Notas em Matemática Aplicada e-ISSN 2236-5915

Volume 11, 2012

Editado por

Eliana X.L. de Andrade
Universidade Estadual Paulista - UNESP
São José do Rio Preto, SP, Brasil

Rubens Sampaio
Pontifícia Universidade Católica do Rio de Janeiro -
Rio de Janeiro, RJ, Brasil

Geraldo N. Silva
Universidade Estadual Paulista - UNESP
São José do Rio Preto, SP, Brasil

Sociedade Brasileira de Matemática Aplicada e Computacional

2012

A Sociedade Brasileira de Matemática Aplicada e Computacional - SBMAC publica, desde as primeiras edições do evento, monografias dos cursos que são ministrados nos CNMAC.

Para a comemoração dos 25 anos da SBMAC, que ocorreu durante o XXVI CNMAC em 2003, foi criada a série **Notas em Matemática Aplicada** para publicar as monografias dos minicursos ministrados nos CNMAC, o que permaneceu até o XXXIII CNMAC em 2010.

A partir de 2011, a série passa a publicar, também, livros nas áreas de interesse da SBMAC. Os autores que submeterem textos à série Notas em Matemática Aplicada devem estar cientes de que poderão ser convidados a ministrarem minicursos nos eventos patrocinados pela SBMAC, em especial nos CNMAC, sobre assunto a que se refere o texto.

O livro deve ser preparado em **Latex (compatível com o Miktex versão 2.7), as figuras em eps** e deve ter entre **80 e 150 páginas**. O texto deve ser redigido de forma clara, acompanhado de uma excelente revisão bibliográfica e de **exercícios de verificação de aprendizagem** ao final de cada capítulo.

Veja todos os títulos publicados nesta série na página
http://www.sbmac.org.br/notas.php

SBMAC **Sociedade Brasileira de Matemática Aplicada e Computacional**

2012

COMPUTABILIDADE: OS LIMITES DA COMPUTAÇÃO

Regivan H. N. Santiago - UFRN
regivan@dimap.ufrn.br

Benjamín R. C. Bedregal - UFRN
bedreal@dimap.ufrn.br

Sociedade Brasileira de Matemática Aplicada e Computacional

São Carlos - SP, Brasil
2012

Coordenação Editorial: Véra Lucia da Rocha Lopes

Coordenação Editorial da Série: Geraldo Nunes Silva

Editora: SBMAC

Capa: Matheus Botossi Trindade

Patrocínio: SBMAC

Catalogação elaborada pela Biblioteca do IBILCE/UNESP
Bibliotecária: Maria Luiza Fernandes Jardim Froner

Santiago, Regivan H. N.
Computabilidade: os limites da Computação
- São Carlos, SP: SBMAC, 2012, 94 p., 20.5 cm
- (Notas em Matemática Aplicada; v. 11)

e-ISBN 978-85-86883-78-1

1. Computabilidade. 2. Funções Recursivas Parciais.
3. Teoria da Computação. 4. Procedimentos Efetivos.
I. Santiago, Regivan H. N. II. Bedregal, Benjamín R.
III. Título. IV. Série.

CDD - 51

Esta é uma republicação em formato de e-book do livro original do mesmo título publicado em 2004 nesta mesma série pela SBMAC.

Para Adriane e Ivanosca.

Prefácio

Entendendo por computação tudo o que os computadores podem realizar, então é necesário definir precisamente o que é um computador. Essa resposta poderia ser dada em termos de hardwares e tecnologias, mas deve haver o cuidado para que não haja uma limitação à tecnologia do momento, pois nessa definição deverão coexistir os primeiros computadores, as calculadoras, os supercomputadores, até os computadores que estão por existir. Ou seja, é necessário unificar essas características essenciais e comuns de TODOS os possíveis computadores e chegar a um modelo matemático capaz de realizar qualquer tarefa realizável nos computadores reais. Com isso é possível definir precisamente o que é uma tarefa executada por qualquer computador, i.e. uma **tarefa computável**, dando, assim, origem à noção de **computabilidade**. A **teoria da computabilidade** procura responder a partir desses modelos questões como: *o que, em princípio, os computadores podem fazer?* (sem qualquer restrição de espaço, tempo, nem recurso) e *quais são as inerentes limitações teóricas?*, i.e. *o que pode e o que não pode ser feito por um computador? ou qual a classe de funções que um computador consegue implementar?*

A computação em cada um desses modelos *implementa* uma noção do que vem a ser um **procedimento efetivo**, i.e. uma regra mecânica, ou um método automático, ou um programa para executar alguma operação matemática (c.f. Cutland [8]). Um exemplo de procedimento efetivo é o algoritmo da divisão de Euclides, que demonstra que os gregos antigos já se preocupavam com esse tipo de procedimento. Mas só em 1936 os matemáticos Alan Turing e Alonzo Church, de maneira independente, propuseram formalizações ("modelos") distintas para esse conceito. Essas e outras formulações, que vieram posteriormente, mostraram-se equivalentes, já que computam a mesma classe de funções, a saber, a **classe das funções recursivas parciais** que é uma sub-classe *própria* da classe de funções sobre os naturais. Isso significa que todos esses modelos têm o mesmo poder computacional e que a noção de computação limita-se a essa classe de funções. A correspondência entre procedimentos efetivos e esses modelos formais é conhecida como **tese de Church-Turing**.

O minicurso apresenta 3 classes de modelos de computação que captam aspectos distintos da computação atual, a saber: (1) RAM: Random Access Machines que capta o aspecto dos hardwares; (2) Programas While, que capta o aspecto das linguagens de programação — "Software" e (3) Funções recursivas parciais, que captam o aspecto funcional da computação. Mostra-se a equivalência entre essas

classes, fortalecendo a validade da Tese de Church-Turing.

A segunda etapa do curso, descrita no capítulo 6, apresenta o conceito de **programas universais**, demonstrando que existem modelos, em cada uma dessas classes, capazes de abstrair as atuais arquiteturas von Neumann. Esses programas universais modelam os computadores de propósito geral, i.e. computadores programáveis que simulam qualquer programa de propósito específico.

A terceira etapa, apresentada no capítulo 7, introduz a noção de **procedimento de decisão**, que são procedimentos que verificam se uma propriedade é satisfeita ou não por um dado de entrada. Esses procedimentos são divididos em três classes: procedimentos decidíveis, semi-decidíveis e indecidíveis. A primeira e a segunda classe são sub-classes dos procedimentos efetivos. Ao passo que a terceira classe é disjunta da primeira e da segunda. Dentre os problemas pertencentes às duas últimas classes encontram-se os **problemas da parada e da divergência**, um sendo a negação do outro, e ambos pertencentes à **teoria da programação**, onde o primeiro é semi-decidível e o outro completamente indecidível.

A quarta e última etapa, contida no capítulo 8, apresenta, de maneira sucinta, alguns tópicos que não foram abordados neste texto, indicando uma bibliografia suplementar e a importância deles no contexto da computação. Dentre esses tópicos, destacamos, por enquanto, as questões de paralelismo, computabilidade no contínuo (já que a computabilidade vista no minicurso será sobre conjuntos contáveis) e computações com oráculos.

Agradecimentos

- Aos alunos de graduação em Engenharia da Computação da UFRN que cursaram a disciplina Computabilidade no semestre 2004-01 e aos alunos de pós-graduação em Sistemas e Computação, da mesma universidade, que cursam a disciplina Teoria da Computação no mesmo semestre, pois ao serem submetidos a esse livro contribuíram com várias correções; e

- À SBMAC pela oportunidade que nos foi dada.

Conteúdo

Capítulo 1

Pré-requisitos Matemáticos

1.1 Conjuntos, relações e funções

Esta seção deve ser lida rapidamente e revista posteriormente caso seja necessário. Ela se propõe a ser uma introdução da linguagem matemática utilizada neste texto. Observe cuidadosamente as definições de função, produto cartesiano e tuplas dadas aqui. Pressupõe-se familiaridade com a teoria elementar de conjuntos e os conceitos de pertinência "$\in$", união de conjuntos "$\cup$", intersecção de conjuntos "$\cap$", complemento "$\overline{A}$", conjunto vazio "$\emptyset$", inclusão de conjuntos "$A \subseteq B$", e inclusão própria de conjuntos $A \subset B$.

Se X e Y são conjuntos, então o conjunto $X \times Y = \{(x, y) : x \in X, y \in Y\}$ chama-se **produto cartesiano de X e Y**, e é o conjunto de todos os pares ordenados, (x, y), formados a partir de X e Y. Um subconjunto $R \subseteq X \times Y$ chama-se **relação** entre X e Y. Observe que se $X = \emptyset$ ou $Y = \emptyset$, então $X \times Y = \emptyset$, nesse caso como $\emptyset$ é único, então a relação $\emptyset$ chama-se **relação vazia**. .

O domínio de uma relação R, escrito como $dom\ R$, é o conjunto $\{x \in X : (x, y) \in R,\ \text{para algum } y \in Y\}$. Se $x \in dom\ R$, então R **está definida para** x, o que pode ser escrito como: "$R(x) \downarrow$". Se $x \notin dom\ R$, então R **não está definida para** x, o que pode ser designado por: "$R(x) \uparrow$". A **imagem** de R é o conjunto $im(R) = \{y \in Y : (x, y) \in R,\ \text{para algum } x \in X\}$. A **imagem direta** de um subconjunto $A \subseteq X$ é o conjunto $R(A) = \{y \in Y : (x, y) \in R,\ \text{para algum } x \in A\}$. Assim, $R(dom\ R) = im(R)$. A **restrição** de R à A, onde $A \subseteq X$, denotada por $R \mid A$, é a relação $\{(x, y) \in R : x \in A\}$. A relação inversa de R, designada por R^{-1}, é o conjunto de pares ordenados $\{(y, x) : (x, y) \in R\}$.

Uma relação $f \subseteq X \times Y$ é uma **função parcial** de X em Y, representada por $f : X \to Y$, se para cada $x \in dom\ f$ existe um único $y \in Y$, tal que $(x, y) \in f$. Se, além disso, $dom f = X$, então a função f chama-se **função total**. Dessa forma, uma função total é parcial, mas a recíproca não é válida.

f é uma **função injetora**, se $f(x) = f(x')$, então $x = x'$. Ou seja, dois objetos distintos sempre estarão relacionados, via f, com dois elementos distintos de Y. f é uma **função sobrejetiva**, se $im(f) = Y$. Assim, uma função é injetora se a sua *relação* inversa f^{-1} é uma função parcial e é sobrejetiva se *dom* $f^{-1} = Y$. Uma função total injetora e sobrejetora chama-se **bijeção** ou **função bijetora** de X em Y.

O **conjunto de todas as funções totais de** X **para** Y é um subconjunto do conjunto potência $\mathcal{P}(X \times Y)$, e será denotado por Y^X.

Proposição 1.1 *Se $f : X \to Y$ é uma função total, então: (1) se $X \neq \emptyset$,então $Y \neq \emptyset$; e (2) se $X = \emptyset$, então $f = \emptyset$ (i.e. $f = \emptyset \subseteq X \times Y = \emptyset$).*

Prova: Suponha que $f : X \to Y$ é uma função total. (1) Se X não é vazio, então existe pelo menos um elemento em X. Portanto, pela definição de função total deve existir pelo menos $y \in Y$ tal que $(x, y) \in f$. Logo $Y \neq \emptyset$. (2) Se $X = \emptyset$, então $X \times Y = \emptyset$. Logo, o único subconjunto de $X \times Y$ que satisfaz a definição de função total é o conjunto $\emptyset$. Note, que $\emptyset$ satisfaz a definição de função total por vacuidade, i.e. $\emptyset$ não possui qualquer par ordenado (x, y) que viole a definição de função total. □

Corolário 1.2 *Existe uma única função de $\emptyset$ em qualquer conjunto A; a saber o conjunto vazio $\emptyset$ (também chamado **função vazia**).*

Proposição 1.3 *Não existe função total de A em $\emptyset$, para $A \neq \emptyset$.*

Prova: Para existir uma função total f de A em $\emptyset$, onde $A \neq \emptyset$, então é suficiente que para cada $a \in A$, exista um único $x \in \emptyset$ tal que $(a, x) \in f$. O que é absurdo. □

Corolário 1.4 *Para todo conjunto A, $A^\emptyset = \{\emptyset\}$, e para todo conjunto $B \neq \emptyset$, $\emptyset^B = \emptyset$.*

1.1.1 Cardinalidade e enumerabilidade

Um dos resultados da teoria dos conjuntos de George Cantor foi a definição da **cardinalidade** ou **tamanho** de um conjunto. Ingenuamente, a cardinalidade ou tamanho de um conjunto é a quantidade de elementos que ele possui. Portanto os conjuntos $\{a\}$, $\{a, b\}$, $\{a, b, c\}$, ... são conjuntos, respectivamente, com 1, 2, 3, ... elementos, e portanto possuem cardinalidade 1, 2, 3, ... Entretanto, isso faz sentido para conjuntos finitos, onde se pode associar a cada conjunto um número natural e dessa forma concluir que os conjuntos que possuem a mesma quantidade de elementos possuem o mesmo tamanho; i.e. a mesma cardinalidade. Entretanto, como estender esse conceito para conjuntos infinitos? Cantor propôs o seguinte:

Definição 1.5 (Cardinalidade, Conjuntos contáveis e incontáveis) *Dado dois conjuntos A e B. Se existe uma bijeção entre A e B, então diz-se que A e B possuem a mesma **cardinalidade**. Se $A = \emptyset$ ou existe uma bijeção entre A e o conjunto $\{0, \ldots, m\}$ (para $m \geq 0$), então A chama-se **conjunto finito**. Quando existe uma bijeção entre A e $\mathbb{N}$, o conjunto A é chamado **conjunto enumerável**. Se A é finito ou enumerável, então A é dito um **conjunto contável**, e se A não for um conjunto contável, então ele chama-se **conjunto incontável**.*

Exemplo 1.6 *Os conjuntos $\{a, b, c\}$ e $\{x \in \mathbb{N} : x = 2 \cdot K \text{ para algum } k \in \mathbb{N}\}$ são conjuntos contáveis, ao passo que $\mathbb{R}$ não é um conjunto contável.*

1.1.2 Famílias e produto cartesiano genérico

"Existem ocasiões em que a imagem de uma função é tida como mais importante do que a própria função. Quando este é o caso, a terminologia e a notação, ambas, passam por radicais alterações." — Paul R. Halmos [14] p.55.

Suponha, que x é uma função total do conjunto I para um conjunto X. Se neste caso quem estará em evidência é a imagem de x em vez da função em si, então serão necessárias algumas terminologias novas. Um elemento $i \in I$ se chamará **índice**, I será chamado **conjunto de índices**, o contradomínio de x será dito **conjunto indexado**, a função em si será denominada **família**, e o valor da função x para cada índice $i \in I$, receberá o nome: **termo da família** e será indicado por x_i.

Exemplo 1.7 *Uma **seqüência** $s : \{1, \ldots, n\} \to \mathbb{R}$ é um bom exemplo de uma família, onde $I = \{1, \ldots, n\}$, $\mathbb{R}$ está indexado por I e s_i é um termo da seqüência s.*

"Um inaceitável mas geralmente admitido caminho de comunicar a notação e indicar a ênfase (no contradomínio) é falar de uma família $\{x_i\}$ em X, ou de uma família $\{x_i\}$ quaisquer que possam ser os elementos de X; quando necessário o conjunto de índices I é indicado por alguma expressão entre parênteses como $(i \in I)$." — Paul R. Halmos [14] p.56. Duas outras alternativas de notação são $\{x_i\}_{i \in I}$ e $\{x_i : i \in I\}$.

Dois conceitos bem simples porém muito importantes são o de família constante e o de família de conjuntos.

Definição 1.8 *Dados dois conjuntos I e X, e um elemento $x \in X$. Uma **família constante** é uma família $c : I \to X$, tal que $c_i = x$, para todo $i \in I$.*

Definição 1.9 *Uma **família de subconjuntos** de X é uma família da forma $A : I \to \mathcal{P}(X)$. Seguindo as observações acima, pode-se, então, designar uma família de subconjuntos como $\{A_i\}_{i \in I}$ ou $\{A_i : i \in I\}$.*

Exemplo 1.10 *Sejam os conjuntos* $I = \{1, 2, 3\}$ *e* $\mathcal{P}(\{a, b, c\})$*, então a função* $A : I \to \mathcal{P}(\{a, b, c\})$*, onde:* $A(1) = \{a, b\}$*,* $A(2) = \emptyset$*, e* $A(3) = \{c\}$*, é uma família de conjuntos. Observe que para o conjunto de índices* $J = \{u, v, x\}$*, a família* $B : J \to \mathcal{P}(\{a, b, c\})$*, onde:* $B(u) = \{a, b\}$*,* $B(v) = \emptyset$*, e* $B(x) = \{c\}$ *é uma família diferente de A, porém equivalente. Infelizmente essa equivalência não será formalizada nesse texto.*

Definição 1.11 *Se* $\{A_i\}_{i\in I}$ *é uma família de subconjuntos de X, i.e.* $A : I \to \mathcal{P}(X)$*, a união da imagem da família é chamada a* ***união da família*** *A; a notação padrão para isso é:*

$$\bigcup_{i\in I} A_i \text{ ou } \bigcup_i A_i.$$

Dessa maneira, a união de uma família de subconjuntos de X é também um subconjunto de X.

A linguagem das famílias pode ser utilizada para generalizar a noção de produto cartesiano.

"O produto cartesiano de dois conjuntos X e Y, foi definido como o conjunto de todos os pares ordenados (x, y) com $x \in X$ e $y \in Y$. Existe uma natural correspondência um-a-um entre este conjunto e um certo conjunto de famílias. De fato, considere qualquer par particular não ordenado $\{a, b\}$, com $a \neq b$, e considere o conjunto Z de todas as famílias z, indexadas por $\{a, b\}$, tal que $z_a \in X$ e $z_b \in Y$. Se a função f de Z para $X \times Y$ é definida por $f(z) = (z_a, z_b)$, então f é a correspondência um-a-um prometida. A diferença entre Z e $X \times Y$ é uma mera questão de notação. A generalização de produtos cartesianos generaliza Z mais do que o próprio $X \times Y$. (Como conseqüência há um pequeno atrito de terminologia na passagem do caso especial para o geral. Não há como evitá-lo; é com o a linguagem matemática é usada atualmente). A generalização agora é direta." — Paul R. Halmos [14] p.58.

Definição 1.12 *Seja* $\{A_i\}_{i\in I}$ *é uma família de subconjuntos de X, i.e. uma função* $A : I \to \mathcal{P}(X)$*, o* ***produto cartesiano da família*** *A, denotado por* $\Pi_{i\in I} A_i$ *ou* $\Pi_i A_i$*, é o conjunto de todas as famílias* $x : I \to \bigcup_{i\in I} A_i$*, tal que* $x_i \in A_i$*. Dessa forma, fazendo* $U = \bigcup_{i\in I} A_i$*, o produto cartesiano* $\Pi_i A_i$ *é um subconjunto do conjunto de todas as funções totais de I em U, i.e.* $\Pi_i A_i \subseteq U^I$*.*

Produto cartesiano e exponenciação de conjuntos. Se X e Y são conjuntos, considere a família constante de conjuntos $c : Y \to \mathcal{P}(X)$, tal que o valor constante é A, onde $A \subseteq X$, i.e. $c_y = A$, para todo $y \in Y$, então $\bigcup_{y\in Y} c_y = A$. Por definição, o produto cartesiano $\Pi_{y\in Y} c_y$ é o conjunto de todas as famílias $f : Y \to A$, tal que $f_y \in A$, ou mais precisamente, o produto cartesiano, nesse caso, é o conjunto de todas as funções totais de Y em A, i.e. "A^Y". Portanto, quando a família de conjuntos $c : Y \to \mathcal{P}(X)$ é constantemente igual a A, então o produto cartesiano

dessa família coincide com o conjunto de todas as funções totais de Y em A.

Observe que para dois conjuntos quaisquer X e Y, quando $Y = \emptyset$, então existe apenas uma e somente uma função de Y em X; ou seja a função vazia $\emptyset$, o que significa que $X^{\emptyset} = \{\emptyset\}$. Portanto, se o conjunto de índices é vazio, i.e. se $I = \emptyset$, então qualquer que seja a família de conjuntos $A : I \to \mathcal{P}(X)$ ela será uma função pertencente ao conjunto $\mathcal{P}(X)^{\emptyset} = \{\emptyset\}$. Mais especificamente, observe que $\emptyset$ é igual a qualquer função $f : \emptyset \to \emptyset$, portanto a imagem de A que é um subconjunto do contradomínio $\mathcal{P}(X)$ só pode ser $\emptyset$, e portanto a união $\bigcup_{i\in I} A_i = \emptyset$. Assim, uma família $x : I \to \bigcup_{i\in I} A_i$, na verdade será uma função da forma $x : \emptyset \to \emptyset$, que será igual a $\emptyset$. Logo, o produto cartesiano $\Pi_i A_i$ que é o conjunto de todas as famílias $x : I \to \bigcup_{i\in I} A_i$, na verdade é o conjunto $\{f : \emptyset \to \emptyset\}$ que é igual a $\emptyset^{\emptyset}$, e por conseguinte igual a $\{\emptyset\}$. Em resumo, o produto cartesiano quando o conjunto de índices é vazio é igual ao conjunto unitário $\{\emptyset\}$.

1.1.3 Produto cartesiano e tuplas

Se I é um par $\{u, v\}$, com $u \neq v$, e $\{A_i\}_{i\in I}$ é uma família de conjuntos, então é costume identificar $\prod_{i\in I} A_i$ com o produto cartesiano $A_u \times A_v$. Observe que os elementos de $\prod_{i\in I} A_i$ são funções, ao passo que os elementos de $A_u \times A_v$ são pares ordenados (p, q). Tem-se, portanto, duas formas de representar um par ordenado, quer como um objeto da forma (p, q) ou como uma função $x : \{u, v\} \to \bigcup_{i\in\{u,v\}} A_i$. Seguindo a generalização do produto cartesiano, vista anteriormente, pode-se então pensar numa **tripla ordenada** como sendo uma família $x : \{u, v, w\} \to \bigcup_{i\in\{u,v,w\}} A_i$, e portanto um elemento do produto cartesiano $\Pi_{i\in\{u,v,w\}} A_i$, onde $A : \{u, v, w\} \to \mathcal{P}(X)$ é uma família de conjuntos. Nesse caso denota-se uma tripla ordenada $x : \{u, v, w\} \to \bigcup_{i\in\{u,v,w\}} A_i$, como (x_u, x_v, x_w). Observe que se o conjunto de índices é igual à $\{1, 2, 3\}$ obtém-se a notação comumente usada: "(x_1, x_2, x_3)". Semelhantemente, pode-se definir **tuplas ordenadas**, i.e. quádruplas ordenadas, quíntuplas ordenadas, etc. como famílias cujos conjuntos de índices são **tuplas não ordenadas**, i.e. triplas não ordenadas, quádruplas não ordenadas, etc.

Um caso particular da generalização do produto cartesiano diz respeito a situação em que o conjunto de índices é um conjunto unitário, digamos $I = \{v\}$. Nesse caso, a imagem de uma família de conjuntos $A : I \to \mathcal{P}(X)$ é um conjunto unitário $A_v = \{B\}$, para $B \subseteq X$. Em outras palavras, cada família está associada à um único subconjunto de X. Além disso, $\bigcup_{i\in I} A_i = A_v$, e o produto $\Pi_i A_i$ que é o conjunto de todas as famílias $x : I \to \bigcup_{i\in I} A_i$, onde $x_i \in A_i$, é o conjunto de todas as funções da forma $f : \{v\} \to A_v$. Dessa forma, é possível construir uma correspondência biunívoca entre A_v e o produto cartesiano $\Pi_{i\in I} A_i$, é suficiente estabelecer a correspondência entre cada $k \in A_v$, e a função $f_k : \{v\} \to A_v$, onde $f_k(v) = k$. Assim, cada elemento de A_v pode ser visto como uma função de $\Pi_{i\in I} A_i$ e vice versa. Logo pode-se aplicar um abuso de linguagem e confundir A_v com $\Pi_{i\in I} A_i$, onde os elementos de A_v são vistos como tuplas de tamanho 1. Observe

que essa construção justifica a visão de um elemento de um certo conjunto como sendo uma função.

Outro caso particular, é o caso em que o conjunto de índices é $\emptyset$. Esse caso já foi comentado anteriormente, e portanto $\Pi_i A_i = \{\emptyset\}$. Dessa forma, a idéia de tupla vazia, que alguns livros representam como se "()", na verdade é formalizada pelo conjunto $\emptyset$. Assim nos casos em que a tupla $(x_1, \ldots, x_n)$ for vazia, por exemplo em situações de vaquidade, i.e. $n = 0$, as notações $(x_1, \ldots, x_n)$ e $f(x_1, \ldots, x_n)$ podem ser entendidas como "()" e "$f()$". O leitor deve observar que existe uma dificuldade de padronização notacional para esses casos, por exemplo $f()$ é na verdade $f(\emptyset)$

O conhecido **produto cartesiano m-ário** sobre um conjunto X qualquer, é comumente definido como $X^m = \{(x_1, \ldots, x_m) : x_i \in X \wedge 1 \leq i \leq m\}$. Pode-se recuperar essa definição através do conceito de famílias da seguinte maneira: Um número natural m pode ser visto como o conjunto de todos os seus antecessores. Por exemplo $0 = \emptyset$, $1 = \{0\}$, $2 = \{0, 1\}$, $3 = \{0, 1, 2\}$, etc. Assim, tomando um número natural como um conjunto, pode-se então pensar num número natural como um conjunto de índices I. Além disso, como as componentes de um produto m-ário ($m \geq 1$) são elementos do mesmo conjunto X, então o produto cartesiano em questão é o produto $\Pi_{y \in \{0,\ldots,m-1\}} c_y$, onde $c : \{0, \ldots, m-1\} \to \mathcal{P}(X)$ é a família de conjuntos constantemente igual à X. Note que, segundo o que foi comentado, o produto $\Pi_{y \in \{0,\ldots,m-1\}} c_y$ coincide com o conjunto de todas as funções totais de $\{0, \ldots, m-1\}$ em X, ou seja X^m. Para o caso em que $m = 0$, i.e. $m = \emptyset$, $X^0 = \{\emptyset\}$ (veja os comentários precedentes). Logo, a linguagem de famílias formaliza também a noção de produto cartesiano m-ário, para $m \geq 0$, sobre um conjunto X.

O conceito de tuplas infinitas generaliza-se diretamente do precedente, bastando para isso tomar um conjunto infinito de índices. Observe que a notação de tuplas $(x_1, \ldots, x_n)$, e $(x_1, \ldots, x_n, \ldots)$, tem um limite, pois quando o conjunto de índices I não é contável essa notação não tem como expressar as componentes da tupla, visto que a quantidade de componentes é maior do que se pode escrever. Portanto, a notação de famílias, embora um pouco rebuscada, é uma notação mais geral para expressar os conceitos de produto cartesiano e tuplas ordenadas. Entretanto, algumas bibliografias utilizam a notação $(x_i)_{i \in I}$ para recuperar a notação usual de tuplas para um conjunto de índices qualquer.

Capítulo 2

RAM: Máquinas de Acesso Randômico

Por questões de eficiência, os computadores modernos permitem que os dados armazenados na memória sejam acessados de maneira aleatória, i.e. não é necessário que eles sejam acessados de maneira seqüencial (um após o outro). Para isso, é atribuído à cada dado um *endereço de memória* que quando localizado pelo computador faz com que os mesmos tornem-se "disponíveis" para a computação. Matematicamente, poderia-se pensar na memória de um computador digital real como um vetor:

$$M : \{0, \ldots, n-1\} \to \{0,1\}^+,$$

onde $\{0,1\}^+$ é o conjunto de seqüências finitas não nulas de 0's e 1's — e.g. $0, 1, 01, 101,\ 001100111 \in \{0,1\}^+$ — que representam as informações no formato digital e os índices $0, \ldots, n-1$ representam os endereços de memória. Essas cadeias são chamadas **strings binárias**, e são uma dentre muitas formas de representar a informação.

O nome **máquina de acesso randômico, RAM,** representa o fato que os antigos modelos de computação não possuíam acesso aleatório; eles eram baseados em fitas seqüenciais (e.g. máquinas de Turing) ou tinham natureza funcional (e.g. λ-calculus). Como ponto de partida, considera-se um modelo que em muito lembra uma máquina assembly, *ele é uma variação do modelo RAM proposto em Smith [27].* Ao contrário de uma máquina assembly real, ele possui instruções muito simples, pois objetiva-se simplificar as provas matemáticas. Entretanto o poder de computação dessa máquina será o mesmo de qualquer máquina assembly concreta.

2.1 Arquitetura da RAM

O leitor pode perceber que no modelo vetorial de computador descrito acima, a quantidade de memória é finita. Entretanto, para que se possa ter uma teoria do

que é computável em *qualquer* máquina, é necessário que ela seja desenvolvida para computadores com qualquer quantidade n de memória. Dessa forma, daqui por diante, um computador terá o seguinte aspecto vetorial:

$$M : \mathbb{N} \to \{0,1\}^+,$$

Isso pode ser ingenuamente interpretado como: "não importa o quanto se aumente a capacidade de memória de um computador, os fatos estabelecidos na teoria da computabilidade continuarão valendo para o computador com memória estendida". Outro ponto que precisa ser observado, é que será considerado tanto memória primária como secundária (tapes, HD's, CD's, DVD's, etc.) como simplesmente memória, pois o que se quer modelar são: dados e computações sobre esses dados, não importando a maneira como eles estão armazenados.

2.1.1 Estrutura de memória das RAM

Antes de definir a estrutura da memória da RAM, vale observar que como as strings binárias, citadas acima, são apenas uma das várias maneiras de representar a informação, é possível encontrar uma maneira equivalente que seja mais intuitiva para representá-la. Essa equivalência é dada pela conversão de base entre os sistemas de numeração binária e decimal. Assim, uma string binária nada mais será do que um número natural, e o modelo vetorial de computador terá o seguinte aspecto:

$$M : \mathbb{N} \to \mathbb{N} \tag{2.1}$$

Máquina de acesso randômico — RAM: Possui uma quantidade enumerável de registradores $R_1, R_2, \ldots$. Cada registrador armazena um número natural. Graficamente, pode-se pensar na memória da RAM da seguinte maneira:

Registrador	R1	R2	R3	...
Conteúdo	19	2	165	...

Ou seja, pode-se pensar na RAM como uma função bijetora

$$M^* : Reg \to \mathbb{N},$$

onde $Reg = \{R_1, R_2, \ldots, R_n, \ldots\}$.

2.1.2 Programas RAM

Os programas RAM são abstrações de programas na linguagem assembly. Em outras palavras, a linguagem de programação da RAM é um conjunto bastante reduzido de instruções básicas de uma linguagem assembly real. Cada programa RAM é uma seqüencia finita de instruções, que referencia apenas uma quantidade finita de registradores.

Definição 2.1 *Sejam* $\Sigma = \{\texttt{N0}, \texttt{N1}, \texttt{N2}, \dots\}$ *um conjunto enumerável de* ***rótulos*** *e* $R = \{\texttt{R1}, \texttt{R2}, \dots\}$ *um conjunto enumerável de* ***nomes de registradores****. O conjunto das* **instruções RAM**, $\mathcal{I}$, *é o menor conjunto cujos elementos são definidos como segue: para todo* $i \in \mathbb{N}$,

1. `INC Ri` — *incrementa o conteúdo do registrador* R_i *em uma unidade;*
2. `DEC Ri` — *decrementa o conteúdo do registrador* R_i *em uma unidade. Se o conteúdo de* `Ri` *é* 0*, o valor permanece inalterado;*
3. `CLR Ri` — *coloca* 0 *no registrador* R_i*;*
4. `Ri <- Rj` — *substitui o conteúdo do registrador* R_i *pelo conteúdo do registrador* R_j*. O conteúdo de* R_j *permanece inalterado.*
5. `JMP Nia` — *executa a próxima instrução com rótulo* `Ni`*, imediatamente precedente a instrução atual. Caso* `Ni` *não seja um rótulo do programa, então a máquina pára*[*]*;*
6. `JMP Nib` — *executa a próxima instrução com rótulo* `Ni`*, imediatamente posterior a instrução atual. Caso* `Ni` *não seja um rótulo do programa, então a máquina pára*[†]*;*
7. `Rj JMP Nia` — *executa a instrução* `JMP Nia`*, caso o conteúdo do registrador* R_j *seja* 0*;*
8. `Rj JMP Nib` — *executa a instrução* `JMP Nib`*, caso o conteúdo do registrador* R_j *seja* 0*;*
9. `CONTINUE` — *faz nada.*
10. *Se* `Nj` $\in \Sigma$ *e* `m` $\in \mathcal{I}$*, então* `Nj m` $\in \mathcal{I}$*.*

Definição 2.2 *Um* ***programa RAM*** *é uma seqüência finita de instruções RAM*[‡]*.*

[*] Essa é uma das mudanças no modelo RAM em relação ao proposto em Smith [27], pois naquele modelo um programa RAM não admite "jumps" para rótulos inexistentes.

[†] Observe que esse tipo de desvio permite que se tenha rótulos repetidos no mesmo programa, e os "jumps" desviarão a execução para o rótulo mais próximo (acima ou abaixo, dependendo). Isso permitirá a junção de programas RAM sem a preocupação com os rótulos das instruções, pois é possível justapor programas que contenham os mesmos rótulos.

[‡] Aqui, um programa RAM é apenas uma seqüência de instruções básicas. Em outras variações da RAM, exige-se também que cada instrução `JMP` (condicional ou não) possua um destino válido (i.e. o rótulo referenciado no `JMP` faça parte do programa) e a instrução final seja `CONTINUE`; entretanto isso não será imposto aqui.

R1	R2	R3	R4	...	Próxima instrução
3	2	0	0	...	`N0 R2 JMP N1b`
3	2	0	0	...	`INC R1`
4	2	0	0	...	`DEC R2`
4	1	0	0	...	`JMP N0a`
4	1	0	0	...	`N0 R2 JMP N1b`
4	1	0	0	...	`INC R1`
5	1	0	0	...	`DEC R2`
5	0	0	0	...	`JMP N0a`
5	0	0	0	...	`N0 R2 JMP N1b`
5	0	0	0	...	`N1b CONTINUE`

Tabela 2.1: Simulação de um programa RAM

Exemplo 2.3 *O programa que segue implementa a função* `SOMA` [§].

```
N0 R2 JMP N1b
   INC R1
   DEC R2
   JMP N0a
N1 CONTINUE
```

Definição 2.4 *Para executar uma* **computação na RAM**, *é necessário que se forneça para máquina uma* ***configuração inicial de memória***, *sobre a qual a RAM executará as instruções do programa em questão — i.e. é fornecida uma seqüência $a_1, a_2, \ldots, a_n$ de números naturais nos registradores $R_1, R_2, \ldots, R_n$. Assume-se que os demais registradores possuem valor igual a zero. Se P consiste de s instruções $I_1, I_2, \ldots, I_s$, então a RAM começa obedecendo a instrução I_1, então I_2, I_3, e assim por diante, a menos que uma instrução* `JMP` *seja encontrada, o que fará com que a máquina obedeça a instrução de acordo com o que foi descrito anteriormente. A computação da RAM* ***pára*** *se ela executa a instrução final* `CONTINUE` *ou se ela executa um "jump" para um rótulo que não existe no programa. Um* ***passo de computação*** *na RAM é a execução de uma simples instrução. Um* ***estado*** *da RAM durante uma computação, é o par $\sigma = (c, j)$, onde c é a configuração de memória atual e j o próximo passo a ser executado. Observe que um passo de computação altera o estado atual da RAM* [¶].

A tabela 2.1, descreve o comportamento do estado da RAM durante a computação do programa acima para as entradas 3 e 2. Observe que se for considerado que o programa acima computa uma função $f : \mathbb{N}^2 \to \mathbb{N}$, então o resultado esperado encontra-se no registrador `R1`, e portanto o programa computa a função

[§] Essa função é básica na maioria das linguagens assembly reais, e aqui ela é construída a partir de instruções mais primitivas.

[¶] Não necessariamente o conteúdo da memória.

soma. Entretanto, convenciona-se que um mesmo programa RAM computa funções $f : \mathbb{N}^m \to \mathbb{N}^n$, para $m, n \geq 0$, onde o resultado contido em `R1`,..., `Rn` após o final de uma computação é interpretado como $f(x_1, \ldots, x_m)$. Essa abordagem é devido ao fato de se querer tratar funções computáveis da forma $f : \mathbb{N}^m \to \mathbb{N}^n$, em vez de apenas funções da forma $f : \mathbb{N}^m \to \mathbb{N}$ (como na abordagem tradicional). Ao, simplesmente, interpretar o conteúdo de n registradores como o resultado de uma função $f : \mathbb{N}^m \to \mathbb{N}^n$, aplicada a m argumentos, evita-se que se precise alterar a arquitetura da RAM estabelecendo uma área de memória para dados de entrada e outra para dados de saída, ou adicionar instruções de entrada e saída aos programas RAM, mantendo dessa forma a estrutura original da RAM.

Exemplo 2.5 *Assim, se o programa:*

```
NO R3 <- R1
   R1 <- R2
   R2 <- R3
   CONTINUE
```

for interpretado como a implementação de uma função $f : \mathbb{N}^2 \to \mathbb{N}$, então ele computa a função segunda projeção $U_2^2(x, y) = y$. Caso ele seja interpretado como a implementação de uma função $f : \mathbb{N}^2 \to \mathbb{N}^2$, então a função calculada é a função permutação $f(x, y) = (y, x)$, e assim por diante.

Mais precisamente, pode-se definir o seguinte:

Definição 2.6 *Um **programa RAM** P **computa uma** **função parcial** $f : \mathbb{N}^m \to \mathbb{N}^n$ $(m, n \geq 0)$, se, e somente se, quando P é iniciado com $\{x_1, \ldots, x_m\}$ nos registradores `R1`, ..., e `Rm`, respectivamente, e todos os demais registradores usados por P contém 0[‖], P pára se, e somente se, $f(x_1, \ldots, x_m)$ está definido e os registradores `R1`, ..., `Rn` contém valores $y_1, \ldots, y_n$, respectivamente, onde $f(x_1, \ldots, x_m) = (y_1, \ldots, y_n)$[**].*

Exemplo 2.7

1. *Seja $SOMA(x, y)$ o programa do exemplo 2.3, então esse programa computa a função $x + y$*
2. *Menos evidente, o programa `CONTINUE` computa as seguintes funções:*
   (a) *$z : \{\emptyset\} \to \mathbb{N}$, onde $z(\emptyset) = 0$. Nesse caso $m = 0$ e portanto o conjunto $\{x_1, \ldots, x_m\}$ da definição é vazio. Assim os "demais registradores usados por P" começam a partir de $R1$. Ou seja, todos os registradores começam com zero e portanto o programa termina com zero no registrador $R1$.*

[‖] Note que no caso de $m = 0$ todos os registradores são inicializados em zero.

[**] Note que no caso de $n = 0$, P computa f se: P pára para a entrada $x_1, \ldots, x_m$ se e somente se $f(x_1, \ldots, x_m)$ está definido.

(b) $\pi : \mathbb{N} \to \{\emptyset\}$*, onde* $\pi(x) = \emptyset$*. Como* `CONTINUE` *pára, para qualquer que seja o conteúdo do registrador* $R1$*,* $\pi(x)$ *está definido e* $\pi(x) = \emptyset$*, então* `CONTINUE` *computa a função* π*.*

(c) $id : \{\emptyset\} \to \{\emptyset\}$*, onde* $id(\emptyset) = \emptyset$*. Como* $\{\emptyset\} = \mathbb{N}^0$ *então* $m = n = 0$ *e portanto este caso é de certa forma a junção dos anteriores, pois o conjunto* $\{x_1, \ldots, x_n\}$ *da definição é vazio,* $id(\emptyset)$ *está definido e* $id(\emptyset) = \emptyset$*. Note que para o caso em que* $m \geq 1$*, o programa* `CONTINUE` *computa também a função* $id : \mathbb{N}^m \to \mathbb{N}^m$*, onde* $id(x_1, \ldots, x_m) = (x_1, \ldots, x_m)$*.*

Observe que para cada $m, n \geq 0$, um programa RAM computa exatamente uma função $f : \mathbb{N}^m \to \mathbb{N}^n$.

Definição 2.8 *Quando existe um programa* P *que computa uma função* $f : \mathbb{N}^m \to \mathbb{N}^n$*,* f *chama-se* ***RAM-computável****. Dessa forma, denota-se por* $\mathbb{RAM}$*, a* ***classe de todas as funções RAM-computáveis****. Sempre que uma computação pára para* $x_1, \ldots, x_n$ *diz-se que ela* ***converge para*** $x_1, \ldots, x_n$*, caso contrário que ela* ***diverge para*** $x_1, \ldots, x_n$*.*

Observe que afirmar que um programa P diverge para $x_1, \ldots, x_n$ é o mesmo que afirmar que ele entra em loop para $x_1, \ldots, x_n$ e que $f(x_1, \ldots, x_n)$ está indefinido quando P computa f. Por questão de brevidade, introduz-se as seguintes notações:

Notação 2.9 *Seja* P *um programa e* $x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{N}$

1. $P(x_1, \ldots, x_m) \downarrow$*, significa: o programa* P *converge para as entradas* $x_1, \ldots, x_m$*;*
2. $P(x_1, \ldots, x_m) \downarrow (y_1, \ldots, y_n)$*, significa: o programa* P *converge para as entradas* $x_1, \ldots, x_m$ *e retorna, respectivamente,* $y_1, \ldots, y_n$ *nos registradores* `R1`*,* $\ldots$*, e* `Rn`*;*
3. $P(x_1, \ldots, x_m) \uparrow$*, significa: o programa* P *diverge para as entradas* $x_1, \ldots, x_m$*.*

Dessa forma, pode-se afirmar que $SOMA(3, 2) \downarrow 5$, ao passo que o programa $SQRT$ que segue e calcula $\sqrt{x}$ converge somente para os quadrados perfeitos, ou seja $SQRT(4) \downarrow 2$, enquanto que $SQRT(3) \uparrow$. Porém, antes de apresentar $SQRT$ apresenta-se outras funções RAM-computáveis.

É conveniente ser capaz de abreviar uma seqüência de programas RAM que serão re-utilizados. Para ilustrar, a re-utilização do programa $SOMA$, dentro de um outro programa, teria o seguinte formato:

```
Rj <- Ru + Rv
```
(2.2)

Essa abreviação, resume o seguinte trecho de programa:

```
R1 <- Ru
R2 <- Rv
execução do programa SOMA(R1,R2);
Rj <- R1
CONTINUE
```

Em geral, uma vez que uma função $f : \mathbb{N}^m \to \mathbb{N}^n$ tenha sido demonstrada como RAM-computável é possível construir seqüências de programas RAM, dentro de qualquer programa, que calcularão as funções cujos argumentos estejam nos registradores $R_{k1}, \ldots, R_{km}$ e o resultado seja colocado nos registradores $R_{p1}, \ldots, R_{pn}$ para que em seguida o controle de execução retorne para a próxima instrução do programa mais externo. Assim, dada uma função $f : \mathbb{N}^m \to \mathbb{N}^n$ RAM-computável, a seqüência de instruções associada a f que será re-utilizada dentro de um programa P pode ser representada por

$$(R_{p1}, \ldots, R_{pn}) \leftarrow f(R_{k1}, \ldots, R_{km}) \tag{2.3}$$

Isso não deve ser interpretado como uma abreviação exata do código RAM associado, mas como um código convenientemente adaptado para utilizar registradores de entrada e de saída [††], não necessariamente todos distintos, e que retorne o controle de execução para a próxima instrução após a função ter sido calculada, a menos que f seja indefinida para os valores contidos em $R_{k1}, \ldots, R_{km}$, neste caso o controle jamais retornará à instrução após. Seguindo essa convenção, apresenta-se as seguintes funções RAM-computáveis. O nome atribuído aos programas referem-se as funções da forma $f : \mathbb{N}^m \to \mathbb{N}$ que são calculadas por eles.

Exemplo 2.10

1. $SQR(x) = x^2$

```
   R4 <- R1
   R3 <- R1
N0 DEC R3
   R3 JMP N1b
   R2 <- R4
   R1 <- R1 + R2
   JMP N0a
N1 CONTINUE
```

2. $SUB(x, y) = x \dot{-} y = \begin{cases} x - y & \text{se } x \geq y, \\ 0 & \text{caso contrário} \end{cases}$

```
 N0 R2 JMP N1b
    DEC R1
    DEC R2
    JMP N0a
 N1 CONTINUE
```

[††] i.e. com as devidas alterações dos nomes de registradores.

3. $DIST(x, y) =\mid x - y \mid$

```
R3 <- R1
R4 <- R2
R1 <- SUB(R1,R2)
R5 <- R1
R1 <- R4
R2 <- R3
R1 <- SUB(R1,R2)
R6 <- R1
R1 <- R5
R2 <- R6
R1 <- R1 + R2
CONTINUE
```

4. $SQRT(x) = \sqrt{x}$

```
R2 <- 0   % inicializa variável de teste
R4 <- 0  % inicializa variável de busca

N0 R3 <- DIST(R1,R2)
R3 JMP N1b % testa para verificar se achou ou não o valor desejado
   INC R4 % incrementa a variável de busca
   R2 <- SQR(R4)
   JMP N0a
N1 R1 <- R4
   CONTINUE
```

Imitando o que foi feito na tabela 2.1, onde foi feita a simulação do comportamento do programa soma, o leitor pode verificar que o programa `SQRT` pára quando o valor inicial do registrador `R1` é um quadrado perfeito, por exemplo 4, e retorna o valor da raiz quadrada no registrador `R1`, ao passo que o programa entra em loop quando o valor inicial do registrador `R1` não é um quadrado perfeito, por exemplo 3. Isso indica que existem funções RAM computáveis cujos programas que as implementam não param para qualquer entrada, mas param justamente para os pontos em que a função está definida.

2.2 Exercícios

1. Verifique se as seguintes funções numéricas são RAM-computáveis:

   (a) $x \times y$;

   (b) $x = 3$;

   (c) x^y

(d) $sg(x) = \begin{cases} 1 & \text{se } x \neq 0, \\ 0 & \text{se } x = 0 \end{cases}$

(e) $\overline{sg} = \begin{cases} 0 & \text{se } x > 0, \\ 1 & \text{se } x = 0 \end{cases}$

(f) $x!$

(g) $min(x, y)$

(h) $max(x, y)$

(i) $rm(x, y)$

(j) $qt(x, y)$

2. Verifique se as funções características $\chi : \mathbb{N}^m \to \mathbb{N}$ associada aos seguintes predicados são RAM-computáveis:

   (a) $x \leq y$;
   (b) x é um quadrado perfeito;
   (c) x é par;
   (d) x é divisível por 5. e
   (e) $x \mid y$ — x é divisível por y.

2.3 Considerações finais

O leitor pode ter percebido que as máquinas RAM são capazes de calcular certas funções numéricas. A questão é:

1. Que tipo de funções ela é capaz de calcular?
2. Existem outras funções que não são capazes de serem calculadas por máquinas RAM?

A primeira questão é respondida no próximo capítulo enquanto que a segunda só será respondida no capítulo 6. Observe que a resposta da última questão será capaz de impor ou não um limite no poder de computação das RAM. Caso esse limite seja imposto, será que ele, como conseqüência, também será limite para toda a computação? Esta questão é respondida no capítulo 5.

No capítulo seguinte apresenta-se uma maneira alternativa para demonstrar que uma determinada função é RAM computável. Por questão de simplicidade, a demonstração da computabilidade de certas funções que serão importantes para o desenvolvimento deste curso será adiada para o capítulo 6.

Capítulo 3

Funções Recursivas Parciais

No capítulo anterior, verificou-se que as funções RAM computáveis são funções da forma $f : \mathbb{N}^m \to \mathbb{N}^n$. Nesse capítulo caracteriza-se uma classe de funções RAM-computáveis. Posteriormente se provará que ela é exatamente a classe de todas as funções RAM-computáveis. Essa classe, chamada *classe das funções recursivas parciais*, é gerada a partir de um conjunto finito de funções básicas e é algebricamente fechada sob certos funcionais. Dessa forma, se concluirá mais adiante que as funções RAM-computáveis apresentadas anteriormente poderão ser expressas através de termos formados a partir das funções básicas mais esses funcionais.

A definição que segue é um pouco diferente da usual, principalmente no que diz respeito as funções básicas. Isso se deve ao fato de se estar lidando com funções da forma $f : \mathbb{N}^m \to \mathbb{N}^n$, onde $m, n \geq 0$. Mais adiante apresenta-se um processo, chamado numeração de Gödel, que é quem fundamenta a abordagem geralmente utilizada em vários textos de funções recursivas ou teoria da computação (c.f. Cutland[8], Peter [23], e Smith [27])*.

3.1 Funções básicas

Definição 3.1 *Seja $\mathbb{N}$ o conjunto dos números naturais†. Chamam-se* ***funções recursivas básicas*** *qualquer uma das seguintes funções:*

1. ***função zero:*** $z : \mathbb{N}^0 \to \mathbb{N}$, *onde* $z(\emptyset) = 0$.
2. ***função terminal:*** $\pi : \mathbb{N} \to \mathbb{N}^0$, *onde* $\pi(x) = \emptyset$.
3. ***funções identidade:*** $id^m : \mathbb{N}^m \to \mathbb{N}^m$, *tal que* $id^m(\vec{x}) = \vec{x}$, *para* $m \geq 0$.
4. ***função sucessor:*** $s : \mathbb{N} \to \mathbb{N}$, *tal que* $s(x) = x + 1$.

*Nesses textos são consideradas funções da forma $f : \mathbb{N}^m \to \mathbb{N}$, onde $m \geq 1$.

†Para uma definição rigorosa de $\mathbb{N}^m$ $(m \geq 0)$ veja o capítulo 1.

5. ***função i-ésima projeção:*** $U_i^m : \mathbb{N}^{i_1} \times \ldots \times \mathbb{N}^{i_m} \rightarrow \mathbb{N}^{i_{m+1}}$, *onde cada* $i_j \in \{0,1\}$ *com* $j = 1, \ldots, m+1$ *e* $m \geq 1$, $U_i^m(x_1, \ldots, x_m) = x_i$, *e* $1 \leq i \leq m$.

Lema 3.2 *Toda função recursiva básica é RAM-computável.*

Prova: O programa `CONTINUE` computa as funções zero, terminal, e identidade (Veja o exemplo 2.7). Os seguintes programas computam as funções sucessor e i-ésima projeção, respectivamente:

1.
```
INC R1
CONTINUE
```

2.
```
R1 <- Ri
CONTINUE
```

□

Funções mais complexas, podem ser expressas a partir da aplicação dos funcionais que seguem.

3.2 Produto

Como este texto leva em consideração funções cujo contra-domínio é da forma $\mathbb{N}^k$, $k \geq 0$, é necessário um operador capaz de criar n-uplas de números naturais, permitindo a síntese de funções.

Definição 3.3 (Produto de funções) *Sejam as funções* $f : \mathbb{N}^m \rightarrow \mathbb{N}^n$ *e* $g : \mathbb{N}^p \rightarrow \mathbb{N}^q$. *A função* $f \times g : \mathbb{N}^{m+p} \rightarrow \mathbb{N}^{n+q}$ *definida por*

$$f \times g(x_1, \ldots, x_m, x_{m+1}, \ldots, x_{m+p}) = (y_1, \ldots, y_n, y_{n+1}, \ldots, y_{n+q}), \tag{3.1}$$

onde $(y_1, \ldots, y_n) = f(x_1, \ldots, x_m)$ *e* $(y_{n+1}, \ldots y_{n+q}) = g(x_{m+1}, \ldots, x_{m+p})$ *chama-se* ***produto de*** f ***e*** g. *Quando* $\vec{x} = (x_1, \ldots, x_n)$ *e* $\vec{y} = (y_1, \ldots, y_p)$, *abrevia-se (3.1) por* $f \times g(\vec{x}, \vec{y}) = (f(\vec{x}), g(\vec{y}))$.

Exemplo 3.4 *Sejam as funções* $Soma : \mathbb{N}^2 \rightarrow \mathbb{N}$ *e* $Fac : \mathbb{N} \rightarrow \mathbb{N}$ *(soma e fatorial respectivamente). Com o operador produto, é possível obter a função* $Soma \times Fac : \mathbb{N}^3 \rightarrow \mathbb{N}^2$, *tal que* $Soma \times Fac(x, y, z) = (x + y, z!)$.

Observe que para funções em que o domínio ou o contradomínio é $\mathbb{N}^0$ a definição não se altera. Por exemplo, dadas as funções s e π, a função $s \times \pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}^0$ é definida por $s \times \pi(x, y) = (x + 1, \emptyset)$. Note que o caso em que alguns textos clássicos de computabilidade (por exemplo [4]) falam que $\mathbb{N}^m \times \mathbb{N}^0 = \mathbb{N}^{m+0} = \mathbb{N}^m$, é na

verdade um abuso de linguagem, pois os elementos de $\mathbb{N}^m \times \mathbb{N}^0$ são da forma $(\vec{x}, \emptyset)$, onde $\vec{x} \in \mathbb{N}^m$, enquanto os elementos de $\mathbb{N}^m$ são da forma $\vec{x}$, ferindo portanto o axioma da extensão da teoria dos conjuntos[‡]. Entretanto, esse abuso de linguagem decorre do fato de que existe uma bijeção entre os conjuntos $\mathbb{N}^m \times \mathbb{N}^0$ e $\mathbb{N}^m$. Esse abuso de linguagem pode ser estendido para funções. Por exemplo, a função $s \times \pi$ pode ser pensada como sendo a função $s' : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definida por $s'(x, y) = x + 1$. Isso só ocorre devido às funções $id^2 : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ e $U_1^2 : \mathbb{N} \times \mathbb{N}^0 \to \mathbb{N}$ da definição 3.1 serem bijeções e o seguinte diagrama comutar.

$$\begin{array}{ccc} \mathbb{N} \times \mathbb{N} & \xrightarrow{s \times \pi} & \mathbb{N} \times \mathbb{N}^0 \\ {\scriptstyle id^2}\downarrow & & \downarrow{\scriptstyle U_1^2} \\ \mathbb{N} \times \mathbb{N} & \xrightarrow{s'} & \mathbb{N} \end{array}$$

Generalizando, funções do tipo $f : \mathbb{N}^{i_1} \times \cdots \times \mathbb{N}^{i_m} \to \mathbb{N}^{j_1} \times \cdots \times \mathbb{N}^{j_n}$, com $i_1, \ldots, i_m, j_1, \ldots, j_n \in \{0, 1\}$, serão tratadas, a menos que se explicite o contrário, como a função $f' : \mathbb{N}^{m'} \to \mathbb{N}^{n'}$ definida por $f' = \beta \circ f \circ \alpha$, onde $m' = \sum_1^m i_k$ e $n' = \sum_1^n j_k$, i.e. m' e n' são a quantidade de uns que ocorrem em $i_1, \ldots, i_m$ e $j_1, \ldots, j_n$, respectivamente, e α e β são as bijeções naturais entre $\mathbb{N}^{m'}$ e $\mathbb{N}^{i_1} \times \cdots \times \mathbb{N}^{i_m}$ e entre $\mathbb{N}^{j_1} \times \cdots \times \mathbb{N}^{j_n}$ e $\mathbb{N}^{n'}$, respectivamente. Por exemplo a função $f : \mathbb{N} \times \mathbb{N}^0 \times \mathbb{N}^0 \times \mathbb{N} \times \mathbb{N}^0 \to \mathbb{N}^0 \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}^0 \times \mathbb{N}$ definida por $f(x, \emptyset, \emptyset, y, \emptyset) = (\emptyset, y, x, \emptyset, y)$ é identificada com a função $f'(x, y) = (y, x, y)$, onde nesse caso $\alpha(x, y) = (x, \emptyset, \emptyset, y, \emptyset)$ e $\beta(\emptyset, x, y, \emptyset, z) = (x, y, z)$.

Observação: A noção de função RAM-computável pode ser estendida para funções do tipo $f : \mathbb{N}^{i_1} \times \cdots \times \mathbb{N}^{i_m} \to \mathbb{N}^{j_1} \times \cdots \times \mathbb{N}^{j_n}$ com $i_1, \ldots, i_m, j_1, \ldots, j_n \in \{0, 1\}$, da seguinte forma: f é RAM-computável (estendido) se e somente se f' é RAM-computável.

Observação: Daqui em diante, só consideraremos funções do tipo $f : \mathbb{N}^m \to \mathbb{N}^n$ com $m, n \geq 0$.

Lema 3.5 *Se $f : \mathbb{N}^m \to \mathbb{N}^n$ e $g : \mathbb{N}^p \to \mathbb{N}^q$ são RAM-computáveis, então a função $f \times g$ também é RAM-computável.*

Prova: Suponha que f e g são funções RAM-computáveis. Sejam `Pf` e `Pg` os programas que computam f e g respectivamente. Suponha que nenhum registrador R_j é referenciado por esses programas para qualquer $j > k$. Assumindo que os registradores `R1` a `Rm` contém inicialmente $x_1, \ldots, x_m$ e os registradores `Rm+1` a `Rm+p` contém $x_{m+1}, \ldots, x_{m+p}$, então o seguinte programa computa $f \times g$:

```
N0 Rk+1 <- Rm+1 % Salva os argumentos da função g
   etc
```

[‡]Dois conjuntos A e B são iguais se eles possuem os **mesmos** elementos [14].

```
   Rk+p <- Rm+p
   CLR Rm+1
   etc
   CLR Rk
N1 Pf           % Calcula f(x_1,...,x_m)
   Rk+p+1 <- R1 % Salva f(x_1,...,x_m)
   etc
   Rk+p+n <- Rn
N2 R1 <- Rk+1 % Recupera os argumentos da função g
   etc
   Rp <- Rk+p
   CLR Rp+1
   etc
   CLR Rk
N3 Pg           % Calcula g(x_1,...,x_p)
   Rk+p+n+1 <- R1 % Salva g(x_1,...,x_p)
   etc
   Rk+p+n+q <- Rq
N4 R1 <- Rk+p+1 % Recupera (f(x_1,...,x_m),g(x_1,..., x_p))
   etc
   Rn <- Rk+p+n
   Rn+1 <- Rk+p+n+1
   etc
   Rn+q <- Rk+p+n+q
N5 CONTINUE
```

Dessa forma, quando funções RAM-computáveis são combinadas via produto, o resultado será uma função RAM-computável. □

3.3 Composição

A composição de funções é uma operação conhecida da matemática. Aqui ela é generalizada para ser aplicada à várias funções computáveis.

Definição 3.6 *Uma função* $h : \mathbb{N}^n \to \mathbb{N}^p$ *é definida por* ***composição*** *ou* ***substituição*** *a partir de* m *funções* $g_1 : \mathbb{N}^n \to \mathbb{N}^{p1}, \ldots, g_m : \mathbb{N}^n \to \mathbb{N}^{pk}$ *e uma função* $f : \mathbb{N}^t \to \mathbb{N}^p$*, onde* $t = p1 + p2 + \cdots + pk$*, se e somente se:*

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \tag{3.2}$$

ou seja $h(x_1, \ldots, x_n) = f(y_1^{q_1}, \ldots, y_{p1}^{q_1}, \ldots, y_1^{q_k}, \ldots, y_{pk}^{q_k})$*, onde para todo* $i = 1, \ldots, mk$*,* $q_i = g_i(x_1, \ldots, x_n)$*.*

Quando $m = 1$*, obtém-se a definição usual:*

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n)). \tag{3.3}$$

e escreve-se $h(x_1, \ldots, x_n) = f \circ g_1(x_1, \ldots, x_n)$.

Alguns livros como o de Kleene [19] em vez de h utilizam a notação $S^n_m(f, g_1, \ldots, g_m)$ para enfatizar a existência de um operador de composição sobre funções. Entretanto, por questões de simplicidade isso será evitado.

Exemplo 3.7 *Sejam as funções de subtração natural,* $SUB(x, y) = x \dot{-} y$*, vista no capítulo anterior, e a primeira projeção:* $U^2_1(x, y) = x$*. A expressão* $min(x, y) = SUB(U^2_1(x, y), SUB(x, y))$ *é definida por composição e descreve a função que calcula o mínimo entre* x *e* y [§].

Observe que em vez de escrever explicitamente a projeção na expressão: $min(x, y) = SUB(U^2_1(x, y), SUB(x, y))$, pode-se omiti-la afim de obter uma expressão mais simples como: $x \dot{-} (x \dot{-} y)$. Seguindo esse princípio, daqui por diante omite-se sempre que for possível o aparecimento das projeções.

Exemplo 3.8 *O valor absoluto de* $x - y$ *pode ser expresso por composição como:*

$$| x - y | = (x \dot{-} y) + (y \dot{-} x)$$

Como na substituição os argumentos em g são repetidos, podemos usar esta operação para obter funções de $\mathbb{N}^m \to \mathbb{N}^n$ com $m < n$.

Exemplo 3.9 *A função* $f : \mathbb{N} \to \mathbb{N}^2$ *definida por* $f(x) = (x, x)$ *é obtida como* $f(x) = id^2(id^1(x), id^1(x))$*, i.e.*

$$f = S^1_2(id^2, id^1, id^1)$$

Já a função $f : \mathbb{N} \to \mathbb{N} \times \mathbb{N}^0$ *definida por* $f(x) = (x, \emptyset)$ *é obtida como* $f(x) = id^1 \times id^0(id^1(x), \pi(x))$*, i.e.*

$$f = S^1_2(id^1 \times id^0, id^1, \pi)$$

Note que esta última função é a inversa da função bijeção U^2_1 *entre* $\mathbb{N} \times \mathbb{N}^0$ *e* $\mathbb{N}$.

Lema 3.10 *Se as funções* $g_1 : \mathbb{N}^n \to \mathbb{N}^{p1}, \ldots, g_m : \mathbb{N}^n \to \mathbb{N}^{pm}$*, e* $f : \mathbb{N}^t \to \mathbb{N}^p$ *(onde* $t = p1 + p2 + \cdots + pm$*) são RAM computáveis, então a função* $h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$*, também é RAM-computável.*

Prova: Suponha que h é uma função de n argumentos definida por composição a partir de $f, g_1, \ldots,$ e g_m. Sejam `Pf, P1,...,Pm` os programas que computam f, $g_1, \ldots,$ e g_m respectivamente. Seja k um número natural tal que nenhum registrador R_j com $j > k$ é referenciado por qualquer desses programas. O seguinte programa computa h:

[§] Se a notação de Kleene [19] fosse utilizada, isso seria explicitamente descrito como $min(x, y) = S^2_2(U^2_1, SUB)(x, y)$.

```
N0 Rk+1 <- R1 % Salva os argumentos das funções g
   Rk+2 <- R2
   etc.
   Rk+n <- Rn
N1 P1   % Computa a função g1 sobre os argumentos R1 à Rn
   Rk+n+1 <- R1 % Assumindo que (y1,...,yq) é o resultado
                % da computação de P1. Este trecho guarda
                % esse resultado nos registradores Rk+n+1 a Rk+n+q
   etc.
   Rk+n+q <- Rq
N2 R1 <- Rk+1 % Recupera os argumentos das funções g
   R2 <- RK+2
   etc.
   Rn <- Rk+n
   CLR Rn+1 % Limpa os registr. utilizados durante a computação
   CLR Rn+2
   etc.
   CLR Rk
   P2   % Computa a função g2 sobre os argumentos R1 a Rn
   Rk+n+(q+1) <- R1 %  Assumindo que (z1,..., zp2) é o resultado
                    % da computação de P2, este trecho guarda esse
                    % resultado nos registradores Rk+n+(q+1) a Rk+n+(q+p2)
   etc.
   Rk+n+(q+p2) <- Rp2
N3 % Assim como anteriormente recupera-se os argumentos,
   % limpa-se os demais registradores e executa P3. Esse
   % processo é, então, repetido até a execução do programa Pm.

Nm R1 <- Rk+1 % Recupera os argumentos das funções g
   R2 <- Rk+2
   etc.
   Rn <- Rk+n
   CLR Rn+1 % Limpa os registr. utilizados durante a computação
   CLR Rn+2
   etc.
   CLR Rk
   Pm   % Computa a função gm sobre os argumentos R1 a Rn
   Rk+n+(j+1) <- R1 % Assumindo que Rk+n+(j+1) é o primeiro registrador
                    % não utilizado até este ponto, e que (w1,...,wu) é o
                    % resultado da computação de Pm, esse trecho guarda esse
                    % resultado nos registradores Rk+n+(j+1),...,Rk+n+(j+u)
   Rk+n+(j+u) <- Ru
Nm+1 R1 <- Rk+n+1 % recupera o resultado de P1
      etc.
     Rq <- Rk+n+q
Nm+2 Rq+1 <- Rk+n+(q+1) % recupera o resultado de P2
      etc.
     Rq+t <- Rk+n+(q+p2)
     etc..
```

```
        Rj+1 <- Rk+n+(j+1)
        etc.
        Rj+u <- Rk+n+(j+u)
        CLR Rj+(u+1) % limpa o restante dos registradores
        etc.
        CLR Rk+n+(j+u)
Nv      CONTINUE
```

Dessa forma, quando funções RAM-computáveis são combinadas via composição, o resultado será uma função RAM-computável. □

3.4 Operador de recursão primitiva

Recursão é um método para definir funções que descreve como uma função retorna valores a partir de resultados previamente obtidos. Pode-se pensar que o valor da função para um determinado argumento é "construído" passo-a-passo.

As progressões geométricas e aritméticas são exemplos de funções definidas por recursão. Por exemplo, na PA $a_0 = 5$ e $a_{n+1} = a_n + 4$, o valor a_2 é obtido calculando-se primeiramente a_0, em seguida a expressão $a_1 = a_0 + 4$, e finalmente a expressão $a_2 = a_1 + 4$. Note que houve a repetição do processo " adicione 4" em cada resultado previamente obtido até a_2 ser alcançado. Mais geralmente, temos a seguinte definição:

Definição 3.11 *Sejam $g : \mathbb{N}^m \to \mathbb{N}^s$ e $h : \mathbb{N}^{m+1+s} \to \mathbb{N}^s$ funções arbitrárias. Então a função $f : \mathbb{N}^{m+1} \to \mathbb{N}^s$ é definida por* ***recursão primitiva*** *a partir de g e h, se*

$$\begin{cases} f(\vec{x}, 0) = g(\vec{x}) \\ f(\vec{x}, y) = h[\vec{x}, y, f(\vec{x}, y \dot{-} 1)], \quad \textit{para } y > 0; \end{cases} \tag{3.4}$$

onde $\vec{x} = (x_1, \ldots, x_m)$. Segundo Kleene [19], isso pode ser designado por $R^k(g, h)$, onde $k = m+1$, entretanto por questão de simplicidade evita-se essa notação sempre que possível. As equações acima são conhecidas como ***equações de recursão****.*

Observe a possibilidade da existência de circularidade na segunda equação. Note ainda, que quando há circularidade, a cada passo, $f(\vec{x}, y)$ solicita uma avaliação $f(\vec{x}, y \dot{-} 1)$, assegurando que se todas as funções estiverem definidas durante o processo de avaliação, em um número finito de passos o valor de $f(\vec{x}, y)$ será obtido.

Quando $n = 0$ as equações de recursão possuem a seguinte forma:

$$\begin{cases} f(0) &= a \\ f(y) &= h(y, f(y \dot{-} 1)). \end{cases} \tag{3.5}$$

onde $a \in \mathbb{N}$.

Exemplo 3.12 *A funções que seguem estão definidas por recursão primitiva.*

1. $\begin{cases} SOMA(x,0) & = U_1^1(x) \\ SOMA(x,y+1) & = s(SOMA(x,y)) \end{cases}$

2. $\begin{cases} 0! & = 1 \\ (n+1)! & = (n+1)\cdot n! \end{cases}$

3. $\begin{cases} f(0) & = 1 \\ f(1) & = 2 \\ f(n) & = f(n-1) + 2f(n-2). \end{cases}$

Observe que as definições estão numa forma simplificada, onde não ocorre a presença de todos os símbolos de função necessários para a definição por recursão, pois a presença deles tornariam as expressões rebuscadas.

Lema 3.13 *Se $g : \mathbb{N}^m \to \mathbb{N}^s$ e $h : \mathbb{N}^{m+1+s} \to \mathbb{N}^s$ são funções RAM-computáveis, então a função $f : \mathbb{N}^{m+1} \to \mathbb{N}^s$ definida por recursão primitiva é RAM-computável.*

Prova: Suponha que as funções g e h do enunciado sejam RAM-computáveis, e que f esteja definida por recursão primitiva. Sejam P_g e P_h programas que computam g e h respectivamente. Assuma que nenhum desses programas referenciam qualquer registrador `Rj` (para $j \geq k$) e utilizem os labels `Nu` e `Nv`. Então o seguinte programa implementa o esquema de recursão primitiva.

```
    Rk <- Rm+1    % salva o valor de y em Rk
    Rk+1 <- R1    % Salva os argumentos de x1..xm
    Rk+2 <- R2
    etc.
    Rk+m <- Rm
    CLR Rk+m+1    % inicializa o número de iterações: y'
    Pg            % computa g(x1,...,xm)
    Rk+m+2 <- R1  % armazena o valor da função
    etc
    Rk+m+s+1 <- Rs
Nu  Rk JMP Nvb    % Testa a terminação do loop
    INC Rk+m+1    % Incrementa y'
    R1 <- Rk+1    % Recupera os argumentos x1..xm
    R2 <- Rk+2
    etc.
    Rm <- Rk+m
    Rm+1 <- Rk+m+1 % copia o valor atual de y'
    Rm+2 <- Rk+m+2 % copia o último valor calculado
    etc
    Rm+s+1 <- Rk+m+s+1
    CLR Rm+s+2 % Limpa os registr. utilizados durante a computação
    CLR Rm+s+3
    etc.
```

```
    CLR Rk-1
    Ph   % Computa a função h sobre os argumentos R1,...,Rm+s+1
    Rk+m+2 <- R1 % Guarda o resultado da computação de Ph
    etc
    Rk+m+s+1 <- Rs
    DEC Rk % decrementa o número de passos da computação
    JMP Nua
Nv CONTINUE
```

Dessa forma, o resultado de combinar funções RAM-computáveis via recursão primitiva produzirá uma função RAM-computável. □

Definição 3.14 (Exponenciação) *Seja uma função $f : \mathbb{N}^m \to \mathbb{N}^m$, a **exponenciação** de f, $f^{\#} : \mathbb{N}^{m+1} \to \mathbb{N}^m$, é definida por:*

$$f^{\#}(\vec{x}, y) = \overbrace{f(f(\ldots(f(\vec{x}))))}^{y \text{ vezes}}, \tag{3.6}$$

onde $f(\vec{x}, 0) = \vec{x}$, para todo $\vec{x}$.

Assim, para calcular $f^{\#}(\vec{x}, y)$, f é composta consigo mesma y vezes. No lugar de $f^{\#}(\vec{x}, y)$, usualmente escreve-se $f^y(x)$. Note que $f^y(x)$ estará definido somente se "$f(x), f(f(x)), \ldots$' estiverem todos definidos, i.e. $(x, y) \in dom\ f^{\#}$ se e somente se $f^k(x) \in dom\ f$, para $0 \leq k < y$.

Exemplo 3.15 *$SOMA(x, y) = s^{\#}(x, y) = s^y(x)$, onde $s(x) = x + 1$.*

Lema 3.16 *Seja $m, k1, k2 \in \mathbb{N}$ tais que $k_1 \leq k_2 \leq m$. A função $U^m_{[k1,k2]} : \mathbb{N}^{i_1} \times \ldots \times \mathbb{N}^{i_m} \to \mathbb{N}^{i_{k1}} \times \ldots \times \mathbb{N}^{i_{k2}}$, onde cada $i_j \in \{0, 1\}$ com $j = 1, \ldots, m+1$ e $m \geq 1$, definida por $U^m_{[k1,k2]}(x_1, \ldots, x_m) = (x_{k1}, \ldots, x_{k2}$ é primitiva recursiva.*

Prova: Trivialmente, $U^m_{[k1,k2]} = U^{k1}_{k1} \times id^{k2-k1-1} \times U^{m-k2+1}_{k2}$. □

Proposição 3.17 *Toda função definida por exponenciação pode ser definida por recursão primitiva.*

Prova: Sejam as funções: $f : \mathbb{N}^m \to \mathbb{N}^m$, $\vec{x} = (x_1, \ldots, x_m)$, $id(\vec{x}) = \vec{x}$ e $h = U^{m+1+m}_{[m+1+1,m+1+m]} \circ (\overbrace{\pi \times \cdots \times \pi}^{(m+1)-vezes} \times f)$, então as seguintes equações definem a exponenciação:

$$\begin{cases} f^{\#}(\vec{x}, 0) = id(\vec{x}) \\ f^{\#}(\vec{x}, y) = h[\vec{x}, y, f^{\#}(\vec{x}, y \dot{-} 1)], \text{ para } y > 0. \end{cases}$$

Ou seja, $f^{\#}(\vec{x},0) = \vec{x}$, e $f^{\#}(\vec{x},y) = f(f^{\#}(\vec{x},y \dot{-} 1))$ (para $y > 0$). □

A proposição acima permite que certas funções como a $SOMA$ sejam expressas mais facilmente através de exponenciação do que através recursão.

Definição 3.18 (Recursão primitiva) *A menor classe de funções sobre os naturais que contém as funções recursivas básicas e é fechada sob as operações de composição, produto e recursão é chamada* ***classe das funções recursivas primitivas****, aqui denotada por* $\mathbb{RP}$*. Em outras palavras, uma* ***função recursiva primitiva*** *ou é uma função básica ou ela é obtida exclusivamente utilizando a aplicação da composição, do produto ou da recursão.*

Observe que todo programa associado à uma função recursiva primitiva é um programa que pára para qualquer entrada, pois tanto as funções recursivas básicas como aquelas que são o resultado da aplicação da composição, do produto ou da recursão são funções totais.

No que segue demonstra-se que alguma funções bem conhecidas são recursivas primitivas, e por conseguinte computáveis.

Proposição 3.19 *As seguintes funções são recursivas primitivas:*

1. *Adição:* $x + y$
2. *Multiplicação:* $x \cdot y$
3. *Exponenciação:* x^y
4. *Predecessor:* $x \dot{-} 1$
5. *Subtração própria:* $x \dot{-} y$
6. *Valor absoluto da diferença:* $| x - y |$
7. *Mínimo de* x *e* y*:* $min(x,y)$
8. *Máximo de* x *e* y*:* $max(x,y)$

Prova: As provas que seguem utilizam os operadores de recursão e composição, as funções recursivas básicas e funções que já foram demonstradas serem recursivas primitivas.

1. usando recursão e sucessor:
$$\begin{cases} x + 0 & = x \\ x + (y+1) & = s(x+y). \end{cases}$$

2. usando recursão e (1):
$$\begin{cases} x \cdot 0 & = 0 \\ x \cdot (y+1) & = (x \cdot y) + x. \end{cases}$$

3. usando recursão e (2):
$$\begin{cases} x^0 & = 1 \\ x^{(y+1)} & = (x^y) \cdot x \end{cases}$$

4. usando recursão e projeção:
$$\begin{cases} 0 \dot{-} 1 & = 0 \\ (x+1) \dot{-} 1 & = x. \end{cases}$$

5. usando recursão e (4):
$$\begin{cases} x \dot{-} 0 & = x \\ x \dot{-} (y+1) & = (x \dot{-} y) \dot{-} 1. \end{cases}$$

6. usando composição (1) e (5): $\mid x - y \mid = (x \dot{-} y) + (y \dot{-} x)$
7. usando composição e (5): $min(x, y) = x \dot{-} (x \dot{-} y)$
8. usando composição e (5): $max(x, y) = x + (y \dot{-} x)$

□

Mais adiante apresenta-se outras funções recursivas primitivas importantes que serão utilizadas neste livro, mas antes de prosseguir é necessário introduzir um conceito necessário para aquelas funções e para alguns capítulos subseqüentes: o conceito de predicado. A seção seguinte introduz este conceito e relaciona-o com a noção de computação até aqui desenvolvida, dando origem a definição de decidibilidade.

3.5 Predicados e decidibilidade

Uma das noções importantes que é utilizada em matemática é a de predicado. Esta seção não apresenta uma definição lógico-formal, mas procura ser intuitiva.

Um **predicado** (de primeira ordem) é uma frase escrita numa certa linguagem e expressa uma propriedade que certos objetos dum determinado universo de discurso possuem. Por exemplo: Ao pensar na classe de todos os animais, a propriedade "x é humano" é uma propriedade de certos elementos desta classe que é expressa na linguagem do Português. A propriedade "x é um número primo" é outra propriedade expressa em Português que caracteriza certos elementos da classe dos números naturais. Dessa forma, ao estabelecer o conjunto dos seres humanos e dos números naturais, obtém-se a partir dos predicados anteriores, respectivamente, os subconjuntos dos seres humanos e dos números primos — que em notação matemática geralmente tem a forma: $\{x \in \text{ Animais} : x \text{ é humano}\}$ e $\{x \in \mathbb{N} : x \text{ é primo}\}$. Além disso, a substituição, por exemplo, de x por algum elemento do conjunto dos números naturais resulta em proposições como: "5 é um número primo" e "4 é um número primo", que são sentenças respectivamente verdadeira e falsa. Dessa forma, pode-se interpretar um predicado como uma função da seguinte forma:

Definição 3.20 *Dado um conjunto A e um subconjunto B, a* ***função característica*** *associada a B é da forma $C_B : A \to \mathbb{N}$, onde*

$$C_B(x) = \begin{cases} 1 & ,se\ x \in B \\ 0 & ,se\ x \notin B. \end{cases} \tag{3.7}$$

onde "0" representa o valor verdade "Falso" e "1" representa o valor verdade "Verdadeiro". Logo, se $P(x)$ é um predicado que descreve os elementos de B, a equação acima pode ser reescrita da seguinte forma:

$$C_B(x) = \begin{cases} 1, se\ P(x)\ é\ verdadeiro \\ 0, se\ P(x)\ não\ é\ verdadeiro. \end{cases} \tag{3.8}$$

Exemplo 3.21 *Seguindo os exemplos acima, a função característica associada ao conjunto dos números primos pode ser descrita como* $C_{Primos} : \mathbb{N} \to \mathbb{N}$*, onde*

$$C_{Primos}(x) = \begin{cases} 1, & \text{“se } x \text{ é primo”} \\ 0, & \text{“se } x \text{ não é primo”.} \end{cases}$$

Já a função característica $C_{\leq} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$*, onde*

$$C_{\leq}(x, y) = \begin{cases} 1, & \text{“se } x \text{ é menor ou igual a } y\text{”} \\ 0, & \text{“se não é o caso que } x \text{ é menor ou igual a } y\text{”.} \end{cases}$$

interpreta o predicado binário (que contém duas variáveis) “x é menor ou igual a y” que está definido sobre o conjunto de todos os pares de naturais $\mathbb{N} \times \mathbb{N}$*.*

Observe que, os predicados podem ser unários, binários, etc. Designa-se um predicado n-ário da seguinte forma $P(x_1, \ldots, x_n)$, onde P é apenas um nome da propriedade em questão. Como os predicados sobre os naturais são interpretados como funções sobre os naturais, então eles se tornam candidatos a serem implementados em programas RAM. Agora, será que todo predicado sobre os naturais pode ser implementado num programa RAM? Isso dá origem a seguinte definição:

Definição 3.22 *Um predicado n-ário* $P(x_1, \ldots, x_n)$ *é um* ***predicado decidível****, se sua função característica é RAM-computável. Se além disso, a função característica é recursiva primitiva, então ele é um* ***predicado recursivo primitivo****.*

3.6 Um pouco de recursão primitiva

Esta seção pretende ser uma referência e apresenta várias funções recursivas primitivas que são importantes.

Proposição 3.23 *O predicado “*$sg(x)$*: x é positivo” e a sua negação “*$\overline{sg}(x)$*” são recursivos primitivos. Além disso, a igualdade “*$x \approx y$*” e a sua negação “*$x \not\approx y$*”também são predicados recursivos primitivos.*

Prova: As seguintes funções recursivas parciais demonstram a proposição acima:

1. usando recursão:
$$\begin{cases} sg(0) & = 0 \\ sg(x+1) & = 1. \end{cases}$$
2. $\overline{sg}(x) = 1 \dot{-} sg(x)$.
3. $x \approx y = \overline{sg}(\mid x - y \mid)$.
4. $x \not\approx y = sg(\mid x - y \mid)$.

□

Proposição 3.24 (Divisibilidade) *As seguintes funções são recursivas primitivas:*

1. *$rm(x, y)$ — "o resto quando y é dividido por x".*
2. *$qt(x, y)$ — "o quociente quando y é dividido por x".*
3. *$div(x, y)$ — "x divide y".*

Convenções: *Para obter uma função total, convenciona-se que $rm(0, y) = y$ e $qt(0, y) = 0$. Observe que, $div(0, 0) = 1$ mas não é o caso que $div(0, y) = 0$ quando $y \neq 0$.*

Prova:

1. o resto pode ser definido como:

$$rm(x, y+1) = \begin{cases} rm(x,y)+1 & \text{se } rm(x,y)+1 \neq x. \\ 0 & \text{se } rm(x,y)+1 = x. \end{cases}$$

   e reescrito na seguinte equação recursiva:

$$\begin{cases} rm(x,0) & = 0. \\ rm(x,y+1) & = (rm(x,y)+1) \cdot sg(\mid x - (rm(x,y)+1) \mid). \end{cases}$$

2. visto que

$$qt(x, y+1) = \begin{cases} qt(x,y) & \text{se } rm(x,y)+1 \neq x. \\ qt(x,y)+1 & \text{se } rm(x,y)+1 = x. \end{cases}$$

   tem-se a seguinte definição recursiva:

$$\begin{cases} qt(x,0) & = 0. \\ qt(x,y+1) & = qt(x,y) + \overline{sg}(\mid x - (rm(x,y)+1) \mid). \end{cases}$$

3. como

$$\begin{cases} div(x,y) & = 1, \text{ se "}x \text{ divide } y\text{"}. \\ div(x,y) & = 0, \text{ se "}x \text{ não divide } y\text{"}, \end{cases}$$

   então faça $div(x, y) = \overline{sg}(rm(x, y))$.

□

3.7 Operador de minimalização

O leitor pode observar que cada função recursiva primitiva está definida para qualquer número natural, logo elas são **funções totais**. Note que se deseja caracterizar a família de todas as funções RAM-computáveis, assim toda função que seja implementada por um programa RAM deverá constar dessa caracterização. Entretanto, nem todo programa RAM pára para todos os seus argumentos. Por exemplo, o programa

```
N1  INC R1
    JMP N1a
    CONTINUE
```

não pára para qualquer número natural. Dessa forma, as funções recursivas primitivas não podem interpretar esses programas. O programa que calcula a raiz quadrada de x, visto na seção anterior, é outro exemplo de um programa que computa uma função numérica que não é total. Portanto, quais tipos de função sobre os naturais interpretam esse tipo de programa? Isso motiva o operador a seguir: o operador de minimalização.

O operador de minimalização é um *operador de busca* análogo aos conhecidos "loops while" do Pascal e de outras linguagens. Basicamente, ele busca nos números naturais (partindo de zero) pelo primeiro número que possui uma determinada propriedade, caso afirmativo ele retorna tal número, caso negativo a busca passa para o sucessor do número corrente. Assim, se a tal propriedade não for satisfeita por algum número natural (e.g. $\mid x^2 - 3 \mid= 0$) então o operador entra numa busca eterna. Logo, dependendo da propriedade em questão o operador pode dar origem à uma função que não é total.

Escreve-se:

$$\mu_y \ (...) \tag{3.9}$$

para expressar: "o menor y tal que ...". Inicialmente, tome a seguinte definição:

Definição 3.25 *Seja uma função $f : \mathbb{N}^{m+1} \to \mathbb{N}$, $\vec{x} = (x_1, \ldots, x_m)$ e o conjunto $A_{\vec{x}} = \{y : f(\vec{x}, y) = 0$ e $f(\vec{x}, y')$ está definido para cada $y' < y\}$. Seja $\min(A_{\vec{x}})$ o menor membro de $A_{\vec{x}}$, desde que $A_{\vec{x}} \neq \emptyset$. A* ***minimalização*** *de f é a função $\mu f : \mathbb{N}^m \to \mathbb{N}$ tal que*

$$(\mu f)(\vec{x}) = \begin{cases} \min(A_{\vec{x}}), & \text{se } A_{\vec{x}} \neq \emptyset; \text{ ou} \\ \text{indefinido}, & \text{se } A_{\vec{x}} = \emptyset. \end{cases} \tag{3.10}$$

Denota-se o elemento $\min(A_{\vec{x}})$ por "$(\mu_y)[f(\vec{x}, y) = 0]$". Assim, $(\mu_y)[f(\vec{x}, y) = 0]$ pode ser calculado computando-se $f(\vec{x}, 0)$, $f(\vec{x}, 1)$, ..., até que um desses valores seja igual a 0. A computação não irá parar se houver uma tentativa de calcular $f(\vec{x}, y)$ usando um valor de y para o qual a função não esteja definida, ou se $f(\vec{x}, y) = 0$ para nenhum valor de y.

Lema 3.26 *Se $f : \mathbb{N}^{m+1} \to \mathbb{N}$ é RAM-computável, então a função $(\mu f) : \mathbb{N}^m \to \mathbb{N}$ é RAM-computável.*

Prova: Suponha que f é RAM-computável e P_f é o programa RAM que implementa esta função. Se k é tal que nenhum registrador R_j, para $j > k$, seja referenciado por Pf e tanto `Nu` quanto `Nv` são rótulos que não são utilizados por Pf, então o seguinte programa computa (μf):

```
   Rk+1 <- R1   % Salva os argumentos da função f
   RK+2 <- R2
   etc.
   RK+m <- Rm
   CLR Rk+m+1   % inicializa a variável de busca
Nu R1 <- Rk+1   % Recupera os argumentos x1..xn
   R2 <- RK+2
   etc.
   Rm <- Rk+m
   Rm+1 <- Rk+m+1 % copia o valor atual da variável de busca
   CLR Rm+2 % Limpa os demais registradores
   etc.
   CLR Rk
   Pf   % Computa f(x1,...,xm,Rk+m+1)
   R1 JMP Nvb % verifica se f(x1,...,xm,Rk+n+1) = 0
   INC Rk+m+1 % incrementa a variável de busca
   JMP Nua    % Tenta novamente
Nv R1 <- Rk+m+1
   CONTINUE
```

□

Corolário 3.27 *Se $R(\vec{x}, y)$ é um predicado decidível, então a função*

$$g(\vec{x}) = \mu_y[R(\vec{x}, y)] = \begin{cases} \textit{o menor } y \textit{ tal que } R(\vec{x}, y) \textit{ é verdadeiro, se tal } y \textit{ existe; ou} \\ \textit{indefinido, caso contrário.} \end{cases} \tag{3.11}$$

é computável.

Prova: Faça $g(\vec{x}) = \mu_y(\overline{sg}(C_R(\vec{x}, y)) = 0)$; onde C_R é a função característica associada à R. □

Exemplo 3.28 *Seja $f(x, y) =\mid x - y^2 \mid$. Essa função é recursiva primitiva, e portanto uma função total. A partir de f defina o seguinte predicado decidível $f(x, y) = 0$, cuja função característica é $\overline{sg}(\mid x - y^2 \mid)$. Segundo o corolário anterior a função $g(x) = \mu_y(f(x, y) = 0)$ é uma função computável. Observe que essa função implementa exatamente a função "$\sqrt{x}$" cujo domínio é o subconjunto próprio dos quadrados perfeitos.*

Note que a função parcial $\sqrt{x}$ é computável e qualquer programa que compute essa função entrará em loop para argumentos que não sejam quadrados perfeitos. Assim, o operador μ pode produzir funções computáveis que não são totais a partir de funções computáveis totais. Portanto, ao utilizar o operador de minimalização juntamente com os operadores de recursão, produto e substituição, é possível se

produzir funções computáveis que não são recursivas primitivas, ou seja a classe das funções RAM-computáveis não é suficientemente descrita somente com recursão, produto e composição, é necessário também a minimalização. Isso dá origem a seguinte definição:

Definição 3.29 (Recursão Parcial) *A **classe das funções recursivas parciais**, aqui denotada por $\mathbb{FRP}$, é a menor classe de funções sobre os naturais que contém as funções básicas e é fechada sob os operadores de recursão primitiva, produto, composição e minimalização.*

Proposição 3.30 *Toda função recursiva parcial é RAM-computável; i.e.* $\mathbb{FRP} \subseteq \mathbb{RAM}$

Prova: Direto dos lemas 3.5, 3.10, e 3.13, e do corolário 3.27. □

A proposição acima, então caracteriza uma classe de funções numéricas que é RAM computável. Entretanto, será que essa classe é exatamente a classe das funções RAM-computáveis? i.e. será que $\mathbb{FRP} = \mathbb{RAM}$? Essa resposta será dada no que segue, por enquanto surge ainda mais uma questão: Será que existem funções totais em $\mathbb{FRP}$ que não recursivas primitivas? Do ponto de vista dos programas RAM, isso significa que mesmo sendo funções que sempre param, é necessário uma operação de busca para implementar essas funções, i.e. é necessário uma minimalização.

3.7.1 A função de Ackermann

Ao contrário do que se possa imaginar, a minimalização não produz sempre funções que não são totais. Existem funções computáveis totais que somente podem ser expressas por meio da minimalização. No que segue apresenta-se um exemplo de uma função recursiva parcial que é total, mas em cuja expressão a presença da minimalização é essencial. A função é uma modificação de Rozsa Péter [23] de um exemplo dado por Ackermann, após o qual a função recebeu o seu nome. A função possui a seguinte definição:

$$\begin{cases} \psi(0,y) = y+1 \\ \psi(x+1,0) = \psi(x,1) \\ \psi(x+1,y+1) = \psi(x,\psi(x+1,y)) \end{cases} \tag{3.12}$$

Essa definição envolve uma espécie de dupla recursão que é mais forte que a recursão primitiva. Entretanto, é possível perceber que essas equações de fato definem uma função, pois é suficiente notar que qualquer valor $\psi(x,y)$ $(x>0)$ está definido em termos de valores anteriores $\psi(x_1,y_1)$ com $x_1 < x$, ou $x_1 = x$ e $y_1 < y$. Pode-se estabelecer por indução sobre x e y que $\psi(x,y)$ pode ser obtida usando-se apenas um número finito desses valores. A prova da computabilidade e do fato dela não ser recursiva primitiva é bastante difícil, e por questões de simplicidade não fazem

parte deste curso. Entretanto o leitor pode verificar esse fato em Rozsa Pèter [23].

Note que na definição da classe das funções recursivas parciais nenhuma restrição foi imposta ao uso do operador de minimalização, de maneira que a classe $\mathbb{FRP}$ possui tanto funções totais como funções que não são totais. Note também que com essa definição não se tem como distinguir algebricamente todas as funções totais das não totais, já que existem funções computáveis totais cuja definição necessita do operador de minimalização. Dessa maneira, daqui por diante define-se **função recursiva** toda função recursiva parcial que é total.

3.8 Um pouco de recursão

Os operadores que seguem são abreviações para se definir funções recursivas. Quando as funções envolvidas forem recursivas o resultado será uma função recursiva, e se elas forem recursivas primitivas, o resultado será uma função recursiva primitiva.

3.8.1 Recursão por curso de valores

Definição 3.31 (Recursão por curso de valores) *Se $f_1(\vec{x}), \ldots, f_k(\vec{x})$ são funções recursivas e $P_1(\vec{x}), \ldots, P_k(\vec{x})$ são predicados decidíveis, tal que para cada $\vec{x}$ exatamente um dos predicados é válido (i.e. eles são mutuamente exclusivos). Então a função*

$$g(\vec{x}) = \begin{cases} f_1(\vec{x}), & \text{se } \text{“}P_1(\vec{x}) \text{ é verdadeiro”}. \\ \vdots & \\ f_k(\vec{x}), & \text{se } \text{“}P_k(\vec{x}) \text{ é verdadeiro”}. \end{cases} \tag{3.13}$$

*chama-se **função recursiva por curso de valores**.*

Observe que não se exige que as funções f_i envolvidas sejam recursivas primitivas.

Proposição 3.32 *Uma função $g(\vec{x})$ recursiva por curso de valores é RAM-computável.*

Prova: Assumindo a definição acima, a função pode ser expressa através da seguinte equação: $g(\vec{x}) = C_{P_1}(\vec{x}) \cdot f_1(\vec{x}) + \cdots + C_{P_k}(\vec{x}) \cdot f_k(\vec{x})$; onde C_{P_j} é a função característica associada ao predicado P_j. □

3.8.2 Soma limitada e produto limitado

Definição 3.33 *Suponha que $f(\vec{x}, z)$ é qualquer função. A **soma limitada** de $f(\vec{x}, z)$, denotada por $\sum_{z<y} f(\vec{x}, z)$, é dada pelo seguinte esquema de recursão primitiva:*

$$\begin{cases} \sum_{z<0} f(\vec{x}, z) = 0, \\ \sum_{z<y+1} f(\vec{x}, z) = \left(\sum_{z<y} f(\vec{x}, z) \right) + f(\vec{x}, y). \end{cases} \tag{3.14}$$

A partir da soma limitada a expressão $\sum_{z\leq y} f(\vec{x}, z)$ *significa* $\sum_{z<y} f(\vec{x}, z) + f(\vec{x}, y)$.

Exemplo 3.34 *A expressão* $\sum_{z<4} f(\vec{x}, z)$ *descreve, portanto, a série finita "*$0 + f(\vec{x}, 0) + f(\vec{x}, 1) + f(\vec{x}, 2) + f(\vec{x}, 3)$*", enquanto que a expressão* $\sum_{z\leq 4} f(\vec{x}, z)$ *descreve a série "*$0 + f(\vec{x}, 0) + f(\vec{x}, 1) + f(\vec{x}, 2) + f(\vec{x}, 3) + f(\vec{x}, 4)$*".*

Definição 3.35 *O* ***produto limitado*** *de* $f(\vec{x}, z)$*, denotado por* $\prod_{z<y} f(\vec{x}, z)$*, é definido como:*

$$\begin{cases} \prod_{z<0} f(\vec{x}, z) = 1, \\ \prod_{z<y+1} f(\vec{x}, z) = \left(\prod_{z<y} f(\vec{x}, z) \right) \cdot f(\vec{x}, y). \end{cases} \tag{3.15}$$

Semelhantemente, a expressão $\prod_{z\leq y} f(\vec{x}, z)$ *significará* $\prod_{z<y} f(\vec{x}, z) \cdot f(\vec{x}, y)$.

Exemplo 3.36 *A expressão* $\prod_{z<4} f(\vec{x}, z)$ *descreve, portanto, o produto "*$1 \cdot f(\vec{x}, 0) \cdot f(\vec{x}, 1) \cdot f(\vec{x}, 2) \cdot f(\vec{x}, 3)$*", enquanto que a expressão* $\prod_{z\leq 4} f(\vec{x}, z)$ *descreve o produto "*$1 \cdot f(\vec{x}, 0) \cdot f(\vec{x}, 1) \cdot f(\vec{x}, 2) \cdot f(\vec{x}, 3) \cdot f(\vec{x}, 4)$*".*

Proposição 3.37 *Se* $f(\vec{x}, z)$ *é uma função recursiva, então as funções* $\sum_{z<y} f(\vec{x}, z)$*,* $\sum_{z\leq y} f(\vec{x}, z)$*,* $\prod_{z<y} f(\vec{x}, z)$*, e* $\prod_{z\leq y} f(\vec{x}, z)$ *são funções recursivas.*

Prova: Direto das definições, já que elas são definidas por recursão primitiva. □

Corolário 3.38 *Suponha que* $f(\vec{x}, z)$ *e* $k(\vec{x}, w)$ *são funções recursivas, então as funções* $\sum_{z<k(\vec{x},w)} f(\vec{x}, z)$*,* $\sum_{z\leq k(\vec{x},w)} f(\vec{x}, z)$*,* $\prod_{z<k(\vec{x},w)} f(\vec{x}, z)$*, e* $\prod_{z\leq k(\vec{x},w)} f(\vec{x}, z)$ *também são.*

Prova: Por substituição de y por $k(\vec{x}, w)$ nas expressões $\sum_{z<y} f(\vec{x}, z)$, $\sum_{z\leq y} f(\vec{x}, z)$, $\prod_{z<y} f(\vec{x}, z)$, e $\prod_{z\leq y} f(\vec{x}, z)$. □

3.8.3 Minimalização limitada

Assim como na minimalização, é possível definir um operador de busca para uma faixa finita de números naturais e garantir que o processo de minimalização também seja finito.

Definição 3.39 *Chama-se* ***minimalização limitada****, e denota-se por*

$$(\mu_z < y)(\dots) \tag{3.16}$$

o processo pelo qual se busca e retorna ***o menor*** *z que é menor que y e possua a propriedade "(...)". Caso esse z não exista então o valor resultante do processo será y. O operador $(\mu_z < y)$ é chamado* ***operador de minimalização limitada*** *ou* ***μ-operador limitado****.*

Lema 3.40 *Dada uma função recursiva $f(\vec{x}, y)$, então a função $(\mu_z < y)(f(\vec{x}, z) = 0)$ é recursiva.*

Prova: Considere a função recursiva $h(\vec{x}, v) = \prod_{u \leq v} sg(f(\vec{x}, u))$. Dados $\vec{x}$ e y, suponha que $z_0 = (\mu_z < y)(f(\vec{x}, z) = 0)$. Logo, caso $v < z_0$, então $h(\vec{x}, z) = 1$; caso $z_0 \leq v < y$, então $h(\vec{x}, v) = 0$. Portanto, z_0 é o número de v's menores que y tal que $h(\vec{x}, v) = 1$, ou seja $z_0 = \sum_{v<y} h(\vec{x}, v)$. Caso tal z_0 não exista, então a expressão $\sum_{v<y} h(\vec{x}, v)$ será igual à y. Portanto, $(\mu_z < y)(f(\vec{x}, z) = 0) = \sum_{v<y} h(\vec{x}, v)$. □

Dessa maneira, quando as funções envolvidas nessa busca são funções recursivas, então a função resultante também será uma função recursiva. Observe, ainda, que se elas forem recursivas primitivas o resultado será uma função recursiva primitiva.

Corolário 3.41 *Se $f(\vec{x}, z)$ e $k(\vec{x}, w)$ são funções recursivas, então a função $(\mu_z < k(\vec{x}, w))(f(\vec{x}, z) = 0)$*

Prova: Por substituição se $k(\vec{x}, w)$ em y. □

Assim como no caso da minimalização, agora generaliza-se o lema acima para um predicado recursivo qualquer.

Corolário 3.42 *Suponha que $R(\vec{x}, y)$ é um predicado recursivo, então a função $f(\vec{x}, y) = (\mu_z < y)(R(\vec{x}, z))$ é recursiva.*

Prova: Faça $f(\vec{x}, y) = (\mu_z < y)(\overline{sg}(C_R(\vec{x}, z) = 0))$. □

Observe que assim como no caso do produto e soma limitados, a expressão $(\mu_z < k(\vec{x}, w))R(\vec{x}, z)$ também é uma função recursiva, quando $k(\vec{x}, w)$ é uma função recursiva.

3.8.4 Álgebra da decidibilidade e quantificação limitada

A partir dos operadores anteriores é possível demonstrar que as operações usuais com proposições e as quantificações limitadas são funções recursivas.

Proposição 3.43 (Álgebra da decidibilidade) *Se $P(\vec{x})$, $P_1(\vec{y})$ e $P_2(\vec{z})$ são predicados decidíveis, então os seguintes predicados também são decidíveis:*

1. *$\neg P(\vec{x})$: "não é o caso que $P(\vec{x})$"*
2. *$P_1(\vec{y}) \wedge P_2(\vec{z})$: "$P_1(\vec{y})$ e $P_2(\vec{z})$"*

3. $P_1(\vec{y}) \vee P_2(\vec{z})$: *"$P_1(\vec{y})$ ou $P_2(\vec{z})$"*

4. $P_1(\vec{y}) \rightarrow P_2(\vec{z})$: *"Se $P_1(\vec{y})$, então $P_2(\vec{z})$"*

Prova: As funções: $C_{\neg P}(\vec{x}) = 1 \dot{-} C_P(\vec{x})$, $C_{P_1 \wedge P_2}(\vec{y}, \vec{z}) = C_{P_1}(\vec{y}) \cdot C_{P_2}(\vec{z})$, e $C_{P_1 \vee P_2}(\vec{y}, \vec{z}) = \max(C_{P_1}(\vec{y}), C_{P_2}(\vec{z}))$ computam, respectivamente, os dois primeiros predicados, enquanto que $P_1(\vec{y}) \rightarrow P_2(\vec{z})$ é equivalente à "$\neg P_1(\vec{y})$ ou $P_2(\vec{z})$". □

Proposição 3.44 *Suponha que $R(\vec{x}, y)$ seja um predicado decidível, então os seguintes predicados são decidíveis:*

1. $P_1(\vec{x}, y) \Leftrightarrow (\forall z < y) R(\vec{x}, z)$

2. $P_2(\vec{x}, y) \Leftrightarrow (\exists z < y) R(\vec{x}, z)$

Prova:

1. $C_{P_1}(\vec{x}, y) = \prod_{z<y} C_R(\vec{x}, z)$

2. $P_2(\vec{x}, y) \Leftrightarrow \neg[(\forall z < y) \neg R(\vec{x}, z)]$

□

Observe que como nos casos da minimalização, soma e produto limitados se $k(\vec{x}, w)$ é uma função recursiva, então os predicados $P_1(\vec{x}, k(\vec{x}, w)) \Leftrightarrow (\forall z < k(\vec{x}, w)) R(\vec{x}, z)$ e $P_2(\vec{x}, k(\vec{x}, w)) \Leftrightarrow (\exists z < k(\vec{x}, w)) R(\vec{x}, z)$ também são predicados decidíveis, já que é só substituir y por $k(\vec{x}, w)$ nas expressões.

3.8.5 Estendendo a minimalização limitada

Se existe somente um z tal que $z < y$, e para o qual o predicado recursivo $R(\vec{x}, z)$ vale, então z é calculado pela minimalização limitada $(\mu_z < y)(R(\vec{x}, z))$. Partindo disso, é possível utilizar essa minimalização limitada para calcular

> "O maior z, tal que $z < y$ e para o qual $R(\vec{x}, z)$ é verdadeiro."

Nesse caso, representa-se essa minimalização, por:

$$(\overline{\mu}_z < y)(R(\vec{x}, z)) \tag{3.17}$$

Se tal z existe, então ele é o único z menor que y (e portanto o menor z: μ_z) para o qual $R(\vec{x}, z)$ vale e para todo k, onde $z < k < y$, $\neg R(\vec{x}, k)$ é verdadeiro. Portanto, isso pode ser escrito em termos da minimalização limitada (μ_z) da seguinte maneira:

$$(\overline{\mu}_z < y)(R(\vec{x}, z)) = (\mu_z < y)(R(\vec{x}, z) \wedge ((\forall k < y)[k > z \rightarrow \neg R(\vec{x}, k)]) \tag{3.18}$$

3.9 Exercícios

1. Usando funções recursivas parciais, mostre que as seguintes funções são recursivas:

   (a) $mmc(x, y)$;
   (b) $mdc(x, y)$;
   (c) $D(x) =$" ao número de divisores de x";
   (d) $p_n =$"o n-ésimo número primo";
   (e) $(x)_n =$ "o expoente de p_n (n-ésimo primo) na fatoração prima de x — convenciona-se que $(x)_n = 0$ caso $x = 0$ ou $n = 0$;

2. Mostre que função $\eta : \mathbb{N}^2 \to \mathbb{N}$, $\eta(x, y) = 2^x \cdot (2y + 1) - 1$ é uma bijeção e encontre as funções $\eta_1, \eta_2 : \mathbb{N} \to \mathbb{N}^2$ tal que $\eta(\eta_1(n), \eta_2(n)) = n$.

3. Usando funções recursivas parciais, mostre que os seguintes predicados são decidíveis:

   (a) x é par
   (b) x é ímpar
   (c) $Pr(x)$:"x é um número primo"
   (d) x é uma potência de um número primo
   (e) $x \mid y$ — i.e. y é divisível por x.

3.10 Considerações finais

Nesse capítulo, provou-se que $\mathbb{FRP} \subseteq \mathbb{RAM}$. No que segue apresenta-se um esboço da prova de que $\mathbb{RAM} \subseteq \mathbb{FRP}$. Logo é possível concluir que o limite computacional das RAM's é exatamente calcular funções recursivas parciais. Isso responde a primeira pergunta das considerações finais do capítulo anterior. Com isso, *o limite da computação* depende de se chegar à conclusão, ou assumir, que todo e qualquer possível modelo de computador é equivalente à RAM.

Teorema 3.45 $\mathbb{RAM} = \mathbb{FRP}$.

Prova: A proposição 3.30 mostra que $\mathbb{FRP} \subseteq \mathbb{RAM}$. O oposto, $\mathbb{RAM} \subseteq \mathbb{FRP}$, é provado da seguinte maneira:

Suponha que $f(\vec{x})$ é uma função RAM-computável por um programa P com instruções $(I_1, \ldots, I_s)$. Um passo na computação de $P(\vec{x})$, onde $\vec{x} = (x_1, \ldots, x_m)$ é a execução de uma única instrução I_j desse programa. Considere as seguintes funções que estão relacionadas com as computações de P:

$$c_n^m(\vec{x},t) = \begin{cases} \text{Conteúdo dos registradores } R_1,\ldots,R_n, \\ \text{após } t \text{ passos da computação de } P(\vec{x}), \\ \text{se } P(\vec{x}) \text{ ainda não parou; ou} \\ \\ \text{Conteúdo final dos registradores } R_1,\ldots,R_n, \\ \text{se } P(\vec{x}) \text{ parou após uma quantidade de passos menor ou igual a } t. \end{cases} \tag{3.19}$$

$$j_n^m(\vec{x},t) = \begin{cases} \text{O índice da próxima instrução na seqüência } (I_1,\ldots,I_s) \\ \text{quando } t \text{ passos da computação de } P(\vec{x}) \text{ tenham sido completados,} \\ \text{se } P(\vec{x}) \text{ ainda não parou após uma quantidade de passos menor ou} \\ \text{igual a } t\text{; ou} \\ \\ 0, \text{ se } P(\vec{x}) \text{ parou após uma quantidade de passos menor ou igual à } t. \end{cases} \tag{3.20}$$

Intuitivamente, c_n^m e j_n^m são funções totais. Observe o seguinte:

Se $f(\vec{x})$ está definida, então $P(\vec{x})$ converge após exatamente t_0 passos. Nesse caso $t_0 = (\mu_t)(j_n^m(\vec{x},t) = 0)$ e $f(\vec{x}) = c_n^m(\vec{x},t_0)$. Se por outro lado $f(\vec{x})$ não está definido, então $P(\vec{x})$ diverge e $j_n^m(\vec{x},t)$ nunca é zero. Nesse caso, a expressão $(\mu_t)(j_n^m(\vec{x},t) = 0)$ está indefinida. Portanto, em ambos os casos tem-se:

$$f(\vec{x}) = c_n^m\Big(\vec{x},(\mu_t)(j_n^m(\vec{x},t) = 0)\Big). \tag{3.21}$$

Portanto, para mostrar que f é recursiva parcial, resta apenas demonstrar que as funções c_n^m e j_n^m também são. Na verdade, essas funções são recursivas primitivas, mas a prova rigorosa disto será adiada até o capítulo 6, onde serão apresentados alguns conceitos necessários para iso. Por enquanto, o leitor deve inicialmente aceitar que essas duas funções são recursivas primitivas. Isso conclui a prova e o capítulo. □

Capítulo 4

Linguagem de programação WHILE

Até o presente foram apresentados dois modelos de computação que capturam dois importantes aspectos da computação real: o primeiro, as máquinas RAM, cuja operacionalidade se assemelha aos computadores reais, e portanto capta o computador enquanto máquina (*hardware*). O segundo modelo, as funções parciais recursivas, que capta a natureza funcional dos computadores.

Neste capítulo será abordado um terceiro modelo: a linguagem de programação teórica WHILE, que, como o próprio nome diz, é uma linguagem de programação e portanto capta o aspecto de *software* da computação real.

Todo modelo de computabilidade se baseia na premissa de que certas operações ou funções básicas são inerentemente computáveis. No caso da linguagem WHILE, considera-se três operações básicas: escrever zero, somar um a qualquer número natural* e comparar dois números naturais para decidir se eles são iguais ou não. Além dessas operações básicas, a linguagem WHILE considera atribuições simples e o comando **while**. Assim a linguagem WHILE pode ser vista como uma sublinguagem das linguagens de programação procedural de alto nível, tipo Pascal, Fortran ou Java, só que mais pobre em comandos assim como em tipos de dados (só considera o tipo de dados dos números naturais). O primeiro ponto não é problema uma vez que, como será visto neste curso, esta linguagem é tão poderosa quanto qualquer linguagem de programação real com a diferença que a linguagem WHILE, por ser teórica, não possui limitações de máquina, como representar somente uma quantidade finita de números. A segunda limitação, também como será vista no decorrer do curso, é aparente, uma vez que a mesma linguagem usada para representar números naturais pode ser usada para representar inteiros, pontos flutuantes, listas de inteiros, etc.

*Na verdade, representações de números naturais em alguma linguagem formal.

4.1 Sintaxe da linguagem WHILE

Para descrever a sintaxe da linguagem WHILE será usado, como usual em linguagens de programação, as formas de Backus-Naur (BNF). Uma BNF é um conjunto de produções da forma

$$\langle nome \rangle ::= \underline{\text{expressão}} \tag{4.1}$$

onde palavras entre "⟨" e "⟩" são variáveis auxiliares e podem ser substituídas pela expressão ao lado direito do símbolo "::=". A expressão do lado direito é uma cadeia de variáveis auxiliares com símbolos terminais (os símbolos básicos da linguagem), podendo até ser vazia (nesse caso denotada por λ). Quando houver mais de uma produção com a mesma variável auxiliar no lado esquerdo, abrevia-se colocando o mesmo lado esquerdo e separando os direitos com símbolo "|". Por exemplo, abrevia-se as produções ⟨a⟩::= expr1, ⟨a⟩ ::= expr2 e ⟨a⟩ ::= expr3, por ⟨a⟩ ::= expr1 | expr2 | expr3.

A linguagem WHILE possui quatro classes básicas de palavras ("alfabeto" da linguagem):

Nomes de variáveis. Um nome de variável válido é qualquer cadeia de letras maiúsculas e dígitos numéricos que comece com uma letra. Por exemplo, TEMP, X, NOME1, N23TXR0, A5, etc. Distinguiremos entre dois tipos de variáveis as de entrada, que sempre começaram com a letra A e as que não são de entradas.

Símbolos de operação. Os únicos símbolos de operação são " ' " e "0", que denotam a função sucessor e a função constante zero, respectivamente. Note que outras constantes, por exemplos 2 (dois) não são permitidas.

Símbolos de relação. Considera-se apenas o símbolo de relação "≠", que denota a não igualdade dos valores de duas variáveis.

Símbolos de programa. Existem os símbolos "←", ";" e ",". O símbolo ← denotará atribuição, isto é, o valor da expressão que está no lado direito de ← é atribuído à variável que está no lado esquerdo. O símbolo ";" denotará o final de um comando e o símbolo "," será usado para separar nomes de uma lista. Além dos símbolos existem seis palavras reservadas: **input**, **output**, **begin**, **end**, **while** e **do**.

Como usual, um programa numa linguagem procedural é uma seqüência de comandos os quais são executados seqüencialmente na ordem em que aparecem a menos que ocorra um desvio. Um programa WHILE, antes de começar o programa propriamente dito declara quais variáveis serão de entrada e quais serão de saída[†]

[†]Declarar as entradas e saídas não é essencial. De fato muitas linguagens teóricas, por exemplo WHILE de [18] e PL em [4], não usam esse tipo de comandos. Já na linguagem LPM em [9] e WHILE de [16], as variáveis de entrada e de saída são declaradas explicitamente.

seguida pelas palavras reservadas (**begin** e **end**) para indicar o início e o fim do programa. Estas declarações das variáveis de entrada e de saída só são feitas se o programa WHILE tiver entradas e/ou saídas. Assim, um programa WHILE é descrito como

⟨programa-while⟩ ::= ⟨comandos de entrada-saída⟩ **begin** ⟨comandos⟩ **end**

⟨comandos de entrada-saída⟩ ::= ⟨entradas⟩; ⟨saídas⟩; | ⟨entradas⟩; | ⟨saídas⟩; | λ

⟨entradas⟩ ::= **input** ⟨lista de variáveis de entradas⟩

⟨saídas⟩ ::= **output** ⟨lista de variáveis de saída⟩

⟨lista de variáveis de entrada⟩ ::= ⟨variável de entrada⟩ | ⟨variável de entrada⟩, ⟨lista de variáveis de entrada⟩

⟨variável de entrada⟩ ::= ⟨variável de entrada⟩⟨letra⟩ | ⟨variável de entrada⟩⟨dígito⟩ | A

⟨lista de variáveis de saída⟩ ::= ⟨variável não de entrada⟩ | ⟨variável não de entrada⟩, ⟨lista de variáveis de saída⟩

⟨variável não de entrada⟩ ::= ⟨variável não de entrada⟩⟨letra⟩ | ⟨variável⟩⟨dígito⟩ | ⟨letra diferente de A⟩

⟨letra diferente de A⟩ ::= $B \mid \ldots \mid Z$

⟨letra⟩ ::= A | ⟨letra diferente de A⟩

⟨dígito⟩ ::= $0 \mid 1 \mid \ldots \mid 9$

Comandos de um programa são uma seqüência (podendo ser vazia) de comandos válidos.

⟨comandos⟩ ::= λ | ⟨comando⟩; ⟨comandos⟩

Um comando pode ser dois tipos: de atribuição que indica que uma variável do programa, independente do seu valor atual, passa a ter o valor zero ou o valor de uma variável do programa (podendo ser ela mesma ou outra variável) incrementado em um. O outro tipo de comando é o **while** que indica que uma seqüência de comandos vai ser executado enquanto o valor de duas variáveis forem diferentes.

⟨comando⟩ ::= ⟨atribuição⟩ | **while** ⟨condição⟩ **do** **begin** ⟨comandos⟩ **end**

⟨atribuição⟩ ::= ⟨variável não de entrada⟩ $\leftarrow$ 0 | ⟨variável não de entrada⟩ $\leftarrow$ ⟨variável⟩′

⟨condição⟩ ::= ⟨variável⟩ $\neq$ ⟨variável⟩

⟨variável⟩ ::= ⟨variável de entrada⟩ | ⟨variável não de entrada⟩

Exemplo 4.1 *Exemplos de programas* WHILE*:*

```
input A;
output B;
begin
    B ← 0;
    while B ≠ A do
    begin
        B ← B′;
    end
end
```

Programa a.

```
input A1, A2;
output B;
begin
    U ← 0;
    while U ≠ A2 do
    begin
        A1 ← A1′;
        U ← U′;
    end
    B ← A1′;
end
```

Programa b.

```
input A1, A2;
output B;
begin
    U ← 0;
    B ← 0;
    while U ≠ A1 do
    begin
        V ← 0;
        while V ≠ A2 do
        begin
            V ← V′;
            U ← U′;
        end
        B ← B′;
    end
end
```

Programa c.

```
input A;
output B;
begin
    U ← 0;
    B ← 0;
    while A ≠ U do
    begin
        U ← U′;
        while A ≠ U do
        begin
            U ← U′;
            B ← B′;
        end
    end
end
```

Programa d.

4.2 Semântica informal de WHILE

A execução de um programa WHILE obedece os seguintes requisitos:

1. Os comandos são executados na ordem que aparecem;
2. Os comandos de atribuição são interpretados da seguinte maneira:
   (a) $X \leftarrow 0$; associa o número natural zero à variável X.

(b) $X \leftarrow Y'$; associa à variável X o valor associado à variável Y incrementado em uma unidade.

3. Um comando **while** $X \neq Y$ **do begin** ⟨comandos⟩ **end**, executará ⟨comandos⟩ enquanto o valor associado à variável X for diferente do valor associado à variável Y. Note que se estes valores nunca tornam-se iguais, então o programa entrará em um ciclo infinito e portanto não parará nem retornará qualquer resultado.

4. O comando **input** ⟨lista de variáveis de entrada⟩; não só declara que as variáveis da lista são de entrada, mas também aciona um mecanismo que permite receber valores de algum "médio físico". Porém como a linguagem WHILE é teórica, pode-se somente supor que ela é executada numa "máquina teórica" e portanto não há um médio físico como nos computadores reais (que são teclado, memória primaria e memória secundária), assim pode-se pensar que a máquina teórica que executa o programa possui uma quantidade ilimitada de registradores cada um permitindo armazenar um número natural de qualquer ordem e que são usados tanto para armazenar valores de entrada quanto de saída e temporários. A máquina teórica associa à cada variável o conteúdo de um dos registradores.

5. O valor inicial de uma variável não declarada como entrada é sempre 0.

6. O comando **output** ⟨lista de variáveis de saída⟩ é análogo ao comando **input**. Quando executado pela máquina teórica ele declara que as variáveis da lista serão consideradas como saída, e que portanto os registradores a elas associados, ao final da execução (caso termine), serão preservados enquanto os outros serão limpados. Note que variáveis declaradas como de entrada não podem ser de saída.

Exemplo 4.2 *Seja*

```
1)  input A1, A2;
2)  output B;
3)  begin
4)        U ← 0;
5)        while U ≠ A2 do
6)        begin
7)              A1 ← A1';
8)              U ← U'
9)        end
10)       B ← A1'
11) end
```

o programa WHILE *b. do exemplo 4.1 com as linhas numeradas. A semântica informal deste programa é a seguinte: Em 1) declara-se que as variáveis* $A1$ *e* $A2$ *serão de entrada e recebe valores para a computação variáveis; em 2) declara-se que*

a variável B será de saída, no final da execução ele retornará o valor associado a B; em 3) inicia-se o programa propriamente dito; em 4) atribuí-se à variável U o valor zero (note que este comando é desnecessário, uma vez que toda variável inicialmente tem associado o valor 0, mas por motivos de claridade optou-se por atribuir explicitamente à U o valor 0); em 5) compara-se os valores associados às variáveis U e $A2$, se os valores são os mesmos então o programa executará a instrução 10), caso contrário executará o bloco de comandos entre o **begin** *e o* **end**, *i.e. as linhas 7) e 8), após uma execução do bloco volta-se comparar os valores de U e $A2$; em 7) atribui-se à variável $A1$ o valor atual de $A1$ mais um; em 8) atribui-se à variável U o valor atual de U mais um; em 10) atribui-se à variável de saída B o valor atual da variável $A1$ mais um; em 11) pára-se a execução do programa retornando o valor associado à variável declarada como de saída, ou seja associado à B. Analisando esse programa, percebe-se que o bloco de comandos ligados ao* **while** *será executado k-vezes, onde k é o valor associado a $A2$. A cada execução, o valor de $A1$ é incrementado em uma unidade, $A1$ terá o seu valor inicial (dado como entrada) mais o valor dado como entrada para $A2$. O sucessor desse valor é então atribuído à variável de saída B. Portanto este programa calcula o sucessor da soma de dois valores dados como entrada.*

4.3 Semântica formal

Existem muitas formas de tratar formalmente a semântica de uma linguagem de programação. Entre as mais conhecidas está a semântica denotacional, introduzida por Dana Scott e Christopher Strachey em [26]. A idéia da semântica denotacional é associar a cada frase da linguagem de programação um objeto matemático, de formas que o significado de um programa será a composição do significado de suas frases [21]. Portanto, a denotação de um programa é determinada pela denotação de suas frases. Semântica denotacional permite que se dê definições canônicas de significados de programas, e portanto livres de técnicas de implementação. A parte matemática é chamada de teoria dos domínios (alguns textos sobre esta teoria são [1, 25, 29]).

A semântica denotacional da linguagem WHILE é dada pela função parcial

$$[\![\ ]\!] : While \longrightarrow [\mathbb{N}^* \longrightarrow \mathbb{N}^*]$$

onde $[\mathbb{N}^* \longrightarrow \mathbb{N}^*] = \{f : \mathbb{N}^k \longrightarrow \mathbb{N}^m \ : \ f \text{ é parcial e } k, m \in \mathbb{N}\}$, ou seja

$$[\mathbb{N}^* \longrightarrow \mathbb{N}^*] = \bigcup_{k,m \in \mathbb{N}} [\mathbb{N}^k \longrightarrow \mathbb{N}^m]$$

com $[\mathbb{N}^k \longrightarrow \mathbb{N}^m]$ sendo o conjunto das funções parciais que mapeiam k-tuplas de números naturais em m-tuplas de números naturais.

Seja $P =$ **input** $A_1, A_2, \ldots, A_k$; **ouput** $B_1, B_2, \ldots, B_m$; **begin** C **end** um programa WHILE, onde $X_1, \ldots, X_k, Y_1, \ldots, Y_m$ são variáveis válidas da WHILE, então

$$[\![P]\!] = \omega^n_{k,m} \circ [\![C]\!] \circ \alpha^n_k \tag{4.2}$$

onde n é o número de variáveis usadas no programa P. α^n_k é chamada **função de entrada**, e é definida por $\alpha^n_k(x_1, \ldots, x_k) = (x_1, \ldots, x_k, \underbrace{0, \ldots, 0}_{(n-k)-vezes})$. Já a função $\omega^n_{k,m}$ é chamada **função de saída**, é definida por $\omega^n_{k,m}(x_1, \ldots, x_n) = (x_{k+1}, \ldots, x_{k+m})$.

Seja $i : Var(P) \longrightarrow \mathbb{N}$ a função que atribui a cada variável que ocorre em P o número de sua aparição. Note que se uma variável ocorre mais de uma vez só é considerada sua primeira aparição. Assim, no caso do programa b do exemplo 4.1 tem-se que $i(A1) = 1$, $i(A2) = 2$, $i(B) = 3$ e $i(U) = 4$.

Se $C = c_1; \ldots; c_p$, então $[\![C]\!] = [\![c_p]\!] \circ \ldots \circ [\![c_1]\!]$, onde para cada $j = 1, \ldots, p$ tem-se que $[\![c_j]\!] : \mathbb{N}^n \longrightarrow \mathbb{N}^n$ é definida por:

1. Se $c_j = X \leftarrow 0$ para alguma variável X então
   $[\![X \leftarrow 0]\!](x_1, \ldots, x_n) = (x_1, \ldots, x_{i(X)-1}, 0, x_{i(X)+1}, \ldots, x_n)$.
2. Se $c_j = X \leftarrow Y'$ para variáveis X e Y quaisquer então
   $[\![c_j]\!](x_1, \ldots, x_n) = (x_1, \ldots, x_{i(X)-1}, x_{i(Y)} + 1, x_{i(X)+1}, \ldots, x_n)$.
3. $[\![$**while** $X \neq Y$ **do begin** C; **end**$]\!](x_1, \ldots, x_n) =$

$$\begin{cases} (x_1, \ldots, x_n) & \text{, se } x_{i(X)} \neq x_{i(Y)} \\ [\![\textbf{while } X \neq Y \textbf{ do begin } C; \textbf{ end}]\!]([\![C]\!](x_1, \ldots, x_n)) & \text{, senão} \end{cases}$$

Exemplo 4.3 *Seja P o programa a. do exemplo 4.1. Então para $k = 1$, $m = 1$, $n = 2$, $i(A) = 1$ e $i(B) = 2$. Assim,*

$\alpha^2_1(x) = (x, 0)$,

$\omega^2_{1,1}(x, y) = y$,

$[\![B \leftarrow 0]\!](x, y) = (x, 0)$,

$[\![$**while** $B \neq A$ **do begin** $B \leftarrow B'$; **end**$]\!](x, y) =$

$$\begin{cases} (x, y) & \text{, } se\ x = y \\ [\![\textbf{while } B \neq A \textbf{ do begin } B \leftarrow B'; \textbf{end}]\!][\![B \leftarrow B']\!](x, y) & \text{, } senão \end{cases}$$

Assim, para a entrada 2 tem-se que

$$
\begin{aligned}
[P](2) &= \omega^2_{1,1} \circ [\textbf{while } B \neq A \textbf{ do begin } B \leftarrow B'; \textbf{ end}] \circ [B \leftarrow 0] \circ \alpha^2_1(2) \\
&= \omega^2_{1,1} \circ [\textbf{while } B \neq A \textbf{ do begin } B \leftarrow B'; \textbf{ end}] \circ [B \leftarrow 0](2,0) \\
&= \omega^2_{1,1} \circ [\textbf{while } B \neq A \textbf{ do begin } B \leftarrow B'; \textbf{ end}](2,0) \\
&= \omega^2_{1,1} \circ [\textbf{while } B \neq A \textbf{ do begin } B \leftarrow B'; \textbf{ end}] \circ [B \leftarrow B'](2,0) \\
&= \omega^2_{1,1} \circ [\textbf{while } B \neq A \textbf{ do begin } B \leftarrow B'; \textbf{ end}](2,1) \\
&= \omega^2_{1,1} \circ [\textbf{while } B \neq A \textbf{ do begin } B \leftarrow B'; \textbf{ end}] \circ [B \leftarrow B'](2,1) \\
&= \omega^2_{1,1} \circ [\textbf{while } B \neq A \textbf{ do begin } B \leftarrow B'; \textbf{ end}](2,2) \\
&= \omega^2_{1,1}(2,2) \\
&= 2
\end{aligned}
$$

Generalizando, tem-se que $[P](x) = x$.

4.4 Funções WHILE-computáveis

Diz-se que uma função $f : \mathbb{N}^k \longrightarrow \mathbb{N}^m$ é computada por uma programa WHILE P, se $[P] = f$. Uma função $f : \mathbb{N}^k \longrightarrow \mathbb{N}^m$ é dita WHILE-**computável** se existe um programa WHILE P que computa f. O conjunto de todas as funções WHILE-computáveis será denotado por $\mathbb{FWC}$.

Os programas a., b., c. e d.,do exemplo 4.1, computam as funções $f(x) = x$, $f(x,y) = x + y + 1$, $f(x,y) = qt(y,x)$ se y divide x e $f(x,y) \uparrow$ caso contrário, e $f(x) = x \dot{-} 1$, respectivamente. Um programa WHILE que computa a função multiplicação entre dois números naturais é o seguinte:

```
input A1, A2;
output B;
begin
   B ← 0;
   U ← 0;
   while U ≠ A1 do
   begin
      V ← 0;
      while V ≠ A2 do
      begin
         V ← V';
         B ← B'
      end
      U ← U'
   end
end
```

Todas estas funções são recursivas parciais. A seguir será provado que todas as funções WHILE-computáveis são também recursivas parciais.

Teorema 4.4 *If* $f : \mathbb{N}^k \longrightarrow \mathbb{N}^m$ *é* WHILE*-computável então* f *é parcial recursiva. Isto é,* $\mathbb{FWC} \subseteq \mathbb{FRP}$.

Prova: Seja P um programa que computa f, isto é $[P] = f$. Mostra-se, então, que $[P]$ é uma função parcial recursiva. Uma vez que composição de funções recursivas parciais é parcial recursiva e que a semântica denotacional de um programa WHILE é a composição da função de entrada com a função que denota cada comando do programa com a função de saída, é suficiente mostrar que cada uma delas é parcial recursiva.

$$\alpha_k^n = Id^k \times \underbrace{Z \times \ldots \times Z}_{(n-k)-vezes}^{\ddagger}$$

$$\omega_{k,m}^n = \underbrace{\pi \times \ldots \times \pi}_{k-vezes} \times Id^m \times \underbrace{\pi \times \ldots \times \pi}_{(n-(k+m))-vezes} .$$

$$[X \leftarrow 0] = Id^{i(X)-1} \times Z \circ \pi \times Id^{n-i(X)}$$

$$[X \leftarrow Y'] = S_3^n(Id^n, Id^{i(X)-1}, (S \circ U_{i(Y)}^n) \times \underbrace{\pi \times \ldots \pi}_{n-i(Y)}, Id^{n-i(X)}).$$

$$[\textbf{while } X \neq Y \textbf{ do begin } C; \textbf{ end}] = S_2^n([C]^\sharp, Id^n, \mu(S_2^n(\approx, U_{i(X)}^n, U_{i(Y)}^n) \circ [C])).$$

Inversamente, a seguir será provado que toda função parcial recursiva é WHILE-computável e portanto $\mathbb{FRP} \subseteq \mathbb{FWC}$.

Teorema 4.5 *Se $f : \mathbb{N}^k \longrightarrow \mathbb{N}^m$ é uma função parcial recursiva então f é* WHILE*-computável.*

Prova: Se f é parcial recursiva então pela definição 3.29, ou f é uma função básica, ou pode ser obtida a partir de duas ou mais funções recursivas parciais via operação de composição, ou via operação de produto cartesiano, ou via recursão primitiva, ou via minimalização. Será provado por indução na quantidade de operações usadas para obter f que existe um programa WHILE que computa f. Mas só se mostrará o programa sem demonstrar que ele realmente computa f, pois seria necessário provar que a semântica denotacional do programa resulta em f, o qual é bastante mais complicado, pois provavelmente a expressão da função obtida via $[P]$ não será igual à expressão da função parcial recursiva original.

- A função zero é WHILE-computável. De fato, um programa WHILE que computa Z é o seguinte:

  input ;
  output B;
  begin
  $B \leftarrow 0$
  end

[‡]Lembre que aqui se está fazendo um abuso de linguagem, na verdade $\alpha_k^n = Id^{k-1} \times S_{n-k+1}(Id^{n-k+1}, Id^1, \underbrace{Z \circ \pi, \ldots, Z \circ \pi}_{(n-k)-vezes})$.

- A função terminal é WHILE-computável. De fato, um programa WHILE que computa π é o seguinte:

  input A;
  output ;
  begin
  end

- A função identidade é WHILE-computável. De fato, um programa WHILE que computa Id^m é o seguinte:

  input $A1, \ldots, Am$;
  output $B1, \ldots, Bm$;
  begin
  $B1 \leftarrow 0$;
  while $B1 \neq A1$ **do**
  begin
  $B1 \leftarrow B1'$
  end
  $\vdots$
  $Bm \leftarrow 0$;
  while $Bm \neq Am$ **do**
  begin
  $Bm \leftarrow Bm'$
  end
  end

- A função sucessor é WHILE-computável. De fato, um programa WHILE que computa S é o seguinte:

  input A;
  output B;
  begin
  $B \leftarrow A'$
  end

- Para cada $n > 1$ e $1 \leq i \leq n$ a função i-ésima projeção é WHILE-computável. De fato, um programa WHILE que computa U_i^n é o seguinte:

  input $A1$, ..., An;
  output XB;
  begin
  $B \leftarrow 0$;
  while $B \neq Ai$ **do**

begin
$B \leftarrow B'$
end
end

- Se $g_1 : \mathbb{N}^n \longrightarrow \mathbb{N}^{p1}, \ldots, g_m : \mathbb{N}^n \longrightarrow \mathbb{N}^{pm}$ e $f : \mathbb{N}^t \longrightarrow \mathbb{N}^p$, onde $t = \sum_{i=1}^{m} pi$, são WHILE-computáveis, então por hipóteses existem programas WHILE $P_{g1}, \ldots, P_{gm}$ e P_f tais que $[P_{gi}] = g_i$ para cada $i = 1, \ldots, m$ e $[P_f] = f$. Claramente, se renomeia as variáveis de um programa, e o novo programa continuará computando a mesma função. Assim, pode-se renomear as variáveis dos programas $P_{g1}, \ldots, P_{gm}$ de tal forma que todos tenham as mesmas variáveis de entrada (por exemplo $A1, \ldots, An$) e diferentes variáveis auxiliares (por exemplo o i-ésimo programa poderia ter as variáveis auxiliares: $B_i1, \ldots, B_ik_i$ onde $k_i =\mid Var(P_{gi}) \mid -n$) de tal modo que as variáveis de saída sejam sempre as primeiras ($B_i1, \ldots, B_ipi$). Assim, o programa WHILE que computa a função g_i tem a seguinte forma:

  input $A1, \ldots, An$;
  output $B_i1, \ldots, B_ipi$;
  begin
  C_i
  end

  onde C_i é o corpo (comandos) do programa. Analogamente o programa P_f pode ser modificado renomeando as variáveis de tal forma que as variáveis de entrada sejam $B_11, \ldots, B_1p1, \ldots, B_m1 \ldots, B_mpm$ e as variáveis auxiliares sejam diferentes das usadas nos P_{gi} (já modificado), por exemplo $F1, \ldots, F_r$, onde $r =\mid Var(P_f) \mid -t$ (assim as variáveis de saída seriam $F1, \ldots, Fp$).

  O programa WHILE que tem como variáveis de entrada $A1, \ldots, An$ e como variáveis de saída as variáveis de saída de P_f e como comandos: $C_1; \ldots; C_m; C_f$, onde C_f são os comandos de P_f (já modificado) computa a função $h : \mathbb{N}^n \longrightarrow \mathbb{N}^p$, definida por:

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)).$$

- Se $f : \mathbb{N}^k \longrightarrow \mathbb{N}^m$ e $g : \mathbb{N}^n \longrightarrow \mathbb{N}^p$ são funções recursivas parciais e WHILE-computáveis, então por hipóteses existem programas WHILE $P_f, \ldots, P_g$ tais que $[P_f] = f$ e $[P_g] = g$. Sem qualquer perda, as variáveis desses programas, podem ser renomeadas de tal forma que ambos tenham diferentes variáveis. Seja o programa WHILE

  input $AF1, \ldots, AFk, AG1, \ldots, AGn$;
  output $BF1, \ldots, BFm, BG1, \ldots, BGp$;
  begin

C_f;
C_g
end

onde $AF1, \ldots, AFk$ são as variáveis de entrada, $BF1, \ldots, BFm$ e C_f os comandos do programa P_f (já modificado) e $AG1, \ldots, AGn$ são as variáveis de entrada, $BG1, \ldots, BGp$ e C_f os comandos do programa P_g (já modificado). Claramente este programa computa $f \times g$.

- Se $g : \mathbb{N}^m \longrightarrow \mathbb{N}^s$ e $h : \mathbb{N}^{m+1+s} \longrightarrow \mathbb{N}^s$ são WHILE-computáveis, então existem programas P_g e P_h que computam g e h, respectivamente. Por simplicidade pense que as variáveis de entrada de P_g são $A1, \ldots, Am$ e as de P_h são $A1, \ldots, An$, onde $n = m + 1 + s$, pense também que as variáveis de saída de P_g e P_h são $BG1, \ldots, BGs$ e $BH1, \ldots, BHs$, respectivamente. Então

input $A1, \ldots, A(m+1)$;
output $BH1, \ldots, BHs$;
begin
C_g;
$B \leftarrow 0$;
while $B \neq A(m+1)$ **do**
begin
C'_h;
$B \leftarrow B'$
end
end

onde C'_h é C_h substituindo todas as ocorrências de A_{m+1}, $A_{m+2}, \ldots, A_{m+1+s}$ por $B, BG1, \ldots, BGs$, respectivamente.

Este programa computa a função f da definição 3.11 (recursão primitiva a partir de g e h).

- Seja $f : \mathbb{N}^{m+1} \longrightarrow \mathbb{N}$ uma função WHILE-computável. Então existe um programa P_f que computa f. Sejam $A1, \ldots, A(m+1)$ as variáveis de entrada, B a variável de saída e C_f os comandos de P_f. O seguinte programa

input $A1, \ldots, Am$;
output Y;
begin
$Y \leftarrow 0$;
C'_f;
while $B \neq 0$ **do**
begin
$Y \leftarrow Y'$;
$X1 \leftarrow 0$;

```
        ⋮
        Xk ← 0;
        C'_f
    end
end
```

onde C'_f é C_f substituindo todas as ocorrências de $A(m{+}1)$ por Y e $X1, \ldots, X_k$ são as variáveis auxiliares de C_f que não são de saída (assim, $k = \mid Var(P_f) \mid -(m+2)$).

Este programa computa a minimalização de f, isto é μf.

Corolário 4.6

$$\mathbb{FWC} = \mathbb{FRP}$$

4.5 Exercícios

1. Identifique os erros sintáticos no seguinte "programa" WHILE:

```
input A1; B2, C3;
output
begin
    A1 ← 0;
    while C ≠ B2 do
    begin
        i ← 1;
        while i = C3 do
        begin
            C ← i;
            i ← i';
        end
    end
end
```

2. Dê a semântica informal e formal para o programa WHILE a seguir:

```
input A1, A2;
output B;
begin
    B ← 0;
    C ← 0;
    while C ≠ A1 do
    begin
        C ← C'
```

```
    end
    while C ≠ A2 do
    begin
        B ← B′;
        B ← B′;
        C ← C′
    end
end
```

3. Descreva informalmente que faz o programa do exercício anterior.

4. Faça programas WHILE que computem as funções:

   (a) $f(x, y) = (y, x)$.

   (b) $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ definida por $f(x, y) = x + y$ (note que o exemplo 4.1, b., faz $f(x, y) = x + y + 1$ enquanto o exemplo 4.1, d., faz $f(x) = x \dot{-} 1$).

   (c) $fac : \mathbb{N} \longrightarrow \mathbb{N}$ definida por $fac(x) = \sum_{i=1}^{x} i$.

   (d) $f(x, y) = 1$ se $x \leq y$, caso contrário $f(x, y) = 0$.

   (e) $f(x, y, z) = \max\{x, y, z\}$.

   (f) $f(x) = 1$ se x é par e $f(x) = 0$ se x é ímpar.

4.6 Considerações finais

A linguagem de programação WHILE, por ser teórica, é uma linguagem de baixo nível (sem uma grande quantidade de variantes de comandos nem de tipos de dados). Porém, assim como na computação real as primeiras linguagens de programação foram de baixo nível e paulatinamente aumentaram de nível, incorporando comandos mais poderosos. Na linguagens de programação While isso também pode ser feito através de um recurso conhecido como "macros". A idéia é incorporar comandos mais complexos à linguagem, por exemplo atribuições do tipo: $B \leftarrow C$, $B \leftarrow C + D$ e $B \leftarrow C * D$ (com a sua semântica natural) ou comandos de controle tipo:

```
for B = 0 to C do
begin
    ⟨comandos⟩
end
```

onde B não ocorre em ⟨comandos⟩, e

```
if ⟨condição⟩
then
begin
    ⟨comandos⟩
end
```

else
begin
    ⟨comandos⟩
end

Como usual, pode-se pensar que um comando desses tipos é, em "tempo de compilação"[§], substituído por uma seqüência de comandos, por exemplo a atribuição $B \leftarrow C$ seria substituída pelos comandos:

$B \leftarrow 0$;
while $B \neq C$ **do**
begin
  $B \leftarrow B'$
end

Já um comando do tipo:

for $B = 0$ **to** C **do**
begin
  ⟨comandos⟩
end

seria substituído pelos comandos:

$B \leftarrow 0$;
while $B \neq C$ **do**
begin
  ⟨comandos⟩;
  $B \leftarrow B'$
end

Note que quem determina o tipo de dados em WHILE é a função semântica. Assim, incluir outros tipos de dados em WHILE poderia ser feito simplesmente, classificando os nomes de variáveis (por exemplo aquelas que terminem em Z seriam do tipo *Inteiro*, as que terminem em F seria do tipo *ponto flutuante*, etc. Assim, WHILE poderia suportar vários tipos de dados.

[§]Essa expressão é usada em linguagens de computação real para indicar que no momento da compilação do programa, o compilador substitui a macro pela seqüência de comandos pré-estabelecidos. Já numa linguagem teórica como WHILE, quem faz a tarefa do compilador é a função semântica, e a semântica de uma macro seria igual à composição da semântica dos comandos que ele sintetiza.

Capítulo 5

Tese de Church-Turing

Neste capítulo serão apresentadas as noções intuitivas de procedimento efetivo e algoritmos e a tese de Church-Turing que liga essas noções com a de funções computáveis por alguns dos modelos apresentados aqui (assim por outros não apresentados neste texto).

5.1 Procedimentos efetivos e algoritmos

A noção de procedimentos efetivo é uma noção intuitiva que tenta refletir processo estritamente mecânico, i.e. processos que se seguidos a risca sempre proporcionem o mesmo resultado. O interesse por este tipo de processos é antigo, de fato já os gregos antigos tentaram encontrar processos efetivos para algumas operações matemáticas, por exemplo o algoritmo da divisão de Euclides é um tal processo.

Definição 5.1 *Um* **procedimento efetivo** *é uma descrição finita e não ambígua de um conjunto finito de operações as quais devem ser efetivas, no sentido que exista um procedimento estritamente mecânico para realizar estas.*

Observe que uma destas operações efetivas num procedimento efetivo pode ser do tipo "va para a operação n". Assim, o conjunto finito de operações pode descrever um conjunto infinito de passos computacionais (execução das operações). Um conceito estreitamente ligado ao de procedimento efetivo é o de algoritmo?

Definição 5.2 *Um* **algoritmo** *é um procedimento efetivo que especifica uma seqüência de operações as quais sempre param.*

5.2 Tese de Church-Turing

No exemplo 2.10 o poder da máquina RAM foi "aumentado" ao se permitir a abreviação, e a re-utilização de programas RAM já conhecidos. Ou seja, os programas

RAM já conhecidos foram incorporados à linguagem. O mesmo ocorre com a linguagem de programação WHILE, que pode ser estendida considerando macros (veja as considerações finais do capítulo 4). Isso não somente mostra como um programa RAM (ou WHILE) pode ser construída de partes mais simples, mas destaca um aspecto negativo ao se trabalhar com modelos de baixo nível, i.e. modelos que possuem instruções muito primitivas, pois enquanto se necessita de pouca imaginação para construir novos programa RAM utilizando macro-instruções (como o programa SQRT do exemplo 2.10 que usa os macro-comandos SQR e DIST), fazê-los sem elas tomaria muito tempo ou facilmente levaria a erros de implementação e não acrescentaria nada ao entendimento. O conjunto de instruções da máquina RAM é tão restrito que qualquer argumento, solução ou prova matemática de que um problema não trivial pode ser implementado na RAM seria extremamente trabalhoso.

Apesar dessa dificuldade, se quer argumentar que as máquinas RAM podem efetuar não somente operações simples para as quais são oferecidos programas explícitos, mas também processos mais complexos. Observe que adicionar novos comandos às máquinas RAM, que podem ser computados pelas próprias RAM, não incrementa o poder computacional das máquinas, só as tornam mais manipuláveis. Por outro lado, pode-se perceber que essas máquinas são capazes de realizar tarefas mais complexas através do artifício da macro-instrução, mas fica a pergunta do quão complexas podem ser? ou em outras palavras, qual o seu verdadeiro poder de computação? Desse modo, seria desejável achar uma maneira de efetuar uma discussão razoavelmente rigorosa que conduzisse a conclusão de que um determinado processo é computável em alguma máquinas RAM sem ter que escrever o código de baixo nível, mas infelizmente não existe maneira completamente satisfatória de fazer isso. A saída para esse problema é a aplicação da tese de Church-Turing, que será apresentada mais adiante.

Suponha que, em algum sentido, as máquinas RAM tem o mesmo poder de computação que qualquer computador típico. Como pode-se defender ou refutar essa hipótese? Para defendê-la seria necessário se tomar uma seqüência de problemas crescentemente mais complicados e mostrar como eles são resolvidos por máquinas RAM. Deveria-se também tomar o conjunto de instruções da linguagem de máquina de algum computador específico e projetar uma máquina RAM que pudesse efetuar todas as instruções no conjunto. Embora isso seja totalmente enfadonho para qualquer um, é algo totalmente possível, em princípio, se a hipótese estiver correta. Entretanto, enquanto todo sucesso nessa direção fortaleceria a convicção na verdade da hipótese, ela não levaria a uma prova mas seria apenas uma evidência do fato. A dificuldade está no fato de que não está claro o que seja exatamente "um computador típico" e não se tem meios de tornar essa definição precisa. Assim, ao comparar as máquinas RAM com um computador específico, por exemplo um Pentium $XXXX$, se deixaria de fora computadores mais poderosos que certamente vão surgir no futuro (inclusive talvez com novos paradigmas computacionais).

Pode-se também abordar o problema de outra perspectiva; a saber, tentando

encontrar algum procedimento para o qual se possa escrever um programa de computador, mas para o qual também se possa mostrar que não pode existir nenhuma máquina RAM que implemente este procedimento. Se isso fosse possível teria-se uma base para rejeitar a hipótese. Entretanto, até agora ninguém foi capaz de produzir um tal contra-exemplo. Portanto, essas infrutíferas tentativas de contradição também devem ser tomadas como uma evidências circunstancial de que tal contra-exemplo não pode ser dado, e portanto que a hipótese é válida. Dessa forma, seguindo as evidências, tudo indica que as máquinas RAM são, em princípio, tão poderosas quanto qualquer computador.

Argumentos como esse levaram Alonzo Church e Alan Turing, em torno de 1936, de maneira independente, e usando formalismos diferentes*, à hoje conhecida **Tese de Church-Turing**†. O texto abaixo é uma citação de Andrew Hodges [15] sobre Alan Turing [31] :

> "Diz-se que uma função é "efetivamente calculável" se seus valores podem ser determinados por um processo puramente mecânico. Embora seja relativamente fácil captar intuitivamente essa idéia, é contudo desejável dispor de alguma definição mais precisa, matematicamente exprimível. Uma definição desta natureza foi formulada primeiro por Gödel em Princeton, em 1934 ... Tais funções foram descritas como 'recursivas gerais' por Gödel ... Outra definição de calculabilidade efetiva foi dada por Church ... que a identifica com a λ-definibilidade. O autor [ou seja, o próprio Turing] sugeriu recentemente uma definição que corresponde mais estreitamente à idéia intuitiva ... Afirmou-se acima que 'uma função é efetivamente calculável se os seus valores podem ser determinados por algum processo puramente mecânico'. Podemos interpretar este enunciado literalmente, entendendo por processo puramente mecânico um processo que poderia ser levado a cabo por uma máquina ... O desenvolvimento destas idéias conduz à definição do autor para uma função computável e a uma identificação da computabilidade [no sentido técnico preciso de Turing] com a calculabilidade efetiva. Não é difícil, embora trabalhoso, provar que estas três definições são equivalentes."

Pode-se perceber nesta citação, que na época já se tinha a noção intuitiva de que um procedimento efetivo era um procedimento capaz de ser realizado por uma máquina. Alan Turing propôs uma máquina conceitual, chamada **máquina de Turing**, para ser a "máquina" a qual se refere o conceito intuitivo de função efetivamente calculável. A proposta de Turing estabeleceu o princípio do computador moderno e deu origem a ciência da computação. A sua proposta mostrou-se equivalente as propostas de Church e Gödel, entretanto o modelo de máquina de Turing, difere das demais, pois, num certo sentido, deixa de lado o universo formal e coloca

*Enquanto Turing usou suas máquinas, Church usou um modelo chamado de funções λ-definíveis.

†Esta tese também é conhecida como "*Tese de Church*" ou "*Tese de Turing*".

em cena o mundo físico como uma alegação do que pode ser feito.

A equivalência entre as propostas é obtida do fato delas "computarem" a mesma classe de funções; a saber a classe das funções recursivas parciais — $\mathbb{FRP}$. Partindo disso, pode-se concluir que as RAM são também equivalentes à esses modelos, e portanto que é permitido se reescrever a tese de Church-Turing da seguinte maneira:

Tese de Church-Turing. "Qualquer computação que pode ser efetuada por meios mecânicos pode ser efetuada por uma máquina RAM".

É importante ter em mente o que é a tese de Church-Turing. Ela não é algo que possa ser provado. Para isso seria necessário uma definição precisa do termo "meios mecânicos" e isso requereria algum outro modelo abstrato que não levaria muito mais longe do que o anterior. Esta tese é vista mais apropriadamente como uma definição do que constitui um procedimento efetivo; ou seja:

Definição 5.3 *Um* ***procedimento efetivo que calcula uma função*** $f : \mathbb{N}^m \longrightarrow \mathbb{N}^n$ *é um programa RAM P que computa f. Uma função chama-se* ***função efetiva, efetivamente calculável, ou função computável*** *se ela é uma função para a qual existe um programa RAM que a computa. Se P é um procedimento efetivo que calcula f e f é uma função total, então P chama-se* ***algoritmo*** *que calcula f.*

Assumir a tese de Church-Turing como uma definição, deixa em aberto a questão do quanto essa definição é suficientemente abrangente, ou seja, com relação aos computadores ela é suficiente para cobrir tudo o que foi feito até hoje e o que é concebido como um futuro factível? Um "sim" inequívoco não é possível, mas as evidências em seu favor são muito fortes. Alguns argumentos para aceitar essa tese como definição são os seguintes:

1. O resultado fundamental: Muitas propostas independentes (por exemplo as propostas de Gödel, Church e Turing citadas acima) para uma formulação precisa da idéia intuitiva de procedimento efetivo tem conduzido a mesma classe de funções; a classe das funções recursivas parciais.

2. Uma vasta coleção de funções efetivamente computáveis tem sido mostrada ser recursiva parcial.

3. A implementação de um programa P numa máquina RAM para computar uma função é claramente um exemplo de um procedimento efetivo; assim diretamente da classe das funções RAM-computáveis $\mathbb{RAM}$, pode-se perceber que todas as funções nessa classe são computáveis no sentido informal.

4. Ninguém até agora encontrou uma função que pudesse ser aceita como computável no sentido informal, que não fosse RAM-computável.

Esses argumentos não podem ser utilizados para provar a tese de Church-Turing, eles são apenas evidências, ou seja, a tese desempenha o mesmo papel em ciência da computação como fazem as leis básicas da física e da química. A física clássica, por exemplo, é baseada fortemente nas leis do movimento de Newton. Embora sejam chamadas de leis, elas não são logicamente necessárias. Em contrapartida elas são modelos plausíveis que explicam o mundo físico. São aceitas porque as conclusões que são tiradas a partir delas concordam com a experiência e a observação da realidade. Semelhantemente, elas não podem ser provadas verdadeiras, embora possam ser refutadas por um experimento que resulte na contradição de alguma conclusão baseada nas mesmas, a partir disso começaria-se a questionar a validade dessas leis. Por outro lado, repetidos insucessos de refutação de uma lei fortalece a confiança nela. Esta é a situação para a tese de Church-Turing. Portanto, não há problema em considerá-la uma lei básica da ciência da computação, pois as conclusões que se tiram a partir dela concordam com o que se sabe acerca dos computadores reais. Entretanto, existe sempre a possibilidade de que alguém apareça com outra definição que explique alguma situação sutil que não possa ser coberta por máquinas RAM, mas que ainda caia no escopo da noção intuitiva de procedimento efetivo. Em tal eventualidade, algumas das discussões subseqüentes teriam que ser consideravelmente modificadas. Algumas discussões mais aprofundadas sobre a validade da tese de Church-Turing, considerando argumentos a favor e contra, podem ser encontradas em [2, 28, 30]. Com base nas evidências, daqui por diante assume-se a tese de Church-Turing, e portanto a definição de procedimento efetivo em termos de máquinas RAM.

Identificar um procedimento efetivo com um programa RAM permite provar rigorosamente argumentos como "existe um procedimento efetivo..." ou "não existe nenhum procedimento efetivo...". Entretanto, para construir explicitamente um procedimento desse tipo, mesmo para problemas relativamente simples, pode ser bastante trabalhoso. Para evitar isso pode-se apelar para a tese de Church-Turing e alegar que qualquer coisa que pode ser realizada com qualquer computador pode também ser feita por uma máquina RAM. Conseqüentemente, pode-se substituir as máquinas RAM por um programa em JAVA, por exemplo, na definição 5.3. Isto facilita consideravelmente a exibição do tal procedimento, visto que a linguagem em questão é mais poderosa que as máquinas RAM, entretanto o leitor deve ter em mente que é a tese de Church-Turing que suporta a possibilidade de se escrever uma máquina RAM para o tal procedimento, já que ele pode ser implementado num computador concreto.

5.3 Considerações finais

Considere, agora, o seguinte argumento contra a tese de Church-Turing:

> "A máquina RAM é um computador de propósito específico, pois uma vez que um programa RAM P seja definido (instalado na máquina), ela fica restrita a efetuar a computação particular que esse programa faz.

> Os computadores digitais, por outro lado, são máquinas de propósito geral, i.e. máquinas que podem ser programadas para fazer diferentes tarefas em tempos diferentes. Conseqüentemente, nenhuma máquina RAM pode ser considerada equivalente aos computadores digitais de propósito geral."

No entanto, esta objeção pode ser superada projetando-se um programa que "imita" todos os programas, o chamado **programa RAM universal**. O que é e como seria construído um programa universal para máquinas RAM é uma assunto para o próximo capítulo.

Capítulo 6

Numeração de Gödel e Programas universais

Este capítulo mostra que programas podem ser armazenados na memória da RAM para serem processados. Dessa forma, a RAM, como qualquer PC, é um computador de propósito geral. A idéia por trás é a mesma de que programas são cadeias binárias carregadas na memória do computador e simulados pela CPU. O conceito matemático associado à CPU é o de programa universal.

No que segue, essas idéias são formalizadas. Mostra-se também que o conjunto de todos os programas RAM é um conjunto enumerável, e portanto tem a mesma quantidade de elementos que o conjunto dos números naturais. Isso dá origem ao primeiro limite da computação nas RAM; a saber que existem mais funções do que programas RAM para computá-las, o que significa, em outras palavras, que não se pode fazer tudo com as RAM.

A enumeração aqui proposta baseia-se nas propriedades de números primos, que permitirão a transformação de seqüências finitas de números naturais em um único número natural. Além da transformação de programas em números naturais, apresenta-se o conceito de cardinalidade e enumerabilidade.

6.1 Enumeração efetiva

Algumas referências bibliográficas introduzem o conceito de **conjuntos efetivamente enumeráveis** (veja Cutland [8] p.73). Um conjunto X é efetivamente enumerável se existe uma bijeção $f : X \to \mathbb{N}$ tal que f e f^{-1} são funções efetivamente computáveis. Quando $X \subseteq \mathbb{N}^m$, para algum $m \geq 0$, não há qualquer problema, visto que para mostrar que X é efetivamente enumerável é suficiente reescrever a bijeção f e a sua inversa f^{-1} como funções recursivas parciais, ou, equivalentemente, encontrar um programa RAM que as implemente. Na verdade isso acontecerá mais

adiante neste capítulo (seção 6.3) quando for estabelecido, por exemplo, que o conjunto de todas as seqüências finitas de números naturais, $\bigcup_{k>o} \mathbb{N}^k$, é efetivamente enumerável. Uma outra maneira nesses casos seria mostrar que existe uma função recursiva parcial $f : \mathbb{N}^m \to \mathbb{N}$ tal que $dom(f) = X$ — veja Brainerd [4].

O "problema" surge quando X não é parte de $\mathbb{N}^m$. Ou seja a natureza dos objetos não está ligada aos números naturais. Por exemplo, para o conjunto dos números inteiros existe uma bijeção "efetiva"com os números naturais. Mas também existem subconjuntos dos inteiros para os quais toda bijeção entre eles e os números naturais é "não efetiva". Só que os modelos usados para capturar a noção de procedimento efetivo geralmente só trabalham com funções de $\mathbb{N}^m$ em $\mathbb{N}^n$. Todavia, a idéia de fundo é a mesma para o caso $\mathbb{N}^m$, i.e. X é recursivamente enumerável se existe um meio efetivo capaz de gerar todos os elementos de X. Entretanto, por questões de espaço isso não será aprofundado neste texto, apenas se mencionará que X é efetivamente enumerável, e quando isso acontecer o leitor deve imaginar que existem meios efetivos capazes de transformar X em $\mathbb{N}$ e vice-versa. Por exemplo, quando X é o conjunto de programas RAM (i.e. um conjunto de seqüências de instruções RAM) pode-se imaginar que existe um programa que analisa cada instrução na seqüência e a transforma num número natural, transformando uma seqüência de instruções $(I_1, \ldots, I_s)$ numa seqüência de naturais $(a_1, \ldots, a_s)$ que em seguida é transformada no número natural n. Assim, no que segue apresenta-se bijeções que indicam que o conjunto de todas as instruções RAM, $\mathcal{I}$, e o conjunto de todos os programas RAM,$\mathbb{P}$, são efetivamente enumeráveis, e como conseqüência existe uma método efetivo capaz de armazenar cada elemento destes conjuntos na memória do computador sob a forma de número natural.

6.2 Números primos e algumas funções recursivas

Os dois teoremas que seguem suportam o desenvolvimento deste capítulo. O primeiro deles garante a existência de tantos números primos quantos forem necessários, enquanto que o segundo estabelece a relação entre números naturais e seqüências finitas de números naturais.

Teorema 6.1 *Existem infinitos números primos.*

Teorema 6.2 *Sejam os números primos dispostos em ordem crescente de magnitude: $p_0, p_1, \ldots, p_n, \ldots$ (i.e. $p_0 = 2, p_1 = 3, p_2 = 5, \ldots$). Todo inteiro positivo "a" pode ser fatorado num produto de números primos que é único dentro da ordem dos fatores.*

Isso significa que para todo número natural $a > 0$,

$$a = p_0^{a_0} \cdot p_1^{a_1} \cdot \cdots \cdot p_i^{a_i} \cdot \ldots \tag{6.1}$$

onde a_i é o número de vezes que p_i ocorre como fator de a. Note que se p_i não é um fator de a, então $a_i = 0$, o que justifica o produto infinito acima e caracteriza uma **representação infinita para números naturais**.

Exemplo 6.3

1. $18 = 2^1 \cdot 3^2 \cdot 5^0 \cdot 7^0 \cdot \ldots$
2. $13 = 2^0 \cdot 3^0 \cdot 5^0 \cdot 7^0 \cdot 11^0 \cdot 13^1 \cdot 17^0 \cdot \ldots$
3. $100 = 2^2 \cdot 3^0 \cdot 5^2 \cdot 7^0 \cdot \ldots$
4. $16 = 2^4 \cdot 3^0 \cdot 5^0 \cdot 7^0 \cdot \ldots$

Observe, entretanto, que existe no máximo uma quantidade finita de $a_j's$ que caracterizam o natural $a > 0$ e o restante dos expoentes será zero. Isso dá origem a seguinte proposição:

Proposição 6.4 *Considerando os números primos em ordem de magnitude: $p_0, p_1, p_2, \ldots, p_n, \ldots$, para cada natural positivo a existe uma única seqüência finita de números naturais $(a_0, a_1, \ldots, a_n)$ que corresponde, respectivamente, aos expoentes de $p_0, p_1, p_2, \ldots, p_n$ na decomposição prima de a — ou seja $a = p_0^{a_0} \cdot p_1^{a_1} \cdot \cdots \cdot p_n^{a_n} \cdot p_{n+1}^0 \cdot p_{n+2}^0 \cdot \ldots$. Inversamente, associado a cada seqüência $(a_0, a_1, \ldots, a_n)$ de números naturais está associado um único número natural a tal que $a = p_0^{a_0} \cdot p_1^{a_1} \cdot \cdots \cdot p_n^{a_n}$.*

Exemplo 6.5

(a) 18 *está associado a* $(1, 2)$

(b) 13 *está associado a* $(0, 0, 0, 0, 0, 1)$

(c) 100 *está associado a* $(2, 0, 2)$

(d) 16 *está associado a* 4

A proposição que segue mostra que algumas funções que serão necessárias para o desenvolvimento deste capítulo são recursivas primitivas.

Proposição 6.6 (Divisibilidade e recursão primitivas) *As seguintes funções são recursivas primitivas:*

1. $D(x) =$ *ao número de divisores de x; onde por convenção* $D(0) = 1$
2. $Pr(x) = \begin{cases} 1, & \text{se } x \text{ é primo;} \\ 0, & \text{se } x \text{ não é primo.} \end{cases}$
3. $p_n =$ *ao n-ésimo numero primo; onde por convenção* $p_0 = 0$, *e obviamente* $p_1 = 2, p_2 = 3$, *etc*
4. $(x)_n = \begin{cases} \text{ao expoente de } p_n \text{ na decomposição prima de } x, \text{ para } x, n > 0, \\ 0, \text{ se } x = 0 \text{ ou } n = 0. \end{cases}$

Prova:

1. $D(x) = \sum_{y \leq x} div(y, x)$
2. $Pr(x) = \overline{sg}(\mid D(x) - 2 \mid)$
3. $\begin{cases} p_0 = 0 \\ p_{n+1} = (\mu_{z \leq (p_n!+1)})(z > p_n \text{ e } Pr(z) = 1.) \end{cases}$
4. $(x)_n = (\mu_{z<x})(\neg div(p_n^{z+1}, x))$

□

6.3 Numeração de Gödel

Como mencionado anteriormente, esta seção se encarregará de mostrar que um programa RAM pode ser transformado num número natural, graças às funções e propriedades de números primos vistas acima, e por conseguinte pode ser carregado na memória das RAM. Esta seção juntamente com a que segue mostra a existência de programas RAM que fazem o papel da CPU em computadores convencionais — os programas universais — demonstrando que a arquitetura da RAM é uma arquitetura de propósito geral.

Antes de desenvolver o capítulo, é necessário demonstrar que algumas funções auxiliares são recursivas primitivas.

Números naturais na base 2 e funções auxiliares. Todo número natural x pode ser escrito na base dois da seguinte maneira: $x = \sum_{i=0}^{\infty} \alpha(i, x)2^i$; onde $\alpha(i, x) = 0$ ou $\alpha(i, x) = 1$. Por exemplo, o número 35 pode ser reescrito como a seguinte expressão: $1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + \ldots$. Entretanto, observe que $\alpha(0, 35) = \alpha(1, 35) = \alpha(5, 35) = 1$, enquanto que $\alpha(j, 35) = 0$, para $j \neq 0, 1, 5$. Isso significa que existe uma quantidade finita, $l(x)$, de i's tal que $\alpha(i, 35) = 1$; no caso $l(35) = 3$. Assim, o número 35 poderia ser escrito com uma quantidade finita de parcelas; ou seja $l(35)$ parcelas: $35 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^5$. Fazendo b_j igual ao j-ésimo expoente na soma finita: "$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^5$", então $b_1 = 0$, $b_2 = 1$, e $b_3 = 5$. Assim, 35 pode ser, então, reescrito como um somatório finito sem parcelas nulas: "$2^{b_1} + 2^{b_2} + 2^{b_{l(35)}}$". Generalizando esses fatos, se $x > 0$ e $l(x) = l$, onde l é a quantidade de posições i tal que $\alpha(i, x) = 1$, então pode-se reescrever x segundo a seguinte equação:

$$x = 2^{b_1} + 2^{b_2} + \cdots + 2^{b_l} \tag{6.2}$$

Sendo assim, dado um número natural x, tem-se associado as seguintes funções:

1. $\alpha : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, onde $\alpha(i, x)$ é α_i na expressão $x = \sum_{i=0}^{\infty} \alpha_i \cdot 2^i$

2. $l : \mathbb{N} \to \mathbb{N}$, onde $l(x) = \begin{cases} l, & \text{como em (6.2), se } x > 0 \\ 0, & \text{caso contrário.} \end{cases}$

3. $b : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, onde $b(i,x) = \begin{cases} b_i, & \text{como em (6.2), se } x > 0 \text{ e } 1 \leq i \leq l(x) \\ 0, & \text{caso contrário.} \end{cases}$

Proposição 6.7 *As funções $\alpha(i,x)$, $l(x)$, e $b(i,x)$ são recursivas primitivas.*

Prova:

1. Como $x = \sum_{i=0}^{\infty} \alpha(i,x)2^i$, então $qt(2^i,x) = \alpha(i,x) + 2 \cdot \alpha(i+1,x) + \ldots$, e portanto $\alpha(i,x) = rm(2, qt(2^i,x))$

2. $l(x)$ é o número de i's tal que $\alpha(i,x) = 1$; logo $l(x) = \sum_{i<x} \alpha(i,x)$

3. Se $x > 0$, então $x = 2^{b(1,x)} + 2^{b(2,x)} + \cdots + 2^{b(l(x),x)}$, assim, se $1 \leq i \leq l(x)$, então $b(i,x)$ é o i-ésimo índice k tal que $\alpha(k,x) = 1$. Portanto,

$$b(i,x) = \begin{cases} (\mu_{z<x})\Big(\sum_{k\leq y} \alpha(k,x) = i\Big), & \text{se } 1 \leq i \leq l(x) \text{ e } x > 0; \\ 0, & \text{caso contrário.} \end{cases}$$

□

Números naturais positivos e seqüências. Assim, como a todo número natural positivo $x > 0$ está associado uma única seqüência de números naturais $b_1, b_2, \ldots, b_{l(x)}$, tal que $x = 2^{b_1} + 2^{b_2} + \cdots + 2^{b_{l(x)}}$, então dada uma seqüência finita de números naturais $(a_1, \ldots, a_k)$, existe uma única expressão para x associada a $(a_1, \ldots, a_k)$; a saber:

$$x = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \cdots + 2^{a_1+a_2+\cdots+a_k+(k-1)} \tag{6.3}$$

Dessa forma, $a_1 = b_1$ e $a_{i+1} = b_{i+1} - b_i - 1$. Portanto, dado um número positivo x, a i-ésima componente da seqüência $(a_1, \ldots, a_k)$ associada a x pela expressão acima, pode ser obtida através da seguinte função recursiva primitiva: $a : \mathbb{N} \times \mathbb{N}^+ \to \mathbb{N}$, onde:

$$\begin{cases} a(i,x) = b(i,x), & \text{se } i = 0 \text{ ou } i = 1; \\ a(i+1,x) = (b(i+1,x) \dot{-} b(i,x)) \dot{-} 1, & \text{se } i \geq 1. \end{cases} \tag{6.4}$$

Para transformar um programa num número natural (número de Gödel), o programa será primeiramente transformado numa seqüência de números naturais que posteriormente será transformada no natural desejado. A proposição que segue demonstra que o passo da transformação da seqüência de naturais num número natural, de fato é um processo efetivo pois pode ser expresso numa função recursiva primitiva, além disso introduz-se duas bijeções efetivas que são necessárias para especificar uma bijeção efetiva entre o conjunto dos programas e o conjunto das seqüências finitas de naturais.

Proposição 6.8 *Os seguintes conjuntos são efetivamente enumeráveis:*

1. $\mathbb{N} \times \mathbb{N}$
2. $\mathbb{N} \times \mathbb{N}^+ \times \mathbb{N}$
3. $\bigcup_{k>0} \mathbb{N}^k$ — *o conjunto de todas as seqüências finitas de números naturais*

Prova:

1. A função $\eta : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, definida por $\eta(m,n) = 2^m(2n+1) - 1$, é uma bijeção recursiva primitiva. A inversa $\eta^{-1} : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, dada por $\eta^{-1}(x) = (\eta_1(x), \eta_2(x))$, é uma função recursiva primitiva, pois $\eta_1(x)$ e $\eta_2(x)$ são funções recursivas primitivas definidas por $\eta_1(x) = (x+1)_1$ e $\eta_2(x) = \frac{1}{2}\left(\frac{x+1}{2^{\eta_1(x)}} - 1\right)$.

2. A função $\delta : \mathbb{N} \times \mathbb{N}^+ \times \mathbb{N} \to \mathbb{N}$, definida por $\delta(m,n,q) = \eta(\eta(m,n-1),q)$ é uma bijeção recursiva primitiva (c.f. o ítem anterior). A inversa $\delta^{-1} : \mathbb{N} \to \mathbb{N} \times \mathbb{N}^+ \times \mathbb{N}$, definida por $\delta^{-1}(x) = (\eta_1(\eta_1(x)), \eta_2(\eta_1(x)) + 1, \eta_2(x))$ é recursiva primitiva.

3. A função $\tau : \bigcup_{k>0} \mathbb{N}^k \to \mathbb{N}$ definida por:

$$\tau(a_1, \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \cdots + 2^{a_1+a_2+\cdots+a_k+(k-1)} - 1 \tag{6.5}$$

é uma bijeção recursiva primitiva. Da própria expressão, pode-se concluir que ela é recursiva primitiva. Observe que a expressão $2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \cdots + 2^{a_1+a_2+\cdots+a_k+(k-1)}$ designa um número positivo, assim para que o 0 seja imagem de uma seqüência foi necessário a subtração de uma unidade nesta expressão — na verdade $0 = \tau(0) = 2^0 - 1$. Por outro lado, dado um número positivo x a função $\tau^{-1}(x)$ é calculada da seguinte forma: Como x possui uma única representação binária, pode-se encontrar unicamente $k \geq 1$ números naturais $b_1, \ldots, b_k$, tal que $0 \leq b_1 < b_2 < \cdots < b_k$ e $x+1 = 2^{b_1} + 2^{b_2} + \cdots + 2^{b_k}$, a partir dos quais calcula-se $\tau^{-1}(x) = (a_1, \ldots, a_n)$, onde $a_1 = b_1$ e $a_{n+1} = b_{n+1} - b_n - 1$ $(1 \leq n < k)$.

□

Exemplo 6.9

1. $\tau(0,2,3) = 2^0 + 2^{0+2+1} + 2^{0+2+3+2} - 1 = 1 + 8 + 128 - 1 = 136$
2. $\tau^{-1}(136)$ *é calculado da seguinte maneira: calcule o sucessor de* 136*, i.e.* 137*; em seguida encontre a expansão literal de* 137 *na base* 2*, no caso* $137 = 1 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7$*. Portanto,* $137 = 2^0 + 2^3 + 2^7$*; fazendo* $b_1 = 0, b_2 = 3$ *e* $b_3 = 7$*, recupera-se a seqüência anterior* $(0,2,3)$ *do seguinte modo:* $a_1 = b_1$*,* $a_2 = b_2 - b_1 - 1$ *e* $a_3 = b_3 - b_2 - 1$*.*

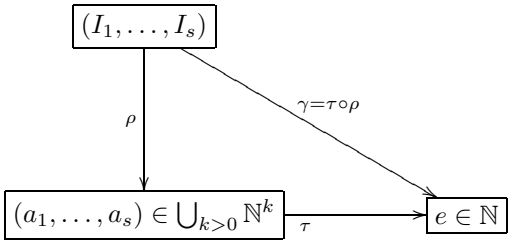

Figura 6.1: Esquema da numeração de Gödel

6.3.1 Numeração de programas

Seja P um programa RAM visto como uma seqüência de instruções $(I_1, \ldots, I_s)$, $(a_1, \ldots, a_s)$ uma seqüência de números naturais e "e" um número natural. Então o processo de numeração de Gödel que segue pode ser visualizado na figura 6.1.

Ou seja, a enumeração do conjunto dos programas se dá através da transformação de um programa numa seqüência de números naturais através da função ρ que é seguida pela transformação da seqüência resultante no número natural correspondente. A proposição 6.8 garante a segunda parte da prova. Como um programa é uma seqüência de instruções, e portanto uma seqüência de objetos que não são números naturais, a bijeção efetiva "ρ" esperada é apresentada a seguir, entretanto como já mencionado, por questões de espaço não há como expressar matematicamente a efetividade da função $\rho : \mathbb{P} \to \bigcup_{k>0} \mathbb{N}^k$, onde $\mathbb{P}$ é o conjunto dos programas, pois seria necessário todo um desenvolvimento para isso. Entretanto, apela-se para a noção informal de efetividade; pode-se por exemplo pensar em ρ como sendo um compilador (um programa) que lê programas RAM e transforma cada instrução do mesmo em números naturais. Partindo disso, pode-se concluir que a **numeração de Gödel**, nada mais é do que a aplicação da bijeção $\tau \circ \rho : \mathbb{P} \to \mathbb{N}$. No que segue mostra-se como ρ pode ser definida.

Para simplificar as provas, a quantidade de instruções básicas da RAM é reduzida, pois nem todas as instruções básicas até aqui apresentadas são primitivas, algumas instruções básicas podem ser obtidas de outras funções básicas, mas, por questão de simplicidade de programação, se preferiu não adotar antes um conjunto mínimo de instruções primitivas para a RAM.

Proposição 6.10 *Para todo programa RAM P, existe um outro programa P' que computa as mesmas funções, e tal que P' não possui as instruções* `CLR Ri`, `Ri <- Rj`, `JMP Nia`, *e* `JMP Nib`.

Prova: Suponha que P é um programa RAM. No que segue elimina-se a cada passo uma das instruções acima. No primeiro passo eliminam-se os jump's incon-

dicionais `JMP Nia` e `JMP Nib`. Escolha o menor número natural n tal que `Rn` não é referenciado por P. Construa o seguinte programa, $P2$, a partir de P substituindo cada "`Nk JMP Nix`", onde "`Nix`" é '`Nia`" ou '`Nib`", pelo seguinte pedaço de código RAM:

```
Nk CLR Rn
   Rn JMP Nix
```

Para eliminar a instrução `Ri <- Rj`, escolha os menores números naturais m e n, tal que $P2$ não faça referência aos registradores `Rm` e `Rn`. Sejam `Nc` e `Nd` dois rótulos que não sejam utilizados em $P2$. Forme então o seguinte programa $P3$ a partir de $P2$ substituindo "`Nk Ri <- Rj`" pelo seguinte código RAM:

```
Nk CLR Ri
   CLR Rn
   CLR Rm
Nc Rj JMP Ndb
   DEC Rj
   INC Ri
   INC Rn
   Rm JMP Nca
Nd Rn JMP Ncb
   DEC Rn
   INC Rj
   Rm JMP Nda
Nc CONTINUE
```

Finalmente elimina-se a instrução `CLR`. Seja Nc um rótulo não utilizado pelo programa $P3$. Escolha um n tal que nenhum registrador `Rm` é referenciado por $P3$ para qualquer $m \geq n$. Isso garante que o registrador `Rn` inicialmente conterá zero. Finalmente construa P' a partir de $P3$ substituindo cada instrução "`Nk CLR Ri`" pelo seguinte código RAM:

```
Nk Ri JMP Ncb
   DEC Ri
   Rn JMP Nka
Nc CONTINUE
```

□

Utilizando o conjunto minimal de instruções, descreve-se agora como codificar um programa RAM em um único número natural. O processo que segue, encarrega-se de especificar a função $\rho : \mathbb{P} \to \bigcup_{k>0} \mathbb{N}^k$. Para isso, é necessário descrever como cada instrução deve ser individualmente codificada num número natural.

Proposição 6.11 *O conjunto das instruções $\mathcal{I}$ é efetivamente enumerável.*

Prova: O método efetivo que transformará cada instrução da RAM num número natural é interpretado na bijeção $\beta : \mathcal{I} \to \mathbb{N}$, onde:

1. $\beta(\texttt{Ni INC Rj}) = 5 \cdot \eta(i, j-1)$
2. $\beta(\texttt{Ni DEC Rj}) = 5 \cdot \eta(i, j-1) + 1$
3. $\beta(\texttt{Ni CONTINUE}) = 5 \cdot i + 2$
4. $\beta(\texttt{Ni Rj JMP Nka}) = 5 \cdot \delta(i, j, k) + 3$
5. $\beta(\texttt{Ni Rj JMP Nkb}) = 5 \cdot \delta(i, j, k) + 4$

□

A função estabelece, portanto, que qualquer bijeção efetiva que seja implementada entre o conjunto das instruções e o conjunto dos números naturais deverá mapear os cinco tipos de instrução, respectivamente, em números naturais da forma $5n$, $5n+1$, $5n+2$, $5n+3$, e $5n+4$, e o valor resultante deverá obedecer as expressões analíticas acima.

A decodificação de um número natural numa instrução RAM, obedece a função $\beta^{-1} : \mathbb{N} \to \mathcal{I}$ a seguir. Como para cada $x \in \mathbb{N}$, existe um único $q \in \mathbb{N}$ e um único $r \in \mathbb{N}$ tal que $x = 5 \cdot q + r$ e $0 \leq r < 5$, então o valor de r indica o tipo da instrução codificada, enquanto que o quociente q contém informações referentes ao rótulo e possivelmente ao registrador referenciado ou ao rótulo de destino (no caso de desvios condicionais). Assim, a decodificação da instrução a partir de um número natural é especificada da seguinte maneira: Dado $x \in \mathbb{N}$ e $q = qt(5, x)$,

1. se $rm(5, x) = 0$, então $\beta^{-1}(x) = \texttt{N}\eta_1(q)\ \texttt{INC R(}\eta_2(q) + 1\texttt{)}$
2. se $rm(5, x) = 1$, então $\beta^{-1}(x) = \texttt{N}\eta_1(q)\ \texttt{DEC R(}\eta_2(q) + 1\texttt{)}$
3. se $rm(5, x) = 2$, então $\beta^{-1}(x) = \texttt{N}q\ \texttt{CONTINUE}$
4. se $rm(5, x) = 3$, então $\beta^{-1}(x) = \texttt{N}U_1^3(\delta^{-1}(q))\ \texttt{R(}U_2^3(\delta^{-1}(q))\texttt{)}\ \texttt{JMP N}U_3^3(\delta^{-1}(q))\texttt{a}$
5. se $rm(5, x) = 4$, então $\beta^{-1}(x) = \texttt{N}U_1^3(\delta^{-1}(q))\ \texttt{R(}U_2^3(\delta^{-1}(q))\texttt{)}\ \texttt{JMP N}U_3^3(\delta^{-1}(q))\texttt{b}$

Se o programa RAM que se está tentando codificar possui instruções sem rótulo, então coloque nestas instruções o menor rótulo que não está sendo usado pelo programa.

Exemplo 6.12 *Segundo essa regra, o programa:*

```
N0 R2 JMP N1b
    INC R1
    DEC R2
N1 CONTINUE
```

é transformado no programa equivalente abaixo, para posterior codificação:

```
N0 R2 JMP N1b
N2 INC R1
N2 DEC R2
N1 CONTINUE
```

Proposição 6.13 *O conjunto de todos os programas $\mathbb{P}$ é efetivamente enumerável.*

Prova: O método efetivo que transformará um programa num número natural é interpretado na bijeção $\gamma : \mathbb{P} \to \mathbb{N}$, onde dado um programa RAM $P = (I_1, \ldots, I_s)$, no formato discutido anteriormente, $\gamma(P) = \tau(\beta(I_1), \ldots, \beta(I_s))$. γ é uma bijeção efetiva pois as funções τ, τ^{-1}, β e β^{-1} são bijeções que interpretam métodos efetivos. Observe que a função $\rho : \mathbb{P} \to \bigcup_{k>0} \mathbb{N}^k$ é definida por $\rho(I_1, \ldots, I_s) = (\beta(I_1), \ldots, \beta(I_s))$ e $\gamma = \tau \circ \rho$. □

Definição 6.14 *Dado um programa RAM P, o valor $\gamma(P)$ chama-se **código de Gödel** ou **número de Gödel** de P. Define-se $\mathbf{P_n}$ como sendo o programa cujo número de Gödel é n.*

Dessa forma, dado um programa P, pode-se efetivamente encontrar o número de Gödel $\gamma(P)$, e dado um número natural n, pode-se efetivamente encontrar o programa P_n, obedecendo assim o esquema da figura 6.1.

Exemplo 6.15 *Tome o programa P*

```
N0 INC R1
N1 INC R2
N2 CONTINUE
```

então $\beta($`N0 INC R1`$) = 5 \cdot \eta(0, 1-1) = 5 \cdot \eta(0, 0) = 5 \cdot (2^0(2 \cdot 0+1)-1) = 5 \cdot (1-1) = 0$, $\beta($`N1 INC R2`$) = 5 \cdot \eta(1, 1) = 5 \cdot (6-1) = 25$, e $\beta($`N2 CONTINUE`$) = 5 \cdot 2 + 2 = 12$. Portanto $\rho(P) = (0, 25, 12)$ e $\gamma(P) = 2^0 + 2^{0+25+1} + 2^{0+25+12+2} - 1 = 2^0 + 2^{26} + 2^{39} - 1 = 1 + 67.108.864 + 549.755.813.888 - 1 = 549.822.922.752$. Assim o código de Gödel associado a P é $549.822.922.752$, o que é denotado por $P_{549.822.922.752}$. Inversamente, dado $549.822.922.752$, calcula-se $\gamma^{-1}(549.822.922.752)$ como segue: primeiramente calcule $\tau^{-1} : \mathbb{N} \to \bigcup_{k>0} \mathbb{N}^k$, i.e. encontre $b_1, b_2, \ldots, b_l$ tal que $0 \leq b_1 < b_2 < \cdots < b_l$ e $2^{b_1} + 2^{b_2} + \cdots + 2^{b_l} = 549.822.922.752 + 1$. Com certeza o leitor encontrará a expressão $2^0 + 2^{26} + 2^{39}$. Em seguida calcule $a_1 = b_1 = 0$, $a_2 = (b_2 - b_1) - 1 = (26 - 0) - 1 = 25$ e $a_3 = (b_3 - b_2) - 1 = (39 - 26) - 1 = 12$. Dessa forma $\tau^{-1}(549.822.922.752) = (0, 25, 12)$. Por fim, o programa listado acima é a seqüência $(\beta^{-1}(0), \beta^{-1}(25), \beta^{-1}(12))$.

Existem, obviamente, outras bijeções efetivas entre $\mathbb{P}$ e $\mathbb{N}$, a escolha aqui foi arbitrária e não obedeceu qualquer critério especial. Para a teoria que segue, qualquer outra bijeção efetiva γ' é suficiente. Todavia, é necessário fixar uma forma de numerar os programas RAM, e para isso elege-se a codificação acima, ou seja para o resto deste livro fixa-se a numeração de Gödel como sendo a função γ proposta acima.

Um primeiro limite. Uma das conclusões desta seção é que existem no máximo tantos programas quantos são os números naturais. Isso significa que é, por exemplo, impossível gerar computacionalmente todos os números reais, ou qualquer outro conjunto incontável*.

6.3.2 Numeração de funções computáveis

A partir do método de numeração proposto, pode-se enumerar também funções computáveis juntamente com o seu domínio e imagem. A seguir se introduz a notação que será utilizada no restante deste livro, esta notação segue a utilizada em Cutland [8]. O principal resultado desta seção é a existência de apenas uma quantidade enumerável de funções computáveis, reescrevendo o comentário anterior de que existem apenas uma quantidade enumerável de programas RAM; entretanto a abordagem aqui é *um pouco* mais rigorosa.

Definição 6.16 *Para cada $a \in \mathbb{N}$, e $m, n \geq 0$:*

1. $\phi_a^{(m,n)}$ *é a função* $\phi_a^{(m,n)} : \mathbb{N}^m \to \mathbb{N}^n$ *implementada por* P_a
2. $W_a^{(m,n)} = dom(\phi_a^{(m,n)})$, *i.e* $\{(x_1, \ldots, x_m) : P_a(x_1, \ldots, x_m) \downarrow\}$
3. $E_a^{(m,n)} =$ *à imagem da função* $\phi_a^{(m,n)}$, *i.e.* $\{(y_1, \ldots, y_n) : P_a(x_1, \ldots, x_m) \downarrow (y_1, \ldots, y_n)\}$.

Escreve-se ϕ_a, W_a e E_a, respectivamente no lugar de $\phi_a^{(1,1)}$, $W_a^{(1,1)}$ e $E_a^{(1,1)}$.

Exemplo 6.17 *Seja $a = 549.822.922.752$ do exemplo anterior. Sabe-se que P_a é o programa (`N0 INC R1, N1 INC R2, N2 CONTINUE`). Portanto,*

1. $\phi_a(x) = x + 1$
2. $\phi_a^{(1,2)}(x) = (x + 1, 1)$
3. $\phi_a^{(1,3)}(x) = (x + 1, 1, 0)$, *etc.*
4. $\phi_a^{(2,1)}(x, y) = x + 1$
5. $\phi_a^{(2,2)}(x, y) = (x + 1, y + 1)$
6. $\phi_a^{(2,3)}(x, y) = (x + 1, y + 1, 0)$,*etc.*
7. $W_a = \mathbb{N}$, $E_a = \mathbb{N}^+$
8. $W_a^{(1,2)} = \mathbb{N}$, $E_a = \mathbb{N}^+ \times \{1\}$

*Na verdade existem conjuntos contáveis cujos elementos não podem ser gerados computacionalmente, mas isso não será tema deste livro.

9. $W_a^{(1,3)} = \mathbb{N}$, $E_a = \mathbb{N}^+ \times \{1\} \times \{0\}$, *etc.*

Se $f : \mathbb{N} \to \mathbb{N}$ é uma função computável, então existe um programa P que computa f, e portanto $f = \phi_a$, onde $a = \gamma(P)$. Diz-se que a é um índice para f. Como existem vários programas que computam f, então f possui vários índices. Assim, toda função computável da forma $f : \mathbb{N} \to \mathbb{N}$ aparece na enumeração[†].

$$\phi_0, \phi_1, \phi_2, \ldots$$

O mesmo se aplica para funções da forma $f : \mathbb{N}^m \to \mathbb{N}^n$; para $m, n \in \mathbb{N}$; i.e. pode-se pensar numa enumeração

$$\phi_0^{(m,n)}, \phi_1^{(m,n)}, \phi_2^{(m,n)}, \ldots$$

para cada $m, n \in \mathbb{N}$.

Proposição 6.18 *Dados $m, n \in \mathbb{N}$, seja $\mathcal{C}^{(m,n)}$ o conjunto de todas as funções computáveis da forma $f : \mathbb{N}^m \to \mathbb{N}^n$, e $\mathcal{C} = \bigcup\{\mathcal{C}^{(m,n)} : m, n \in \mathbb{N}\}$. Então $\mathcal{C}^{(m,n)}$ e $\mathcal{C}$ são enumeráveis.*

Prova: Tomando a enumeração $\phi_0^{(m,n)}, \phi_1^{(m,n)}, \phi_2^{(m,n)}, \ldots$, que possui repetições, constrói-se a seguinte enumeração sem repetições:

$$\begin{cases} f(0) = 0 \\ f(k+1) = \mu_z(\phi_z^{(m,n)} \neq \phi_{f(0)}^{(m,n)} \wedge \cdots \wedge \phi_z^{(m,n)} \neq \phi_{f(k)}^{(m,n)}) \end{cases}$$

assim, $\phi_{f(0)}^{(m,n)}, \phi_{f(1)}^{(m,n)}, \phi_{f(2)}^{(m,n)} \ldots$ é uma enumeração de $\mathcal{C}^{(m,n)}$ sem termos repetidos [‡].

Como $\mathcal{C} = \bigcup \mathcal{C}^{(m,n)}$, então a enumerabilidade do conjunto $\mathcal{C}$ segue do fato que a união enumerável de conjuntos enumeráveis é um conjunto enumerável. □

Revisitando o limite anterior. O teorema que segue é na verdade uma outra versão do limite descrito anteriormente; a saber: que existe uma quantidade enumerável de programas disponíveis para a toda e qualquer computação.

Teorema 6.19 *Existe uma função $f : \mathbb{N} \to \mathbb{N}$, tal que $f \notin \mathcal{C}^{(1,1)}$, e portanto $f \notin \mathcal{C}$.*

[†]Observe que esta enumeração possui ocorrências repetidas da mesma função, já que vários programas computam a mesma função, e portanto vários índices de programas estão associados à mesma função.

[‡]A enumeração f não é efetiva. Entretanto existe uma bijeção efetiva proposta em Friedberg [11].

Prova: No que segue constrói-se uma função total f que é simultânemente diferente de toda função na enumeração $\phi_0, \phi_1, \phi_2, \ldots$ do conjunto $\mathcal{C}^{(1,1)}$. Explicitamente define-se:

$$f(k) = \begin{cases} \phi_k(k) + 1, & \text{se } \phi_k(k) \text{ estiver definido;} \\ 0, & \text{caso contrário.} \end{cases} \tag{6.6}$$

Assim, para cada $k \in \mathbb{N}$, $f(k)$ difere de $\phi_k(k)$, pois se $\phi_k(k)$ estiver definido, então f difere de ϕ_k, já que retorna o sucessor de $\phi_k(k)$. Se $\phi_k(k)$ estiver indefinido, então f difere de ϕ_k, pois $f(k)$ estará definido. Como f difere de toda função computável ϕ_k, então significa que f não aparece na enumeração acima e portanto não é uma função do conjunto $\mathcal{C}$ [§]. Logo, existem funções numéricas que não são computáveis. □

6.4 Programas universais

O computador moderno é capaz de carregar programas na memória juntamente com os seus dados e simular o comportamento descrito pelo programa sobre os dados informados. Programa e dados de entrada são carregados no formato binário e a CPU se encarrega de simular o comportamento do programa carregado. Dessa forma, pode-se pensar na CPU como sendo um programa que imita todos os programas, i.e. ela é um programa universal. A prova da existência de programas universais é um dos pilares da computabilidade e depende do processo de numeração descrito anteriormente. No que segue demonstra-se a existência de programas e funções computáveis universais.

6.4.1 Funções e programas universais

Considere a função $U(x, y)$ definida por

$$U(x, y) = \phi_x(y). \tag{6.7}$$

Num certo sentido, a função U engloba todas as funções computáveis "$\phi_0, \phi_1, \phi_2, \ldots$", pois para um k particular, a função $f : \mathbb{N} \to \mathbb{N}$ definida por:

$$f(y) = U(k, y) \tag{6.8}$$

é por transitividade a função computável ϕ_k. Diz-se então que U é a **função universal associada às funções computáveis** "$\phi_0, \phi_1, \phi_2, \ldots$". Mais geralmente, dá-se a seguinte definição:

[§]O argumento desta prova utiliza um método de prova criado por George Cantor, chamado método da diagonalização de Cantor, que é geralmente utilizado para demonstrar que certos conjuntos são enumeráveis.

Definição 6.20 *A **função universal** associada as funções computáveis "$\phi_0^{(m,n)}, \phi_1^{(m,n)}, \phi_2^{(m,n)}, \ldots$", designada por $U^{(m,n)} : \mathbb{N}^{m+1} \to \mathbb{N}^n$, é definida como*

$$U^{(m,n)}(e, x_1, \ldots, x_m) = \phi_e(x_1, \ldots, x_n) \tag{6.9}$$

Escreve-se $U(e, x)$ no lugar de $U^{(1,1)}(e, x)$.

A questão que surge é: As funções universais são computáveis?. Se sim, então isso garante a existência de **programas universais**, i.e. programas que são capazes de simular programas. A proposição que segue, demonstra a computabilidade das funções universais. A prova descreve as funções universais como funções recursivas parciais. Mais especificamente, como funções que decodificam o código de Gödel de um programa e simulam o seu comportamento diante de um argumento. Isso já era de se esperar, visto que os programas universais são versões da CPU dos nossos dias.

Proposição 6.21 *Para cada $m, n \in \mathbb{N}$, a função universal $U^{(m,n)}$ é computável.*

Prova: Na definição 2.4, foi comentado que o par ordenado (c, j), caracteriza o estado corrente de uma computação, onde c é a configuração atual dos registradores e j o próximo passo de computação a ser executado. Na verdade, como c contém as informações do conteúdo dos registradores $r_1, r_2, \ldots$, então c pode ser especificado como sendo o número $c = 2^{r_1} \cdot 3^{r_2} \cdot \ldots$. Diz-se que c é o **código de configuração** do programa de índice "e". Dessa forma, o número natural $\sigma = \eta(c, j)$, armazena a informação sobre o **estado corrente**. Note que o conteúdo r_i do registrador R_i pode ser facilmente recuperado de σ pela expressão $(\eta_1(\sigma))_1$ e j pela expressão $\eta_2(\sigma)$. Convenciona-se que se P_e parou, então $j = 0$ e c é a configuração final.

A mudança dos valores c, j e σ durante a computação de P_e, a dependência desses valores em relação ao código de Gödel e, da entrada $\vec{x}$ e o número t de passos completados é expresso através das seguintes funções: Para $\vec{x} = (x_1, \ldots, x_m)$,

1. $c_n^m(e, \vec{x}, t) =$"à configuração c após t passos da computação de $P_e(\vec{x})$ haver sido completada; ou à configuração final, se $P_e(\vec{x}) \downarrow$ em t ou menos passos.

2. $j_n^m(e, \vec{x}, t) = \begin{cases} \text{"ao número da próxima instrução de } P_e(\vec{x}), \text{ quando tenham sido completados } t \text{ passos"}, & \text{se } P_e(\vec{x}) \text{ não parou após } t \text{ passos ou menos;} \\ 0, & \text{se } P_e(\vec{x}) \downarrow \text{ em } t \text{ passos ou menos.} \end{cases}$

3. $\sigma_n^m(e, \vec{x}, t) = \eta(c_n^m(e, \vec{x}, t), j_n^m(e, \vec{x}, t))$ — i.e. o estado da computação de $P_e(\vec{x})$ após t passos.

No que segue define-se a função σ_n^m e mostra-se que ela é computável. Ao definir esta função, tem-se que $c_n^m(e,\vec{x},t) = \eta_1(\sigma_n^m(e,\vec{x},t))$ e $j_n^m(e,\vec{x},t) = \eta_2(\sigma_n^m(e,\vec{x},t))$. Logo, a computabilidade dessas funções está em função da computabilidade da função σ_n^m. Observe ainda que, em demonstrando a computabilidade de σ_n^m, se a computação de $P_e(\vec{x})$ pára, então isso ocorre em $(\mu_t)(j_n^m(e,\vec{x},t) = 0)$ passos, e a configuração final da RAM será $c_n^m(e,\vec{x},(\mu_t)(j_n^m(e,\vec{x},t) = 0))$, e portanto se terá:

$$U^{(m,n)}(e,\vec{x}) = ((c)_1,(c)_2,\ldots,(c)_n) \tag{6.10}$$

onde $c = c_n^m(e,\vec{x},(\mu_t)(j_n^m(e,\vec{x},t) = 0))$.

A função σ_n^m é recursiva primitiva. A prova que segue é uma adaptação de Cutland [8]. Define-se duas funções "$config$" e "nxt" que descrevem as mudanças em c_n^m e j_n^m durante a computação. Suponha que em algum estágio da computação de P_e o estágio corrente é $\sigma = \eta(c,j)$ e que P_e possui s instruções. Pode-se definir o efeito da j-ésima instrução de P_e no estado σ através das seguinte funções:

1. $config(e,\sigma) = \begin{cases} \text{a nova configuração após} \\ \text{a } j\text{-ésima instrução de } P_e \\ \text{ter sido executada,} & \text{se } 1 \leq j \leq ln(e); \\ c, & \text{caso contrário.} \end{cases}$

2. $nxt(e,\sigma) = \begin{cases} \text{o número da próxima instrução} \\ \text{após a } j\text{-ésima instrução de } P_e \\ \text{ter sido executada sobre} \\ \text{a configuração } c, & \text{se } 1 \leq j \leq ln(e) \text{ e;} \\ & \text{a próxima instrução} \\ & \text{existe em } P_e \\ 0, & \text{caso contrário.} \end{cases}$

Assim, σ_n^m é definida por recursão primitiva a partir de $config$ e nxt:

$$\begin{cases} \sigma_n^m(e,\vec{x},0) = \eta(2^{x_1}\cdot 3^{x_2}\cdot\ldots\cdot p_m^{x_m},1) \\ \sigma(e,\vec{x},y+1) = \eta(config(e,\sigma_n^m(e,\vec{x},t)),nxt(e,\vec{x},t))). \end{cases} \tag{6.11}$$

Observe que $\sigma_n^m(e,\vec{x},0)$ expressa o estado inicial do programa, i.e. o estado antes da primeira instrução de P_e ser executada. Note ainda que não se deve confundir a j-ésima instrução do programa $P_e = (I_1,\ldots,I_s)$ — i.e. $1 \leq j \leq s$ — com o código $\beta(I_j)$.

σ_n^m será recursiva primitiva, caso $config$ e nxt forem recursivas primitivas. Para isso, é suficiente estabelecer que as funções: ln, gn, ch, e v que seguem são recursivas primitivas:

1. $ln(e) =$ "é o número de instruções do programa P_e"

2. $gn(e,j) = \begin{cases} \text{“O código da } j\text{-ésima instrução em } P_e\text{”}, & \text{se } 1 \leq j \leq ln(e); \\ 0, & \text{caso contrário.} \end{cases}$

3. $ch(c,z) =$ “a configuração resultante quando a configuração c é operada com a instrução de código z”

4. $v(c,j,z) = \begin{cases} \text{“O número } j' \text{ da próxima instrução,} \\ \text{quando a configuração } c \text{ é operada com} \\ \text{a instrução de código } z\text{, e isso ocorre} \\ \text{como a } j\text{-ésima instrução do programa”}, & \text{se } j > 0; \\ 0, & \text{caso contrário.} \end{cases}$

Como $\sigma = \eta(c,j)$, $c = \eta_1(\sigma)$, e $j = \eta_2(\sigma)$, se essas quatro funções forem recursivas primitivas, então, as funções $config$ e nxt definidas abaixo também serão recursivas primitivas:

$$config(e,\sigma) = \begin{cases} ch(\eta_1(\sigma), gn(e,\eta_2(\sigma))), & \text{se } 1 \leq \eta_2(\sigma) \leq ln(e); \\ \eta_1(\sigma), & \text{caso contrário.} \end{cases} \tag{6.12}$$

$$nxt(e,\sigma) = \begin{cases} v(\eta_1(\sigma), \eta_2(\sigma), gn(e,\eta_2(\sigma))), & \text{se este número é} \leq ln(e); \\ 0, & \text{caso contrário.} \end{cases} \tag{6.13}$$

As funções $ln(e)$ e $gn(e)$ são recursivas primitivas. Com efeito, $ln(e) = l(e+1)$ e $gn(e,j) = a(j,e+1)$, onde l e a são as funções da proposição 6.7.

As seguinte funções também são recursivas primitivas:

1. $u : \mathbb{N} \to \mathbb{N}$, tal que se $z = \beta(\texttt{Ni INC Rj})$ ou $z = \beta(\texttt{Ni DEC Rj})$, então $u(z) = j$, para isso é suficiente fazer $u(z) = \eta_2(qt(5,z)) + 1$, sempre que $rm(5,z) = 0$ ou $rm(5,z) = 1$.

2. se $z = \beta(\texttt{Ni Rj JMP Nka})$ ou $z = \beta(\texttt{Ni Rj JMP Nkb})$, então $v_0(z) = i$, $v_1(z) = j$ e $v_2(z) = k$, para isso faça $v_0(z) = U_1^3(\delta^{-1}(qt(5,z)))$, $v_1(z) = U_2^3(\delta^{-1}(qt(5,z)))$ e $v_2(z) = U_3^3(\delta^{-1}(qt(5,z)))$, sempre que $rm(5,z) = 3$ ou $rm(5,z) = 4$.

3. $inc(c,j) =$ “a mudança na configuração c causada pelo efeito da instrução `Ni INC Rj`” $= c \cdot p_j$

4. $dec(c,j) =$ “a mudança na configuração c causada pelo efeito da instrução `Ni DEC Rj`” $= qt(p_j, c)$

5. a função $ch(c,z)$ descrita acima: $ch(c,z) = \begin{cases} inc(c,u(z)), & \text{se } rm(5,z) = 0; \\ dec(c,u(z)), & \text{se } rm(5,z) = 1; \\ c, & \text{caso contrário;} \end{cases}$

6. a função $v(c, j, z)$ descrita acima, onde:

$$v(c,j,z) = \begin{cases} j+1, & \text{se } rm(5,z) \neq 3 \text{ ou } rm(5,z) \neq 4 \\ & \text{— i.e. se } z \text{ é o código de uma} \\ & \text{instrução } \texttt{Ni INC Rj}, \texttt{Ni DEC Rj}, \\ & \text{ou } \texttt{Ni CONTINUE}; \\ j+1, & \text{se } rm(5,z) = 3 \text{ ou } rm(5,z) = 4, \\ & \text{e } (c)_{v_1(z)} \neq 0; \\ ln(e)+1, & \text{se } rm(5,z) = 3,\ (c)_{v_1(z)} = 0, \text{ e} \\ & (\overline{\mu}_k < j)(v_2(z) = v_0(gn(e,k))) = j \\ (\overline{\mu}_k < j) & \\ (v_2(z) = v_0(gn(e,k))), & \text{se } rm(5,z) = 3,\ (c)_{v_1(z)} = 0, \text{ e} \\ & (\overline{\mu}_k < j)(v_2(z) = v_0(gn(e,k))) \neq j \\ (\mu_k < ln(e)+1) & \\ (k > j \wedge v_2(z) = & \\ v_0(gn(e,k))), & \text{se } rm(5,z) = 4 \text{ e } (c)_{v_1(z)} = 0 \end{cases} \tag{6.14}$$

Observe que $v(c, j, z)$ retorna o valor $ln(e)+1$, nos casos em que a minimalização não satisfaz a igualdade $v_2(z) = v_0(gn(e, k))$, o que significa que o destino do "jump", codificado em $v_2(z)$, não é encontrado no programa, e portanto a máquina deverá parar. Isso completa a prova que a função σ_n^m é recursiva primitiva e portanto que a função universal $U^{(m,n)}$ é recursiva parcial.

□

6.5 Exercícios

1. Encontre o número de Gödel dos programas abaixo:

```
INC R1                    INC R1
INC R2                    DEC R2
CONTINUE                  CONTINUE
```

2. Encontre os programas que correspondem ao código de Gödel 100 e 453.

3. Encontre o domínio e imagem das funções ϕ_{100}, $\phi_{100}^{(2,1)}$ e $\phi_{100}^{(2,2)}$

6.6 Considerações finais

Como mencionado a prova original da existência de programas universais, proveu o princípio para o computador de propósito geral dos dias de hoje. Além disso,

uma importante conseqüência do fato da função σ_n^m ser recursiva primitiva é que é possível se provar que toda função recursiva parcial pode ser obtida de funções recursivas primitivas com no máximo uma aplicação do operador de minimalização — este importante fato é conhecido como a forma normal de Kleene.

No que segue, apresenta-se outros limites da computação além daqueles apresentados aqui. Lá mostra-se que certos problemas, não podem ser tratados computacionalmente devido à sua natureza.

Capítulo 7

Problemas Não Computáveis

Os capítulos anteriores caracterizaram o que pode ser feito com os computadores, entretanto não ficou claro o que não pode ser feito. Embora a tese de Church-Turing leve à crença de que essas limitações não são muitas, em várias ocasiões observa-se que não existe um algoritmo que resolva certos problemas que sabe-se que têm solução — ou seja, não existe uma solução efetiva para o problema. De fato, pela enumeração de Gödel das máquinas RAM, só existe uma quantidade enumerável de tais máquinas, porém a quantidade de funções parciais de $\mathbb{N}$ em $\mathbb{N}$ é incontável. Logo, existe uma quantidade incontável de funções que não podem ser calculadas por máquinas RAM. Assim, existem mais questões que não são computáveis do que computáveis. Felizmente, a maioria das funções conhecidas são calculáveis por máquinas RAM. Na verdade é difícil encontrar funções que não sejam computáveis, entretanto não se deve confundir o fato de não conhecer um algoritmo (ou máquina RAM) que compute uma determinada função com o fato de não existir um tal algoritmo.

O argumento que o poder da computação é limitado não é surpreendente. Intuitivamente, sabe-se que muitas questões vagas e especulativas requerem idéias e raciocínios bem além da capacidade de qualquer computador previsível. O que é mais interessante para o cientista da computação é que existem questões que podem ser claras e estabelecidas de forma simples, aparentando terem solução algorítimica, mas que não são computacionalmente solúveis.

Partindo dessas considerações, este capítulo apresenta os conceitos de problema de decisão e redução de problemas. São apresentados alguns problemas clássicos de indecidibilidade como o famoso **problema da parada**. Desse problema segue um número de problemas relacionados que também são indecidíveis.

7.1 Computabilidade e decibilidade

A definição 2.6, estabelece que uma função $f : \mathbb{N}^m \to \mathbb{N}^n$ é RAM-computável se existe um programa RAM P que computa o valor de f, para todos os argumentos no seu domínio, isto é $P(x_1, \ldots, x_m) \downarrow f(x_1, \ldots, x_m)$ quando $(x_1, \ldots, x_m) \in dom(f)$ e $P(x_1, \ldots, x_m) \uparrow$ caso contrário.

Por simplicidade, se estudará a classe dos **problemas de decisão**, que são os problemas de determinar se um elemento x de algum universo U pertence ou não a um determinado subconjunto A de U (ou equivalentemente, se satisfaz uma determinada propriedade); i.e. dado $x \in U$, determinar se $x \in A$. Se existir um algoritmo que receba x e dê como resultado um simples "sim", caso $x \in A$ ou "não", caso $x \notin A$, diz-se que o problema de decisão para o conjunto A é **decidível**. Se tal algoritmo não existir, então diz-se que o problema de decisão para o conjunto A é **indecidível**.

Observe que todo problema decidível é computável, no sentido de que existe um algoritmo (programa RAM) que computa uma solução para o problema, mas a recíproca não é verdadeira, isto é, existem problemas que são computáveis mas por não serem problemas de desição não são decidíveis.

Quando se estabelecem resultados de decibilidade ou indecibilidade, deve-se, sempre, saber qual é o domínio em questão, porque isso pode afetar a conclusão. Um problema pode ser decidível sobre algum domínio mas não sobre outro. Especificamente uma única instância do problema é sempre decidível, pois a resposta é sempre ou verdadeira ou falsa. No primeiro caso a máquina RAM que sempre responde "verdadeiro" dá a resposta correta, enquanto no segundo caso uma que responde sempre "falso" é apropriada. Isso pode parecer uma resposta falaciosa, mas enfatiza um ponto importante: O fato de não se saber qual a resposta correta não faz nenhuma diferença, o que importa é que existe alguma máquina RAM que dê a resposta correta. Por exemplo, o problema de dado qualquer número natural determinar se sua expansão decimal ocorre ou não na expansão decimal de π, não é decidível, porém qualquer instância deste problema é decidível. Por exemplo, o problema de decidir se a seqüência 1234567890 ocorre ou não na expansão decimal de π é decidível, pois mesmo não sabendo, ou ela ocorre ou não ocorre. Se de fato ocorrer então o programa RAM abaixo do lado esquerdo computaria a resposta correta para esse problema. Senão ocorrer, então o programa do lado direito seria quem computaria a resposta correta.

```
CLR R1
INC R1
CONTINUE
CLR R1
```

```
CONTINUE
```

7.2 O problema da parada para máquinas RAM

Dentre os problemas indecidíveis, o mais conhecido é o **problema da parada**, para as máquinas RAM. Estabelecido de modo simples, ele possui o seguinte enunciado: *dado um programa RAM P e valores $\vec{x} \in \mathbb{N}^m$, $P(\vec{x}) \downarrow ?$.* O domínio desse problema é o conjunto de todos os programas RAM e $\mathbb{N}^m$, isto é $\mathbb{RAM} \times \mathbb{N}^m$. Assim, se está procurando um programa RAM H que, em dado o número de Gödel e de um programa RAM arbitrário P e uma, também arbitrária, entrada $\vec{x}$, $H(e, \vec{x}) \downarrow 1$ ou $H(e, \vec{x}) \downarrow 0$.

A resposta para esse problema não pode ser encontrada simplesmente através da simulação da ação de P sobre $\vec{x}$ (efetuada num programa RAM universal) pois não existe limite no comprimento da computação. Se P entra num laço infinito, então não importa quanto tempo se espere, não se poderá, jamais, estar seguro que P, realmente, está num laço. Pode acontecer, simplesmente, o caso de ser uma computação muito longa* O que se precisa é um algoritmo que possa determinar a resposta correta para qualquer P e $\vec{x}$, ao efetuar uma análise da descrição do programa (código de Gödel) e da entrada. Mas, como se verá, esse algoritmo não existe. Para discussões posteriores, é conveniente ter uma idéia precisa do que se quer dizer com o problema da parada. Por isso, se dará a seguinte definição.

Definição 7.1 *Suponha que k_P é a codificação de Gödel de um programa RAM, P, e seja $\vec{x}$ qualquer elemento de $\mathbb{N}^m$. Uma* **solução para o problema da parada** *é um programa RAM H, onde para qualquer k_P e $\vec{x}$, fornece a computação:*

$$H(k_P, \vec{x}) \downarrow 1 \text{ se } P(\vec{x}) \downarrow$$

e

$$H(k_P, \vec{x}) \downarrow 0 \text{ se } P(\vec{x}) \uparrow$$

Note que o problema da parada só faz sentido se $m \geq 1$, pois caso contrário ($m = 0$) somente o programa que sempre fica em laço infinito (e o qual pode ser computacionalmente detectado) não pára o resto sempre irá parar. Assim, só se considerará daqui para frente $m \geq 1$.

Teorema 7.2 *Não existe qualquer programa RAM, H, que se comporta como as exigências da definição 7.1. O problema da parada é portanto indecidível.*

Prova: Assuma que existe um algoritmo, e conseqüentemente um programa RAM H, que resolve o problema da parada. A entrada para H será a descrição (codificada de alguma forma) de P, k_P, assim como a entrada $\vec{x}$. A exigência é, então, que dado qualquer $(k_P, \vec{x})$, o programa RAM H parará com um sim ($R1 = 1$) ou um não ($R1 = 0$) conforme a definição 7.1. H é da forma:

*Por exemplo, o problema de se achar os fatores primos de um número grande, por exemplo de 200 dígitos levaria bilhões de anos, usando um computador que efetue uma instrução a cada microsegundo.

```
Instrução 1 de H
  :
Instrução s de H
CONTINUE
```

Seja o programa RAM H' descrito a seguir:

```
Instrução 1 de H
    :
Instrução s de H
Nl  R1 JMP Nfb
    JMP Nla
Nf  CONTINUE
```

Comparando H e H' observa-se que, na situação onde H retorna 1, o programa modificado H' entra num laço infinito. Formalmente, a ação de H' é descrita por

$$H'(k_P, \vec{x}) \downarrow 0 \text{ se } P(\vec{x}) \uparrow$$

e

$$H'(k_P, \vec{x}) \uparrow \text{ se } P(\vec{x}) \downarrow$$

Defina agora o programa RAM $\widehat{H}$. Este novo programa toma a entrada k_P, a copia para o registrador $R2$, e então se comporta exatamente com H'. Formalmente, $\widehat{H}$ é o programa descrito a seguir:

```
R2 <- R1
Instrução 1 de H
    :
Instrução s de H
Nl  R1 JMP Nfb
    JMP Nla
Nf  CONTINUE
```

Assim, a ação de $\widehat{H}$ é tal que $\widehat{H}(k_P, \vec{x}') = H'(k_P, k_P, \vec{x}')$, onde $\vec{x}' = (x_2, \ldots, x_m)$ quando $\vec{x} = (x_1, x_2, \ldots, x_m)$. Assim,

$$\widehat{H}(k_P, \vec{x}') \uparrow \text{ se } P(k_P, \vec{x}') \downarrow \quad \text{e} \quad \widehat{H}(k_P, \vec{x}') \downarrow 0 \text{ se } P(k_P, \vec{x}') \uparrow$$

Agora $\widehat{H}$ é um programa RAM que tem uma descrição (codificação) $\widehat{k}$. Esse valor além de ser a descrição de $\widehat{H}$ pode, também, ser usado como entrada. O que aconteceria se $\widehat{H}$ fosse aplicado a $\widehat{k}$? Assim, identificando P com $\widehat{H}$, se obteria:

$$\widehat{H}(\widehat{k}, \vec{x}') \uparrow \text{ se } \widehat{H}(\widehat{k}, \vec{x}') \downarrow \quad \text{e} \quad \widehat{H}(k_P, \vec{x}') \downarrow 0 \text{ se } \widehat{H}(\widehat{k}, \vec{x}') \uparrow$$

o qual é um absurdo. Essa contradição resultou da suposição de que uma máquina RAM H existe e portanto a decibilidade do problema da parada, deve ser falsa.

É importante ter em mente o que o teorema 7.2 diz. Ele não proíbe a solução algorítmica do problema da parada para casos específicos. Freqüentemente pode-se, via uma análise de P e $\vec{x}$, dizer se a máquina RAM parará ou não. O que o teorema diz é que isso não pode ser feito sempre, ou seja não existe um algoritmo que retorne a decisão correta para todo P e $\vec{x}$.

7.3 Redução de um problema indecidível ao problema da parada

Diz-se que um problema A é **reduzido** a um problema B, se a decibilidade de A acarreta a decibilidade de B. A idéia é expressar a solução do problema B em termos da solução do problema A e de problemas conhecidos como decidíveis. Assim, se A for decidível, então poderia-se concluir que B também é decidível. Inversamente, se é sabido que B é indecidível então necessariamente A é indecidível.

Exemplo 7.3 O problema da parada para registradores zerados *Seja P um programa RAM qualquer. Existe um procedimento de decisão que mostre que P ao final da computação parará ou não quando todos os registradores forem inicializados com 0? Suponha que este problema seja decidível, então existe um programa RAM Z tal que $Z(k_P, 0, \ldots, 0) = 1$ se $P(0, \ldots, 0) \downarrow$ e $Z(k_P, 0, \ldots, 0) = 1$ caso contrário. Defina o seguinte programa RAM P':*

```
R1 <- x1
   :
Rm <- xm
P
```

Claramente, $Z(k_{P'}, 0, \ldots, 0) \downarrow$ se e somente se $H(k_P, x_1, \ldots, x_m) \downarrow$. Logo, se Z for decidível então H também será.

Exemplo 7.4 O problema de zerar um registrador *Seja P um programa RAM qualquer e $i \in \mathbb{N}$ tal que $1 \leq i \leq n$. Decidir se P parará ou não com 0 no registrador Ri é um problema indecidível. Suponha que este problema seja decidível, então existe um programa RAM I tal que $I(k_P, \vec{x}) = 1$ se $P(\vec{x}) = (y_1, \ldots, y_{i-1}, 0, y_{i+1}, \ldots, y_n)$ e $I(k_P, \vec{x}) = 0$ caso contrário. Defina $\widehat{P}$ como sendo o seguinte programa RAM:*

```
P
CLR Ri
CLR R1
INC R1
CONTINUE
```

Este programa pára (com $R1 = 1$ e $Ri = 0$) se P pára (com $Ri = 0$ ou não), e não pára caso contrário. Assim, $I(k_{\widehat{P}}, \vec{x}) = 1$ se e somente se $H(k_{\widehat{P}}, \vec{x}) \downarrow$. Logo, se I for decidível então H também será.

A construção dos argumentos desses dois exemplos ilustra uma abordagem comum para estabelecer resultados de indecibilidade.

Uma solução para um problema de decisão é uma função cuja imagem é $\{0, 1\}$, isto é, uma função característica de um determinado conjunto. Para ver se funções mais gerais são computáveis a técnica de redução ao problema da parada é, também, adequada. Devido à tese de Church-Turing, esperasse que funções encontradas em circunstâncias práticas sejam computáveis. Assim, para achar exemplos de funções não computáveis deve-se ir um pouco além. Muitos exemplos de funções não computáveis estão associados à tentativa de prever o comportamento de um programa RAM.

Exemplo 7.5 *Considere a função $f(n)$ cujo valor é o número máximo de comandos que são executados por programas RAM de n instruções que páram quando inicializados todos seus registros em zero. Esta função, como não poderia deixar de ser, não é computável.*

Antes de provar a afirmativa, mostra-se que $f(n)$ é definida para todo n. Observe que existe um número infinito de máquinas RAM com n instruções. De todas essas máquinas, existem algumas que sempre param. Por exemplo, aquelas que não têm instruções JMP. Por outro lado, algumas dessas máquinas não param quando iniciam com todos seus registros zerados, mas essas não entram na definição de f. Toda máquina que pára executará um certo número de comandos. Desses tomasse o maior para fornecer $f(n)$.

Tome qualquer programa RAM, P, e m um inteiro positivo. Tudo o que se tem a fazer é simular P, via um programa RAM universal, contar os comando executados e terminar quando este número exceder m. Este programa RAM universal modificado será denotado por $\widehat{P_U}$. Assuma, agora, que $f(n)$ é computável por algum programa RAM F. Pode-se, então, juntar P' e F. Primeiro obtém-se via F, o valor de $f(\mid P \mid)$, onde $\mid P \mid$ é a quantidade de instruções de P. O valor obtido junto com o número de Gödel do programa P é dado como entrada para $\widehat{P_U}$ quem retornará 1 se o programa P quando inicializado com seus registradores em zero pára antes de executar $f(\mid P \mid)$ instruções ou retornará 0 caso contrário. é, então, usado como m para construir P', conforme já esboçado. Se P quando inicializada com todos seus registradores zerados executa mais do que $f(\mid P \mid)$ movimentos, então, devido à definição de f, se pode concluir que P nunca pára. Portanto, teria-se uma solução para o problema da parada para registradores zerados do exemplo 7.3. A impossibilidade da conclusão leva a aceitar que f não é computável.

7.4 Semi-decibilidade e divergência

Na verdade problemas, como o problema da parada são chamados problemas **semi-decidíveis**, pois existe um algoritmo (programa RAM) que consegue detectar quando

o programa RAM pára, embora nem sempre consiga detectar quando não pára. Ou seja resolve parcialmente o problema da parada. Formalmente um problema de decisão para um conjunto A é semi-decidível se existe um programa RAM P_A tal que $P_A(\vec{x}) \downarrow 1$ se $\vec{x} \in A$. Analogamente, um problema de decisão para um conjunto A é **co-semi-decidível** se existe um programa RAM P_A tal que $P_A(\vec{x}) \downarrow 1$ se $\vec{x} \notin A$. Seja A um conjunto e $\overline{A}$ seu complemento. Assim, claramente P_A é semi-decidível se e somente se $P_{\overline{A}}$ é co-semi-decidível. O problema $P_{\overline{A}}$ é chamado de **complemento do problema** P_A e algumas vezes é denotado por $\overline{P_A}$. Dualmente, P_A é co-semi-decidível se e somente se $P_{\overline{A}}$ é semi-decidível. Note ainda que P_A é semi e co-semi decível se e somente se P_A é decidível. Como corolário tem-se que P_A é decidível se e somente se, P_A e $P_{\overline{A}}$ são semi-decidíveis. Note também que se P_A é semi-decidível e $P_{\overline{A}}$ é co-semi-decidível não implica que P_A seja decidível.

Assim, todo problema decidível é semi-decidível mas a reversa nem sempre é verdade, por exemplo o problema da parada é semi-decidível mas não é decidível. Analogamente, todo problema decidível é co-semi-decidível mas a reversa nem sempre é verdade, por exemplo o complemento de problema da parada, também conhecido como **problema da divergência**, é co-semi-decidível mas não é decidível nem semi-decidível. Ou seja existem problemas que são semi-indecidíveis (ou co-semi-indecíveis) que não são decidiveis (são indecidíveis) mas nem todo problema indecidível é semi ou co-semi decidível, ou seja existem problemas que são **completamente indecidíveis**, por exemplo o problema de "dado dois programas RAM determinar se um pára e o outro não pára quando todos seus registradores forem inicializados em zero", ou seja computar a função

$$f(n,m) = \begin{cases} 1 & \text{, se } P_n(0,\ldots,0) \downarrow \text{ e } P_m(0,\ldots,0) \uparrow \\ 1 & \text{, se } P_n(0,\ldots,0) \uparrow \text{ e } P_m(0,\ldots,0) \downarrow \\ 0 & \text{, caso contrário} \end{cases}$$

onde P_n é a n-ésima máquina RAM na enumeração de Gödel.

7.5 Exercícios

1. Mostre que os seguintes problemas são indecidíveis:

   (a) Dado um programa RAM P arbitrário e $k \geq 1$ fixo, decidir se a função $f : \mathbb{N}^k \longrightarrow \mathbb{N}^k$ computada por P é uma função contante ou não.

   (b) Dado um programa RAM P arbitrário e $k \geq 1$ fixo, decidir se a função $f : \mathbb{N}^k \longrightarrow \mathbb{N}^k$ computada por P é a função identidade ou não.

   (c) Dado um programa RAM P arbitrário e $k \geq 1$ fixo, decidir se a função $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ computada por ela satisfaz $f(\vec{x}) \geq k$ para todo $\vec{x} \in \mathbb{N}^k$.

   (d) Dado um programa RAM P arbitrário e $k \geq 1$ fixo, decidir se a função $f : \mathbb{N}^k \longrightarrow \mathbb{N}^k$ computada por ela tem uma imagem finita ou infinita.

2. Dos problemas descritos no item anterior, indicar quais deles são semi-decidíveis, quais são co-semi-decidíveis e quais são completamente indecíveis.

3. Seja F o conjunto dos números de Fermat, onde n é um número de Fermat se existem números naturais x, y, z maiores que 0 tais que $x^n + y^n = z^n$. Determinar se $P_{\overline{F}}$ é decidível, semi-decidível, co-semi-decidível ou completamente indecidível.

7.6 Considerações finais

Os problemas indecidíveis mostrados neste capitulo são aparentemente artificiais. Mas, é só aparentemente, pois a indecidibilidade do problema da parada tem como conseqüência a impossibilidade de se determinar computacionalmente (construir um programa) se um programa qualquer ficará em laço infinito para alguma entrada. Outro problema mais natural que pode ser reduzido ao problema da parada é o de determinar se dois programas RAM (o dois programas em alguma linguagem de programação) computam a mesma função. Existem, também, problemas matemáticos que são indecidíveis, por exemplo o problema do exercício 3. Um outro problema que não é computável é o problema de "dado um número natural n arbitrário determinar se a expansão decimal de π possui n 7's (setes) consecutivos".

Capítulo 8

Assuntos não abordados

8.1 Paralelismo

Um programa RAM é uma seqüência de instruções que modela algoritmos seqüênciais e portanto computadores convencionais mono-processador. De fato, para muitos autores, esses programas são considerados a formalização mais elegante da arquitetura von Neumann [32]. Esses modelos teóricos de computação proporcionam um ambiente propício para desenvolvimento de softwares, sem a necessidade de que haja preocupações com detalhes de implementação ou com restrições físicas, dando ao software resultante uma portabilidade e uma performance previsível.

Para modelar algoritmos paralelos e computadores multi-processadores, diversos modelos teóricos foram desenvolvidos [17]. Entre eles destacam-se: máquinas de vetores (*Vector Machines*) [22], máquinas de Turing alternadas (*Alternating Turing Machines*) [6], máquinas apontadoras paralelas (*Parallel Pointer Machines*) [7] e máquinas paralelas de acesso aleatório (*Parallel Random Access Machines* – PRAM).

As PRAM foram introduzidas por Fortune e Wyllie em [10] com o objetivo de modelar algoritmos paralelos e computadores multi-processadores sem custo de sincronização ou *overhead* de acesso à memória. Esse modelo consiste de uma coleção de processadores RAM que são executados em paralelo e se comunicam via uma memoria compartilhada que é ilimitada [13].

A máquina PRAM pode ser usada para obter limites teóricos de rendimento (*performance*), escabilidade e programabilidade de computadores paralelos. Assim, modelos PRAM são computadores paralelos ideais sem custo de sincronização de acesso à memória. Na realidade, não existem arquiteturas paralelas de computadores concretos baseados nas PRAM, porém as PRAM tem-se mostrado um veículo extremamente útil no estudo da estrutura lógica da computação paralela em um contexto diferente da comunicação paralela [17]. Algoritmos desenvolvidos para ou-

tros modelos mais realísticos são geralmente baseados em algoritmos originalmente projetados para a PRAM.

Implementações da arquitetura PRAM devem especificar como ocorrem as operações de leitura e escrita concorrente na memória. Quatro opções de acesso são usualmente considerados:

ER (Exclusive-Read): permite que no máximo um processador leia de qualquer posição da memória em cada ciclo,

EW (Exclusive-Write): Permite que no máximo um processador escreva em uma posição de memória,

CR (Concurrent-Read): Permite que múltiplos processadores escrevam em uma posição de memória por ciclo de escrita, e

CW (Concurrent-Write): Permite que escritas simultâneas possam ser feitas na mesma posição de memória.

A combinação delas, resulta nas arquiteturas PRAM-EREW, PRAM-CREW, PRAM-ER-CW e PRAM-CRCW.

Obviamente, pela tese de Church-Turing, o poder computacional da PRAM é exatamente o mesmo que o da RAM, ou seja só computa funções recursivas parciais. Isto pode ser levado para o âmbito de computadores paralelos concretos: tudo o que um computador paralelo pode fazer, uma RAM também pode, a diferença entre computadores paralelos e seqüênciais reais está na capacidade de armazenamento (que em uma RAM, por ser teórica, é ilimitada) e a velocidade de computação.

8.2 Computabilidade sobre conjuntos não contáveis

As abordagens clássicas para teoria da computabilidade, como as vistas até agora, tratam com problemas discretos (por exemplo, sobre os números naturais, números inteiros, strings sobre um alfabeto finito, ou sobre grafos, etc.). No entanto, campos da matemática pura e aplicada tratam com problemas envolvendo números reais, números complexos, superficies, etc. Isto acontece, por exemplo, em análise numérica, sistemas dinâmicos, geometria computacional e teoria da otimização. Assim, uma abordagem computacional para problemas contínuos é desejável, ou ainda necessária, para tratar formalmente com computações analógicas e em computações científicas em geral.

A computabilidade sobre conjuntos contáveis é obtida a partir do desenvolvimento de uma teoria da computabilidade sobre os números naturais de tal modo que nos outros conjuntos discretos (contáveis) a computabilidade se reduz à computabilidade no conjunto dos números naturais. Como o representante natural dos conjuntos contínuos são os números reais, é de se esperar que eles tenham um papel

numa teoria da computabilidade para conjuntos com a cardinalidade do contínuo semelhante à dos números naturais na teoria da computabilidade para conjuntos contáveis.

Na literatura, existem diferentes abordagens para a computabilidade sobre os números reais, mas, uma importante diferença entre estas abordagens, está na maneira como são representados os números reais [12], dentre as quais destacam-se os intervalos encaixantes, expansão r-ádica, seqüências de Cauchy, números reais preguiçosos, etc. Num dos primeiros artigos sobre teoria dos domínios, D.Scott [24] sugeriu que o cpo consistindo de intervalos da reta real poderia ser usado para estudar computabilidade sobre os números reais. Previamente Martin-Löf [20] construiu um espaço similar de aproximações. Em ambos os casos a reta real é mergulhada num espaço de aproximações onde a noção de computabilidade pode ser definida de uma maneira natural. Muitos resultados de computabilidade sobre os números reais, podem ser obtidos neste contexto. Nesta abordagem, uma aproximação da saída com precisão arbitrária é computada a partir de uma aproximação razoável da entrada [5]. Esta abordagem parece natural, uma vez que seja qual for o modelo computacional nunca se terá como manipular um dado infinito em sua totalidade, assim se recorre às aproximações do valor, como por exemplo usando pontos flutuantes dinâmicos. Do ponto de vista teórico, pode-se dizer que uma função $f : \mathbb{R} \longrightarrow \mathbb{R}$ é computável num determinado modelo, se para cada número real x e seqüência de números racionais $q_1, q_2, \ldots$ convergindo para x, tem-se uma máquina M, no modelo, tal que $M(q_1), M(q_2), \ldots$, onde $M(q_i)$ é a saída da máquina M para a entrada q_i, é uma seqüência de números racionais que convergente para $f(x)$.

Uma outra linha de pesquisa para computabilidade real foi desenvolvida por Blum, Shub e Smale [3]. Nesta abordagem, um número real é visto como uma entidade acabada e as funções computáveis são geradas a partir de uma classe de funções básicas (numa maneira similar às funções parciais recursivas). Um outro modelo nesta linha altera o tipo de dado dos registradores das máquinas RAM de naturais para permitirem números reais [33].

Mesmo sabendo que cada uma destas propostas tem seus méritos, nenhuma tem sido aceita pela maioria dos matemáticos e cientistas da computação.

8.3 Oráculos

O uso de oráculos em teoria da computação tem por objetivo modelar agentes externos ao computador. Por exemplo, o acoplamento de um sensor que possa ser acessado e manipulado por programas através do acréscimo de comandos "novos" da linguagem.

Incorporar oráculos em modelos computacionais então é simplesmente adicionar um novo comando ao modelo, de forma análoga como se usam as macros, só que

aqui não se exige que esse comando seja traduzido em termos de outros comandos mais primitivos do modelo (como nas macros). Ou seja pode-se adicionar como oráculo uma função não computável. Claramente, se o oráculo é computável (uma macro) então as funções computadas pelo modelo mais o oráculo serão exatamente as mesmas que são computadas pelo modelo sem o oráculo, já se for adicionada uma função não computável como oráculo, necessariamente a quantidade de funções computadas irá aumentar.

O estudo das funções que são computáveis está bem consolidado, porém, para entender melhor este universo é necessário conhecer o que não é computável. Dentre o conjunto de todas as funções de $\mathbb{N}$ em $\mathbb{N}$, que é incontável, tem-se que só há uma quantidade contável de funções computáveis. Felizmente, a maioria das funções que se conhecem (e portanto se usam) são computáveis. Um exemplo de uma função não computável é aquela que resolve o "problema da parada", denotada por H. Como visto no capítulo anterior, diversos outros problemas podem ser reduzidos ao problema da parada, isto é, se eles forem computáveis implicariam na computabilidade do problema da parada, porém nem todo problema não computável pode ser reduzido ao problema da parada. Assim, se a função H for considerada como um oráculo, todas as funções computáveis e aquelas que resolvem problemas que se reduzem ao problema da parada seriam agora computáveis neste modelo estendido. Só, que novamente surgiria um novo problema da parada para esta nova classe (denotado por H'). Agora, H' pode ser considerado com um oráculo, obtendo uma classe maior de problemas, e assim por diante. Ou seja teriam-se infinitas classes de problemas ou graus de redutibilidade.

Bibliografia

[1] S. Abramsky and A. Jung. *Handbook of Logic in Computer Science*, volume 3, chapter Domain Theory, pages 1–168. Oxford University Press, 1994.

[2] R. E. Biraben. Tese de church: Algumas questões históricos-conceituais. In *Coleção CLE*, volume 20. Centro de lógica, epistemologia e história da ciências — Unicamp, Campinas - SP, 1996.

[3] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over real number: Np-completness, recursive functions and universal machines. *Bull. of the Amer. Math. Soc.*, 21:1–46, 1989.

[4] W. S. Brainerd and L.H. Landweber. *Theory of Computation*. John Wiley & Sons, New York-USA, 1974.

[5] V. Brattka. Recursive characterization of computable real-valued functions and relations. *Theoretical Computer Science*, 162(1):45–77, August 1996.

[6] A.K. Chandra, D.C. Kosen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, january 1981.

[7] P.W. Cook, S.A.and Dymond. Parallel pointer machines. *Computation Complexity*, 3(1):19–30, 1993.

[8] N.J. Cutland. *Computability: An introduction to recursive function theory*. Cambridge university press, Cambridge-UK, 1997.

[9] R.L. de Carvalho and C.M.G.M. de Oliveira. Modelos de computação e sistemas formais. In 11^a *Escola de Computação*. DCC/IM,COPPE/Sistemas, NCE/UFRJ, Rio de Janeiro, julho 1998.

[10] S. Fortune and J. Wyllie. Parallelism in random access machine. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, CA, may 1978.

[11] R. M. Friedberg. Three theorems on recursive enumeration: I decomposition, ii maximal set, iii enumeration without duplication. *Journal of Symbolic Logic*, 23(3):309–316, 1958.

[12] P. Di Gianantonio. *A Functional Approach to Computability on Real Numbers*. PhD thesis, Universitá di Pisa, Genova-Udine, Italy, march 1993.

[13] R. Greenlaw, H.J. Hoover, and W.L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[14] P. R. Halmos. *Teoria Ingênua dos Conjuntos*. Ciência Moderna, 2001.

[15] A Hodges. *Turing. Um filósofo da natureza*. Grande Filósofos. UNESP, 2001.

[16] N.D. Jones. *Computability and Complexity: From a programming perspective*. Foundations of computing. The MIT Press, London-UK, 1997.

[17] R.M. Karp and V. Ramachandran. *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter Parallel Algorithms for Shared-Memory Machines, pages 869–941. Elsevier, 1990.

[18] A.J. Kfoury, R.N. Moll, and M.A. Arbib. *A programing approach to computability*. Text and monographs in computer science. springer-Verlag, Berlin-De, 1882.

[19] S.C. Kleene. *Introduction to metamathematics*, volume 1 of *Monographs on applied mathematics*. North-hollad, Amsterdam-Holland, 1964.

[20] P. Martin-Löf. *Notes on Contructive Mathematics*. Almqvist & Wiksell, Stockholm, 1970.

[21] P. D. Mosses. *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter Denotational semantics. Elsevier Science Publishers and MIT Press, Amsterdam, 1990.

[22] V.R. Pratt and L.J. Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and System Science*, 12(2):198–221, april 1976.

[23] R. Péter. *Recursive Functions*. Academic Press, New York, 3rd edition, 1967.

[24] D. S. Scott. Outline of a mathematical theory of computation. In *4th Princeton Conference on Inf. Science and Systems*, page 65:106, 1970.

[25] D.S. Scott and C.A. Gunter. *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter Semantic Domains, pages 633–674. Elsevier Science Publishers and MIT Press, Amsterdam, 1990.

[26] D.S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proceedings, 21st Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, 1971. Also, Programming Research Group Technical Monograph PRG-6, Oxford University.

[27] C.H. Smith. *A recursive introduction to the theory of computation*. Springer-verlag, New York-USA, 1994.

[28] J. Z. Sobrinho. Aspectos da tese de church-turing. *Matemática universitária*, (6):1–23, 1987.

[29] V. Stoltenberg-Hansen, I. Lindström, and E.R. Griffer. *Mathematical Theory of Domains*. Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge, 1994.

[30] W. J. D. Thomas. *Logic, Language and Probability*, chapter About some standard arguments for Church's thesis, pages 55–65. D. Reidel, Dorbrecht, 1973.

[31] A. M. Turing. Systems of logic based on ordinals. In *Proceedings of London Mathematical Society*, number 45 in 2, pages 161–288, 1939.

[32] L.G. Valiant. *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter General Purpose Parallel Architectures, pages 943–971. Elsevier, 1990.

[33] K. Weihrauch. *Computable Analysis — an introduction*. Springer Verlag, 1997.

Índice