ e $ch_{Q \vee P} = sg(ch_Q + ch_P)$ e são PR se ch_P e ch_Q o forem.

Exercício 5.0.10 Mostra que se $g_1 \ldots g_m$ são $k+1$-árias e PR, então

$$f(\mathbf{x}, m) = \Pi_{i=1}^{m} g(\mathbf{x}, i) = g(\mathbf{x}, 0) \cdot \cdots \cdot g(\mathbf{x}, m)$$

também o é. Sugestão: $f(\mathbf{x}, 0) = g(\mathbf{x}, 0)$ e $f(\mathbf{x}, m+1) = f(\mathbf{x}, m) \cdot g(\mathbf{x}, m+1)$

Definição 5.0.8 *Dado um predicado $Q \subseteq \mathbb{N}^k$ a quantificação existencial limitada de Q é qualquer predicado da forma $\exists i \leq m\, Q(\mathbf{x}m)$, e $\mathbf{x} \in \mathbb{N}^{k-1}$. Analogamente se define quantificação universal limitada.*

[1] Esta definição está de acordo com associar 1 ao valor lógico verdade

Lema 5.0.2 *Se Q é um predicado PR, então também o são as suas quantificações universais e existenciais.*

Exercício 5.0.11 Mostra o lema anterior. Sugestão: Use o exercício anterior.

Exercício 5.0.12 Mostra que se $g_1 \dots g_l$ são funções k-árias e PR, e $P_1 \dots P_l$ são predicados PR tal que para cada $\mathbf{x} \in \mathbb{N}^k$ exactamente um e só um $P_i(\mathbf{x})$ é verdade, então a função f definida por:

$$f(\mathbf{x}) = \begin{cases} g_1(\mathbf{x}) & se\ \ P_1(\mathbf{x}) \\ \vdots & \\ g_l(\mathbf{x}) & se\ \ P_l(\mathbf{x} \end{cases}$$

também é PR. Sugestão: Usa as funções ch_{P_i}.

Definição 5.0.9 *Dado $k \geq 0$ e Q um predicado $k+1$-ário, a minimização limitada de Q é uma função $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ tal que:*

$$f(\mathbf{x}, m) = \begin{cases} \textit{menor } p,\ 0 \leq p \leq m \textit{ tal que } Q(\mathbf{x}m) & \textit{se existir} \\ 0 & \textit{caso contrário} \end{cases}$$

Escreve-se $f(\mathbf{x}, m) = \mu p \leq m[Q(\mathbf{x}p)]$.

Lema 5.0.3 *Se Q é um predicado PR, então a sua minimização limitada também o é.*

Exercício 5.0.13 Prova o lema anterior. Sugestão: Considera a função h:

$$h(\mathbf{x}, m, p) = \begin{cases} p & \text{se } \exists i \leq m[Q(\mathbf{x}i)] \\ m+1 & \text{se } Q(\mathbf{x}m+1) \text{ e é falso } \exists i \leq m[Q(\mathbf{x}i)] \\ 0 & \text{é falso } \exists i \leq m[Q(\mathbf{x}i])] \end{cases}$$

Mostra que h é PR e f pode ser obtida de h por recursão primitiva.

Capítulo 6

Codificações e Numerações de Gödel

Para codificar k-tuplos de números inteiros em inteiros pode-se usar as funções $p^k : \mathbb{N}^k \longrightarrow \mathbb{N}$ (Cantor):

$$\begin{aligned} p^1(x) &= x \\ p^{k+1}(x_1, \ldots, x_{k+1}) &= p(x_1, p^k(x_2, \ldots, x_{k+1})) \end{aligned}$$

onde $p(x, y) = \frac{(x+y)(x+y+1)}{2} + x$

Lema 6.0.4 *Para todo o k:*

1. *p^k é bijectiva.*
2. *p^k é PR*
3. *$\forall i,\ 1 \leq i \leq k,\ p_i^k = \pi_i^k (p^k)^{-1}$*

Escreve-se $p^k(x_1, \ldots, x_k) =< x_1, \ldots, x_k >$ e $(< x_1, \ldots, x_k >)_i = x_i$.

Exercício 6.0.14 Prova o lema 6.0.4. Sugestão: Nota que $p^{-1}(z) = (u(z), d(z) - u(z))$ onde $d(z) = (\mu y \leq z[p(0, y) > z]) - 1$ e $u(z) = z - p(0, d(z))$.

Usando esta codificação uma função $f : \mathbb{N}^n \longrightarrow \mathbb{N}^k$ pode ser codificada como uma função $h : \mathbb{N} \longrightarrow \mathbb{N}$. Tem-se:

$$\begin{aligned} cod(f)(x) &= p^k(f((p^n)^{-1}(x))) \\ cod^{-1}(h)(x) &= = (p^k)^{-1}(h(p^n(x))) \end{aligned}$$

Estas codificações permitem enumerar todas os k-tuplos de números inteiros.

Exercício 6.0.15 Define uma função que codifique uma qualquer sequência finita de inteiros em inteiros. Sugestão: Define $L(x)$ a função comprimento de $x \in \mathbb{N}^\star$.

Genericamente sendo Σ um conjunto finito ou numerável de símbolos, funções bijectivas computáveis de $\Sigma^\star$ em $\mathbb{N}$ (ou f injectiva de $\Sigma^\star$ em $\mathbb{N}$ e f^{-1} sobrejectiva, ou de domínio restrito ao contradomínio de f, também computável) denominam-se *numerações de Gödel*.[1]

Vamos dar alguns exemplos:

[1] Pelo facto de Kurt Gödel (1930) ter usado pela primeira vez estas codificações para aritematizar a teoria axiomática dos números, isto é, codificar em inteiros proposições, teoremas e demonstrações

1. Seja $\Sigma = \{d_1, \ldots, d_{n-1}\}$. Então qualquer $w \in \Sigma^\star$, $w = d_{i_1} \ldots d_{i_k}$ pode ser representado como um número inteiro na base $n =\mid \Sigma \mid +1$:

$$c(w) = \sum_{j=1}^{k} n^{k-j} i_j$$

   Por exemplo, se $\Sigma = \{a, b, c\}$ então

$$c(aabb) = 4^3 \times 1 + 4^2 \times 1 + 4^1 \times 2 + 2 = 90$$

   Verifica que nesta codificação nem todos os inteiros são códigos válidos!

2. Outra hipótese é ordenar os elementos de $\Sigma^\star$ pelo comprimento e se tiverem o mesmo comprimento por ordem lexicográfica. Dado $\Sigma = \{d_1, \ldots, d_n\}$ vem

$$\begin{aligned} c(\epsilon) &= 0 \\ c(d_i) &= i \qquad \text{para todo } 1 \leq i \leq n \\ c(wd_i) &= nc(w) + i \quad \text{para todo } w \in \Sigma^\star \text{ e } i,\ 1 \leq i \leq n \end{aligned}$$

   Então, $c(d_{i_1} \ldots d_{i_k}) = n^{k-1} i_{i_1} + \ldots + n i_{k-1} + i_k$. No caso do exemplo anterior ter-se-ia:

0	1	2	3	4	5	6	...	44	...
ϵ	a	b	c	aa	bb	cc	...	aabb	...

3. Baseada na decomposição única em factores primos[2] de qualquer $n > 1$, tem-se a seguinte codificação $c : \Sigma^\star \longrightarrow \mathbb{N}$, sendo $\Sigma = \{d_0, d_1, \ldots\}$:

$$\begin{aligned} c(d_i) &= p_i \qquad \text{onde } p_i \text{ é o } i\text{-ésimo primo, sendo } p_0 = 2 \\ c(d_{i_0} d_{i_1} \ldots d_{i_k}) &= 2^{c(d_{i_0})} 3^{c(d_{i_1})} \ldots p_k^{c(d_{i_k})} \end{aligned}$$

   Para o exemplo anterior tem-se que $c(aabb) = 2^2 3^2 5^3 7^3 = 1543500$

   Nota que esta codificação não é sobrejectiva em $\mathbb{N}$.

Exercício 6.0.16 Define uma numeração de Gödel para programas $WHILE$, tal que g represente o programa P_g que calcula f_g. Isso permite em particular enumerar a classe R de funções. Escreve um programa em $WHILE$ que dado g e $x \in \mathbb{N}^n$ simule o programa P_g.

Exercício 6.0.17 [Máquina Universal de Turing]

a) Inventa uma descrição genérica $m : TM \to \{0,1\}^*$ de máquinas de Turing utilizando apenas o alfabeto $\{0, 1\}$.

b) Descreve em linguagem C uma função $f(m, x)$ onde m é um "string" que representa uma máquina de Turing M (isto é, $m = d(M)$), x é o "string" dos dados e

   (a) f retorna SIM se M aceita x.

   (b) f retorna NAO se M rejeita x.

   (c) f entra em ciclo infinito se M com dados x entra em ciclo infinito.

c) Dadas duas sequências x e y define uma função $c(x, y)$ cujo resultado é uma sequência de 0s e 1s a partir da qual se pode extrair x e y. Obviamente a concatenação xy não satisfaz.

[2] Esta é semelhante à usada por Kurt Gödel.

d) Repete a questão 5 utilizando uma função de um só argumento, $g(c(m,x))$ onde c é a função descrita em c).

e) Mostra que existe uma máquina de Turing U que faz o "mesmo" da função g da alínea anterior (inicialmente a fita contém $c(m,x)$).

f) Define a máquina de Turing U.

g) Mostra que U imita qualquer máquina com quaisquer dados no sentido seguinte

   (a) M/x termina em s_1 sse $U/c(d(M,x))$ termina em s_1.

   (b) M/x termina em s_2 sse $U/c(d(M,x))$ termina em s_2.

   (c) M/x não termina sse $U/c(d(M,x))$ não termina.

Nos problemas seguintes supõe-se que as instâncias de máquinas de Turing são codificadas em palavras com símbolos de um alfabeto Σ. É evidentemente decidível se um determinado $x \in \Sigma$ representa ou não a codificação de uma máquina de Turing. Essa decidibilidade corresponde a uma função (total) $t : \Sigma^* \to \{F, V\}$.

Exercício 6.0.18 Mostra que existe uma função total $f : \mathbf{N} \to \Sigma^*$ cujo resultado é a máquina de Turing de ordem n (na ordem lexicográfica).
Nota: descrever uma implementação de f numa linguagem de alto nível onde se supõe a existência da função d referida.

Exercício 6.0.19 Mostra que existe uma função total $f : \Sigma^* \to \mathbf{N} \cup \{-1\}$ cujo resultado é n se x representa a máquina de ordem n e -1 se x não representa nenhuma máquina de Turing.

Exercício 6.0.20 Mostre que é possível codificar o conjunto das máquinas de Turing por forma que o contradomínio seja Σ^*, isto é, por forma que todo $x \in \Sigma^*$ seja a representação de uma máquina de Turing.
Sugestão: Seja y a codificação usual da máquina. Seja n o seu número. A codificação sobrejectiva z é a sequência cuja ordem lexicográfica é n.

Exercício 6.0.21 Mostra que não se perde generalidade se numa codificação supusermos $||\Sigma|| = 2$ ou $||\Sigma|| = 1$.

Bibliografia

[BC94] Daniel P. Bovet and Pierluigi Crescenzi, *Introduction to the theory of complexity*, Prentice Hall, 1994.

[Dew89] A. K. Dewdley, *The turing omnibus – 61 excursions in computer science*, Computer Science Press, 1989.

[eAP95] Armando B. Matos e António Porto, *Ackermann and the superpowers*, Tech. report, Faculdade de Ciências da Universidade do Porto, 1995.

[Hof79] Douglas R. Hofstadler, *Gödel, escher, bach: an eternal golden braid*, Basic Books, 1979.

[HU79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to automata theory, languages and computation*, Addison Wesley, 1979.

[Kar72] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations (R. E. Miller and J. W. Tatcher, eds.), Plenum Press, 1972, pp. 85–103.

[LP81] Harry R. Lewis and Christos H. Papadimitriou, *Elements of the theory of computation*, Prentice Hall, 1981.

[LV90] Ming Li and Paul M. B. Vitànyi, *Kolmogorov complexity and its applications*, Handbook of Theoretical Computer Science (J. van Leewen, ed.), vol. A, Elsevier Science Publishers, 1990, pp. 188–254.

[Man74] Zohar Manna, *Mathematical theory of computation*, Addison Wesley, 1974.

[Mat91] Armando B. Matos, *Pesar sem pesos*, Tech. report, Centro de Informática da Universidade do Porto, 1991.

[Mat95a] ———, *Algoritmos probalísticos – algumas notas*, Tech. report, Faculdade de Ciências da Universidade do Porto, 1995.

[Mat95b] ———, *Entropia, algoritmos e complexidade mínima*, Tech. report, Faculdade de Ciências da Universidade do Porto, 1995.

[Min74] M. Minsky, *Finite and infinite machines*, Addison Wesley, 1974.

[Pap94] Christos H. Papadimitriou, *Computational complexity*, Addison Wesley, 1994.

[Phi92] I. C. C. Phillips, *Recursion theory*, Handbook of Logic in Computer Science (S. Abramsky D. M. Gabbay and T.S.E. Maibaum, eds.), vol. 2, Oxford University Press, 1992, pp. 79–187.

[Rog67] H. Rogers, *Theory of recursive functions and effective computability*, McGraw-Hill, 1967.

[Sha48] Claude E. Shannon, *A mathematical theory of communication*, Bell System Technical Journal **27** (1948), 379–423,623–656, Part I,Part II.

[SvW88] R. Sommerhalder and S.C. van Westrhenen, *The theory of computability: Programs, machines, effectiveness and feasibility*, Addison Wesley, 1988.

[Wei87] Klaus Weihrauch, *Computability*, Springer-Verlag, 1987.

[Yas71] Ann Yasuhara, *Recursive function theory and logic*, Computer Science and Applied Mathematics, Academic Press, 1971.