Rodolfo Cristian Ertola Biraben

QUESTÕES CONCEITUAIS DE COMPUTABILIDADE

Tese de doutorado apresentada ao Departamento de Filosofia do Instituto de Filosofia e Ciências Humanas da Universidade Estadual de Campinas sob a orientação da Profª Drª. Ítala Maria Loffredo D'Ottaviano.

Este exemplar corresponde à redação final da tese defendida e aprovada pela Comissão Julgadora em ..../..../2001

BANCA

Profª.Drª. Ítala Maria Loffredo D'Ottaviano

Prof. Dr. Oswaldo Chateaubriand Filho

Prof. Dr. Edward Hermann Haeusler

Prof. Dr. Frank Thomas Sautter

Prof. Dr. Marcelo Esteban Coniglio

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IFCH - UNICAMP

Er88q

Ertola Biraben, Rodolfo Cristian
Questões conceituais de computabilidade / Rodolfo Cristian Ertola Biraben. - - Campinas, SP : [s. n.], 2001.

Orientador: Ítala Maria Loffredo D'Ottaviano.
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Filosofia e Ciências Humanas.

1. Teoria da recursão. 2. Funções computáveis. 3. Matemática construtiva. I. D'Ottaviano, Ítala M. Loffredo (Ítala Maria Loffredo). II. Universidade Estadual de Campinas. Instituto de Filosofia e Ciências Humanas. III. Título.

AGRADECIMENTOS

Agradeço a

Prof. Dr. Antônio Mário Sette (*in memoriam*)

Prof. Dra. Itala Maria Loffredo D'Ottaviano

Prof. Dr. Oswaldo Chateaubriand Filho

Prof. Dr. Paulo Augusto S. Veloso

Prof. Dr. Xavier Caicedo

Prof. Dr. Edward Hermann Hauesler

Prof. Dr. Luiz Carlos Pereira

Prof. Dr. Barry Hartley Slater

Prof. Dr. Frank Thomas Sautter

Prof. Dr. Carlos Gustavo González

Prof. Dr. Marcelo Esteban Coniglio

Prof. Dr. Walter Alexandre Carnielli

Prof. Dr. Hércules de Araújo Feitosa

Funcionários do CLEHC, em especial, Emerson Luis Francisco

CAPES

Djavan, Dori Caymmi, Edu Lobo, Egberto Gismonti, Elis Regina, Hermeto Pascoal, João Bosco, Milton Nascimento e Tom Jobim

Paraty

Brasil

RESUMO

Nesta Tese examinamos algumas questões relacionadas com o conceito de função computável. Entre as análises destacamos um estudo de uma objeção às definições habituais desse conceito, quanto ao caráter construtivo do mesmo. Desenvolvemos uma teoria de computabilidade onde são distinguidas as noções de auto-referência e automodificação. Além disso, estudamos uma reformulação da Tese de Turing-Church, devida a Yiannis Moschovakis.

ABSTRACT

In this Thesis we examine some problems related to the concept of computable function. In particular, we present an analysis of an objection to the usual definition as regards its constructive content. We also develop computability theory distinguishing self-reference and self-modification and we make an evaluation of a reformulation of Turing-Church Thesis by Yiannis Moschovakis.

SUMÁRIO

INTRODUÇÃO

Dada a notória importância da Informática no mundo atual, tanto em seus aspectos científicos como em seus aspectos sociais, há boas razões para se investigar os fundamentos lógicos e matemáticos desta ciência. Em particular, uma melhor compreensão destes fundamentos parece ser de grande utilidade no ensino da Lógica e da Informática.

É claro que o conceito de computabilidade é considerado um dos conceitos básicos da Informática. Neste contexto, o objetivo desta Tese consiste em analisar algumas questões conceituais relativas à noção de computabilidade.

A seguir faremos uma descrição dos capítulos que compõem esta Tese.

No primeiro capítulo apresentamos uma versão da noção de computabilidade a partir do conceito de máquina de registros, introduzida por **Shepherdson e Sturgis (1963)**, segundo a versão de **Cutland (1992)**. Esta noção vai servir como referência à noção de computabilidade nos diversos capítulos da Tese. Neste mesmo capítulo retomamos a discussão apresentada em **Ertola Biraben (1996)**, da não eleição da noção de definibilidade lambda como conceito básico por parte de Church para definir o conceito de computabilidade. Esta retomada justifica-se pela aparição de novos documentos apresentados por **Sieg (1997)**. Também neste capítulo tratamos a questão do *status* da Tese de Turing-Church.

No segundo capítulo desenvolvemos uma série de questões relativas a uma objeção de Péter ao caráter construtivo da definição de função computável. Em particular, vinculamos esta questão com o princípio de Markov e o princípio de menor número, ambos de caráter não intuicionista.

O conteúdo do Capítulo 3 é original. Nele apresentamos uma definição da noção de computabilidade baseada no fato de que em computadores reais os programas e os dados residem na mesma memória, mas, nos conceitos habituais de computabilidade, os programas residem fora da memória (ou fita, no caso das máquinas de Turing). Este conceito de computabilidade permite distinguir entre computabilidade autoreferencial e automodificante. A partir destes conceitos esboçamos algumas definições para uma teoria dos vírus. É digna de menção uma prova de que a complexidade das computações autoreferenciais e automodificantes não varia em relação à computabilidade padrão.

No quarto capítulo examinamos as propostas de Friedman e Moschovakis para ampliar a noção de computabilidade em relação à Tese de Turing-Church.

No anexo constam os textos originais das traduções citadas nos capítulos, ordenados segundo o aparecimento no corpo da Tese.

Observamos que as questões tratadas nesta Tese referem-se ao conceito clássico de computabilidade, mesmo quando nossa análise considera o construtivismo, no Capítulo 2, ou generalizações como a de Moschovakis, no Capítulo 4.

CAPÍTULO 1

QUESTÕES RELATIVAS À ORIGEM DO CONCEITO (PRECISO) DE COMPUTABILIDADE

Na Seção 1 deste Capítulo apresentamos algumas observações terminológicas, a Tese de Turing-Church e uma definição do conceito de função computável. Na Seção 2 apresentamos uma nova resposta para a questão de por que Church não deu primazia a seu próprio conceito de definibilidade lambda. Na Seção 3 analisamos a questão do *status* da Tese de Turing-Church.

1.1 QUESTÕES BÁSICAS

Usaremos a terminologia de computabilidade, conforme as sugestões de **Soare (1996)**. Em resumo, isto significa que falaremos de *Computabilidade* em lugar de *Teoria da Recursão* ou da *Teoria das Funções Recursivas*. Além disto, em lugar de falar de *funções efetivamente calculáveis*, falaremos de *funções intuitivamente computáveis*. Usaremos o termo *computável*, por exemplo, quando estiver envolvida alguma variante da noção de máquina de Turing ou de máquina de registros. O termo *recursivo* fica reservado para a noção tradicional de recursão primitiva e as ampliações de μ-recursão, recursão geral e outras.

Em **Ertola Biraben (1996)** já tínhamos usado a terminologia conforme as sugestões de Soare, quem, no artigo supra-citado dá muitas e boas razões para sermos cuidadosos na forma de expressão. Em particular, cita Georg Cantor, entretanto sem indicar a fonte:

> "Sou extremamente cuidadoso com a escolha daqueles [isto é, novos termos], pois tomo a posição de que o desenvolvimento e a propagação em não pequena medida depende de uma terminologia afortunada e adequada." **(Cantor, apud Soare, 1996, p. 313)**

É relevante acrescentar que estas escolhas terminológicas coincidem, em grande medida, com a forma de expressão de Gödel e Turing, dois dos fundadores da Computabilidade. No caso de Turing, a expressão *computable* já aparece no título do seu famoso artigo (ver **Turing, 1937** ou **Davis, 1965, p. 130-145**). No caso de Gödel, a palavra *berechenbar* (calculável) já aparece numa nota de 1936 (ver **Gödel, 1986, p. 396-399**).

Esta terminologia também foi usada num texto recente:

> "*A teoria da computabilidade* poderia ser um nome melhor para a teoria da recursão, já que é o estudo matemático do que é computável." **(Philips, 1992, p. 80)**

A seguir, enunciamos a Tese de Turing-Church:

(TC) Uma função parcial f de números naturais é intuitivamente computável se, e somente se, f é computável.

Entretanto, para compreendermos (TC), é necessária uma explicação dos dois conceitos envolvidos, o que faremos no resto desta seção.

Uma função é *intuitivamente computável* se *existe* um algoritmo que proporciona os valores da função para os seus argumentos.

Os homens têm construído e usado algoritmos desde a antigüidade mais remota. Portanto, é corriqueiro que as introduções à computabilidade comecem com uma análise do conceito de *algoritmo*. Porém, a análise passa rapidamente para o conceito de *função computável*, o qual é restrito a funções entre números naturais. Depois, aparece o conceito de *programa*. Então, fica claro que temos três conceitos em jogo: o conceito de algoritmo, de função e de programa. Também é corriqueiro, em computabilidade, encontrar definições de função computável e de programa. Mas, em relação ao conceito de algoritmo, nunca encontramos uma definição até

lermos alguns artigos do lógico grego Yiannis Moschovakis, os quais comentaremos no Capítulo 4.

No século XX, os matemáticos se preocuparam em obter uma *definição* (precisa) do conceito de *função computável* de números naturais. Não falaremos aqui do por quê desta necessidade. Na década de trinta foram alcançadas, quase simultaneamente, *diferentes* definições. Porém, foi rapidamente *provado* (com precisão) que estas definições são *equivalentes*.

Novas definições foram surgindo. Na década de sessenta, as definições propostas eram *bem* diferentes: algumas eram mais intuitivas (no sentido que estavam "mais próximas" do conceito de computabilidade intuitiva). Outras eram "mais simples" e outras estavam "mais próximas" da maneira de agir dos computadores reais, que foram se desenvolvendo paralelamente com os conceitos teóricos.

Aqui apresentaremos *uma* destas definições. Porém, não escolheremos o conceito mais intuitivo, que segundo nossa opinião é o conceito que usa as máquinas de Turing. Apresentaremos o conceito de função computável que achamos mais simples e que é "mais próximo" da maneira de agir de um computador real. Além do caráter *simples*, pensamos que a nossa escolha pode ser útil para quem deseja dispor de uma noção que não esteja longe da computação técnica e, ao mesmo tempo, permita-lhe liberar-se das linguagens de máquina ou linguagens *assembler* habituais. O leitor que deseja mais detalhes, ou exemplos ou exercícios, pode consultar os livros de **Cutland (1992)** e de **Davis, Sigal & Weyuker (1994)**.

1.2 A NOÇÃO DE FUNÇÃO COMPUTÁVEL

Nesta Seção apresentamos a definição de função computável utilizando as instruções usadas por **Cutland (1992)**. A seguir, introduzimos versões modificadas das definições desse autor.

A letra N denotara o conjunto $\{0, 1, ...\}$ de números naturais.

Definição 1.2.1. Uma *instrução* é uma expressão numa das seguintes formas:

$Z(\overline{n})$
$S(\overline{n})$
$T(\overline{m},\overline{n})$
$J(\overline{m},\overline{n},\overline{q})$,

onde m, n e q são números naturais maiores ou iguais a 1 e $\overline{n}$ é o numeral correspondente ao número natural n.

Exemplos: $S(\overline{1})$, $Z(\overline{9})$, $J(\overline{1},\overline{1},\overline{3})$, $T(\overline{4},\overline{3})$.

Definição 1.2.2. Um *programa* é uma sucessão finita e não vazia de instruções.

Escrevemos os programas em ordem vertical, como no seguinte exemplo.

Exemplo 1.2.3.

$J(\overline{2},\overline{3},\overline{0})$
$S(\overline{1})$
$S(\overline{3})$
$J(\overline{1},\overline{1},\overline{1})$.

Usamos a notação $P = I_1, \ldots, I_n$ para dizer que o programa P consta das instruções $I_1, \ldots, I_n$, nesta ordem.

Dado $P = I_1, \ldots, I_n$, o número i, tal que $1 \leq i \leq n$, chama-se *índice* da instrução I_i no programa P.

Definição 1.2.4. Uma *memória* é uma sucessão de números naturais (maiores ou iguais a 0).

Escrevemos as memórias em ordem horizontal, como no seguinte exemplo:

(2, 0, 3, 0, 435, 9, 20, 1, 0, 0, 5, ...).

Usamos a notação $m = (m_0, m_1, m_2, \ldots)$ para dizer que a memória m consta dos números $m_0, m_1, m_2, \ldots$, nesta ordem. Dizemos que m_i é o *conteúdo* do *endereço* i da memória m.

Intuitivamente, uma memória corresponde à memória de um computador num dado momento. Diferentemente da memória de um computador real, uma memória tem um número infinito de endereços e um número arbitrariamente grande em cada endereço. No primeiro endereço de uma memória aparece o que se chama usualmente de *contador de programa*.

Comentário 1.2.5. As instruções $Z(\overline{n})$, $S(\overline{n})$, $T(\overline{q}, \overline{n})$ e $J(\overline{r}, \overline{n}, \overline{q})$ significam, respectivamente, as operações de colocar um zero no endereço n da memória, colocar o sucessor no endereço n da memória, transferir o conteúdo do endereço q da memória para o endereço n e pular para a instrução q do programa caso o conteúdo do endereço r for idêntico ao conteúdo do endereço n.

Antes de definirmos o conceito de *computação*, damos um exemplo para melhor compreensão da questão.

Exemplo 1.2.6 Seja P o seguinte programa:

$J(\overline{1},\overline{2},\overline{5})$
$S(\overline{2})$
$S(\overline{3})$
$J(\overline{1},\overline{1},\overline{1})$
$T(\overline{3},\overline{1})$

e seja m a memória:

$(1, 3, 2, 0, \ldots, 0, \ldots)$.

Então a computação de P sobre m é a seguinte sucessão *finita* de memórias:

1	3	2	0	0	...
2	3	2	0	0	...
3	3	3	0	0	...
4	3	3	1	0	...
1	3	3	1	0	...
5	3	3	1	0	...
6	1	3	1	0	...
0	1	3	1	0	...

A computação parou porque não existe a instrução *6*. Com *d* a última memória, em *d(1)* ficou *1*, isto é, a diferença entre m_1 e m_2.

Dado o mesmo programa *P*, consideremos a memória *m'* seguinte :

(1, 2, 3, ..., 0, ...).

Então a computação de *P* sobre *m'* é a seguinte sucessão *infinita* de memórias:

1	*2*	*3*	*0*	*0*	...
2	*2*	*3*	*0*	*0*	...
3	*2*	*4*	*0*	*0*	...
4	*2*	*4*	*1*	*0*	...
.	.	.	.	.	...
.	.	.	.	.	...
.	.	.	.	.	... .

Diremos que a computação não *pára* quando é infinita, como neste caso.

Para definirmos o conceito de computação de forma rigorosa usaremos a seguinte definição.

Definição 1.2.7. Dados um programa P e uma memória m, definimos *a memória resultante de aplicar o programa P sobre a memória m*, que anotamos $mr_P(m)$, assim:

$$[mr_P(m)](i) = \begin{cases} m(i)+1 & se\ I_{m(0)} = Z(\overline{n})\ e\ i = 0; \\ 0 & se\ \ldots \quad i = n; \\ m(i) & se\ldots \quad i \neq 0, n; \\ m(i)+1 & se\ I_{m(0)} = S(\overline{n})\ e\ i = 0; \\ m(n)+1 & se\ \ldots \quad i = n; \\ m(i) & se\ldots \quad i \neq 0, n; \\ m(i)+1 & se\ I_{m(0)} = T(\overline{q},\overline{n}) \quad i = 0; \\ m(q) & se\ \ldots \quad i = n; \\ m(i) & se\ldots \quad i \neq 0, n; \\ q & se\ I_{m(0)} = J(\overline{r},\overline{n},\overline{q}), m(r) = m(n)\ e\ i = 0; \\ m(i) & se\ \ldots \quad i \neq 0; \\ m(i)+1 & se\ldots \quad , m(r) \neq m(n)\ e\ i = 0; \\ m(i) & se \quad i \neq 0. \end{cases}$$

Nesta definição $I_{m(0)}$ denota a $m(0)$-ésima instrução do programa P, caso exista; caso contrário, definimos

$$[mr_P(m)](i) = \begin{cases} 0 & se\ i = 0; \\ m(i) & se \quad i > 0. \end{cases}$$

Para esclarecer a definição, consideremos um exemplo. Seja m a seguinte memória:

$(1, m_1, m_2, m_3, \ldots)$.

Se $I_1 = Z(\overline{n})$, então $mr_P(m)$ é $(2, m_1, m_2, \ldots, m_{n-1}, 0, m_{n+1}, \ldots)$.

Se $I_1 = S(\overline{n})$, então $mr_P(m)$ é $(2, m_1, m_2, \ldots, m_{n-1}, m_n + 1, m_{n+1}, \ldots)$.

Se $I_1 = T(\overline{q}, \overline{n})$, então $mr_P(m)$ é $(2, m_1, m_2, \ldots, m_{n-1}, m_q, m_{n+1}, \ldots)$.

Se $I_1 = J(\overline{r}, \overline{n}, \overline{q})$, então a única alteração é em $m(0)$ caso m_r seja igual a m_n.

Definição 1.2.8. A iteração n-ária da função $mr_P(m)$, que chamamos *a computação de t passos de P sobre o memória m*, e denotamos por $mr_P^t(m)$: $N^n \rightarrow N^n$, é definida por indução:

$$mr_P^0 = id_N N;$$
$$mr_P^{t+1} = mr_P \; o \; mr_P^t.$$

A seqüência de memórias $\{mr_P^t\}_{t \in N}$ será chamada de *computação de P sobre m*. Dados um programa P, uma memória m e $t \geq 0$, a memória $mr_P^t(m)$ é chamada de *memória resultante de aplicar t passos do programa P sobre a memória m.*

Definição 1.2.9. Para $n \geq 1$, $\vec{x} = (x_1, \ldots, x_n) \in N^n$, *a memória para* $\vec{x}$, que denotamos por $m_{\vec{x}}$, é a sucessão

$$m(i) = \begin{cases} 1 & se\ i = 0; \\ x_i & se\ 1 \leq i \leq n; \\ 0 & se\ i > n. \end{cases}$$

Até agora estivemos usando funções totais, que são as funções usuais na matemática. Em computabilidade é natural usar o conceito de função *parcial*, que introduzimos a seguir: se n é um número natural, *uma função parcial de* N^n *em* N é uma função de A em N, para $A \subseteq N^n$.

Notação 1.2.10. Os símbolos $\downarrow$ e $\uparrow$ significam, respectivamente, existência e não existência do valor de uma função para um argumento.

Notação 1.2.11. A notação $f(\vec{x}) \simeq g(\vec{x})$ significa que $f(\vec{x})\uparrow$ e $g(\vec{x})\uparrow$, ou $f(\vec{x})\downarrow$, $g(\vec{x})\downarrow$ e $f(\vec{x}) = g(\vec{x})$.

Definição 1.2.12 Dados $n \geq 1$ e um programa P, fica definida uma função parcial φ_P^n, que chamamos *a função parcial n-computada por P*, do seguinte modo, para todo $\vec{x}$ (onde só por razões de edição usamos x em lugar de $\vec{x}$):

$$\varphi_P^n(x) = \begin{cases} [mr_P^k(m_x)](1) & \text{se } \exists\, t \in N \text{ tal que } [mr_P^t(m_x)](0) = 0; \\ (\text{onde } k \text{ é o menor } t \text{ que satisfaz a condição à direita}) & \\ \uparrow & \text{caso contrário.} \end{cases}$$

Comentário 1.2.13. Preferimos dar a definição de φ_P^n sem abreviações prévias, para que se possa ver melhor os detalhes das questões envolvidas. Observar o uso de um quantificador existencial na condição à direita. Este uso de quantificadores existenciais é usual nos textos de computabilidade (ver, por exemplo, **Monk, 1976, p. 76-77** e **Cutland, 1992**). Observar, também, que na Definição 1.1.12 não se procede supondo *dada* uma função. Além disso, o argumento n corresponde ao número de argumentos da função.

Definição 1.2.14. Dados um programa P, $n \geq 1$ e uma função parcial $f(\vec{x})$, dizemos que P *n-computa* f se $f = \varphi^n{}_P$ [isto é, $f(\vec{x}) \simeq \varphi^n{}_P(\vec{x})$, para todo $\vec{x}$].

Definição 1.2.15. Seja $f(\vec{x})$ uma função parcial e $n \geq 1$. Dizemos que f *é computável* se existe um programa P tal que P n-computa f.

Observar que nesta definição se procede supondo *dada* uma função.

Podemos estender o conceito de computabilidade para *relações*. Quando a relação é unária, dizemos que se trata de uma *propriedade*.

Definição 1.2.16. Seja R uma relação n-ária. Chamamos de *função característica* C_R à função (onde só por razões de edição usamos x em lugar de $\vec{x}$)

$$C_R(x) = \begin{cases} 1 & \text{se } Rx; \\ 0 & \text{caso contrário.} \end{cases}$$

Definição 1.2.17. Seja R uma relação n-ária. Dizemos que R é *computável* se, e somente se, C_R é computável.

A seguir, como exemplos, provaremos que algumas funções são computáveis, construindo programas que cumpram a condição da definição de função computável.

Exemplo 1.2.18. Seja f a função soma de N^2 em N, definida por $f(x, y) = x + y$. Então, f é computável porque o programa apresentado no Exemplo 1. 1. 3 cumpre a condição da definição de função computável.

Exemplo 1.2.19. Seja g a função de N^2 em N definida por

$$g(x, y) = \begin{cases} x - y & se\ y \leq x; \\ \uparrow & se\ x < y. \end{cases}$$

Então, g é computável porque o programa do Exemplo 1.1.6 cumpre a condição da definição de função computável.

Exemplo 1.2.20. Seja h a função de N em N definida por $h(x) = 2x$. Para provarmos que h é computável, temos que construir um programa que cumpra a condição da definição de função computável. Dada uma memória

$$x \quad 0 \quad 0 \quad \ldots \quad ,$$

um tal programa teria que efetuar, por exemplo, a computação

$$\begin{array}{cccc} x & 0 & 0 & 0 \\ x & 1 & 2 & 0 \\ . & . & . & . \\ . & . & . & . \end{array}$$

.	.	.	.
x	x	$2x$	0
$2x$	x	$2x$	0.

O controle do programa estará dado por uma instrução da forma $J\ (1,\ 2,\ q)$, com q como número da última instrução. De acordo com isto, temos o seguinte programa:

$J(\overline{1},\overline{2},\overline{6})$
$S(\overline{2})$
$S(\overline{3})$
$S(\overline{3})$
$J(\overline{1},\overline{1},\overline{1})$
$Z(\overline{1})$
$J(\overline{1},\overline{3},\overline{10})$
$S(\overline{1})$
$J(\overline{1},\overline{1},\overline{7})$.

Exemplo 1.2.21. Podemos provar facilmente que as seguintes funções são computáveis:

1) $f_1(x) = 0$.

2) $f_2(x) = x + 1$.

3) $f_3(x_1, x_2, x_3) = x_2$.

4) $f_4(x) = 3x$.

5) $f_5(x) = \begin{cases} 0 & se\ x = 0; \\ 1 & se\ x \neq 0. \end{cases}$

6) $f_6(x) = \begin{cases} 1 & se\ x = 0; \\ 0 & se\ x \neq 0. \end{cases}$

7) $f_7(x) = \begin{cases} x - 1 & se\ x > 0; \\ \uparrow & se\ x = 0. \end{cases}$

8) $f_8(x) = \begin{cases} x-1 & se\ x > 0; \\ 0 & se\ x = 0. \end{cases}$

9) $f_9(x,y) = \begin{cases} 1 & se\ x \leq y; \\ \uparrow & se\ y < x. \end{cases}$

10) $f_{10}(x,y) = \begin{cases} 1 & se\ x \leq y; \\ 0 & se\ y < x. \end{cases}$

Exemplo 1.2.22. O seguinte programa não pára para nenhuma memória:

$J(\overline{1},\overline{1},\overline{1})$.

Definido o conceito de função computável e explicado o conceito de computabilidade intuitiva, fica claro o sentido da Tese de Turing-Church. Observamos apenas que, originalmente, Church formulou a sua tese fazendo referência somente às funções totais, que são um caso particular das funções parciais, e usando os conceitos de recursão ou definibilidade-λ, que são equivalentes ao conceito de função computável aqui definido, por resultados de **Shepherdson e Sturgis (1963)**.

1.3 POR QUE CHURCH NÃO DEU PRIMAZIA A SEU PRÓPRIO CONCEITO DE DEFINIBILIDADE-λ PARA ENUNCIAR A SUA TESE?

Examinamos esta questão em **Ertola Biraben (1996)**. Dada a existência de nova evidência apresentada por **Sieg (1997)**, que sugere novas conclusões, acreditamos conveniente reestudá-la.

A nova informação surge da correspondência entre Alonzo Church (1902-1995) e Paul Bernays (1888-1977), entre dezembro de 1934 e agosto de 1937. Na primeira carta de Church a Bernays, de 23 de janeiro de 1935, são mencionados dois resultados: 1) as funções recursivas primitivas são λ-definíveis e 2) as funções recursivas gerais também são λ-definíveis. Church diz que o primeiro resultado é de

Stephen Cole Kleene (1909-1994) e o segundo de John Barkley Rosser (1907-1989). Nessa carta fala-se de outras questões, mas em lugar nenhum se diz algo sobre a proposição inversa, isto é, que as funções λ-definíveis são recursivas gerais. Também não consta nada sobre a equivalência entre ambos os conceitos. Esta nova evidência consta em **Sieg (1997, p. 161)**.

Posteriormente, em 19 de abril de 1935, Church enunciou o que depois passou a ser conhecido como Tese de Church, usando o conceito de recursão de Gödel, em lugar de seu próprio conceito de definibilidade-λ. Considere-se a seguinte passagem do *abstract* de Church:

> "Afirma-se que a noção de função efetivamente calculável de inteiros positivos deveria ser identificada com a noção de função recursiva, já que outras definições plausíveis de calculabilidade efetiva acabam fornecendo noções que são ou equivalentes ou mais débeis que a de recursividade." **(Church, 1935, p. 333)**

Este *abstract* foi recebido para publicação no dia 22 de março de 1935.

Dos fatos que i) na carta citada não se fala da prova de que as funções λ-definíveis são recursivas, ii) no *abstract* de Church se menciona a noção de função recursiva mas não a noção de função λ-definível e iii) no referido *abstract* se fala de noções mais débeis, *concluímos* que em 22 de março de 1935 ainda não tinha sido provado que as funções λ-definíveis são recursivas, sendo a noção de definibilidade-λ uma dessas noções supostamente mais débeis.

A equivalência entre recursão geral e definibilidade-λ foi demonstrada por **Kleene (1936b)** via a noção de μ-recursão. A equivalência entre μ-recursão e recursão geral é provada em **Kleene (1936a)**. *Abstracts* destes dois artigos de Kleene foram recebidos em 1 de julho de 1935.

Concluímos que a prova de que as funções λ-definíveis são recursivas (e, portanto, da equivalência entre ambos os conceitos) foi completada entre 22 de março e 1 de julho de 1935.

Concluímos também que a análise histórica de **Davis (1982, p. 12)** está errada ou é enganosa e que a seguinte passagem também está errada:

> "Durante o ano 1934, Church e Kleene demonstraram que as funções λ-definíveis são recursivas. ". **(Ertola Biraben, 1996, p. 52)**

O interessante é que, em relação às análises de Davis e Ertola Biraben, a *ordem* das descobertas foi *invertida*. *Primeiro*, provou-se que as funções recursivas são λ-definíveis e, *em segundo lugar*, a proposição recíproca. Pensamos que esta ordem é mais razoável que a anterior, pois acreditamos ser mais fácil provar que as funções recursivas são λ-definíveis do que o enunciado recíproco. Para o primeiro caso, basta encontrarmos as expressões que definam as funções inicias (zero, sucessor e projeção) e então encontrarmos as expressões para as operações de composição, recursão e minimalização, dadas as expressões para as funções sobre as quais estas operações são definidas. Para o segundo, é preciso fazermos a complicada gödelização do formalismo do cálculo λ.

Resta darmos uma resposta à questão desta seção. A resposta parece ser a de que a comunidade matemática, incluindo Church, Kleene e Rosser, achou mais persuasiva (no sentido de mais próxima do conceito de computabilidade intuitiva) a noção de recursão geral do que a noção de definibilidade-λ. Textos de Kleene e Rosser sugerem isto:

> "A noção resultante [de recursão geral] é de especial interesse, dado que a noção intuitiva de função construtiva ou efetivamente calculável de números naturais pode identificar-se com ela satisfatoriamente." **(Kleene, 1936c, p. 544)**

> "A computabilidade de Turing é intrinsecamente persuasiva no sentido que as idéias nela incorporadas sustentam diretamente a tese de que as funções abarcadas são todas as funções para as quais existem algoritmos; a definibilidade-λ não é intrinsecamente persuasiva (a tese que a usa estava sustentada não pelo mesmo conceito e sim pelos resultados estabelecidos com relação a ele) e a recursão geral é escassamente persuasiva (o seu autor, Gödel, não estava, na época, de nenhum modo persuadido.)" **(Kleene, 1981b, p. 49)**

> "Church, Kleene e eu pensávamos que a recursividade geral parecia incorporar a idéia de calculabilidade efetiva e, portanto, queríamos mostrar que era equivalente à definibilidade-λ." **(Rosser, 1984, p. 345)**.

Estas citações sugerem que Church, Kleene e Rosser não renunciaram à primazia do conceito de definibilidade-λ simplesmente por reações de indiferença a este conceito por parte da comunidade matemática, como indica o seguinte comentário:

> "Eu mesmo, quem sabe influenciado injustificadamente por recepções frias das audiências de 1933-35 à investigações sobre a definibilidade-λ, optei, depois da aparição da recursividade geral, por colocar meu trabalho nesse formato." **(Kleene, 1981a, p. 62)**

Em resumo, em primeiro lugar, Church procurou o conceito preciso "mais próximo" do conceito de computabilidade, para definir as funções computáveis. Em segundo lugar, pensou que o conceito de definibilidade-λ era "menos próximo" do conceito de computabilidade intuitiva que a noção de recursão geral.

1.4 O STATUS DA TESE DE TURING-CHURCH

Nesta seção, o nosso objetivo é analisar as seguintes questões:

1) O ponto de vista tradicional em relação à (TC);
2) As razões da existência de uma tese em computabilidade.

1. Chamamos de *ponto de vista tradicional* a qualquer posição que inclua as seguintes três afirmações:

(a) (TC) é um bicondicional que envolve *dois conceitos*, o conceito de função de números naturais intuitivamente computável e o conceito de função de números naturais computável;
(b) O conceito de função intuitivamente computável é *intuitivo* e o conceito de função computável é *preciso*;

(c) (TC) não tem nem pode ter prova.

A seguir faremos comentários sobre cada um destes pontos, levando em consideração as opiniões de lógicos (um deles contrário à (TC)) e de filósofos da matemática.

Ninguém parece ter criticado (a). Porém, às vezes, em lugar de simplesmente tomar a (TC) como um bicondicional, diz-se que se trata de uma *identificação* do conceito de computabilidade intuitiva com o conceito de computabilidade. É o caso do mesmo Church:

> "Afirma-se que a noção de função efetivamente calculável de inteiros positivos deveria ser *identificada* com a noção de função recursiva, [...]". (grifo nosso) **(Church, 1935, p. 333)**

Kalmár, por sua vez, usa a expressão "identidade":

> "[TC] enuncia a *identidade* de duas noções [...]" (grifo nosso) **(Kalmár, 1959, p. 72)**

A filósofa da matemática Folina também usa a expressão "identidade":

> "A *identidade* conhecida como Tese de Church [...]" (grifo nosso) **(Folina, 1998, p. 302)**

Parece-nos que pode resultar confuso falar de identidades ou identificações se se deseja que a análise sobre o status de (TC) tenha sentido. Portanto, consideramos que apenas se deva pensar em um bicondicional.

Em relação a (b) podemos dizer que a distinção já aparece em Church (que usou a expressão "formal" em lugar de "preciso") e Kleene:

> "[...] a seleção de uma definição *formal* que corresponda a uma noção *intuitiva*." (grifo nosso) **(Church, 1936, p. 356** e **Davis, 1965, p. 100)**

> "... a nossa noção original de calculabilidade efetiva de uma função é um pouca vaga e *intuitiva* [...]". (grifo nosso) **(Kleene, 1952, p. 317)**

Turing tinha uma opinião similar:

> "Todos os argumentos que podem ser dados estão destinados a ser, fundamentalmente, apelos à intuição, e por esta razão, são um pouco insatifatórios matematicamente **(Turing, 1937, p. 249** e **Davis, 1965, p. 135)**

Em relação à citação de Church, em **Ertola Biraben (1996, p. 40)** já comentamos que teria sido mais apropriado usar a expressão "conceito" em lugar de "definição".

A distinção apresentada em (b) também pode ser atribuída a *lógicos* como Kalmár, que usava a expressão "pré-matemático" em lugar de "intuitivo":

> "Existem conceitos pré-matemáticos que devem permanecer pré-matemáticos, pois não podem permitir qualquer restrição imposta por uma definição matemática precisa. Estou convencido que o conceito de calculabilidade efetiva está entre esses conceitos [...]" **(Kalmár, 1959, p. 79)**.

Também se pode atribuir a filósofos da matemática, como Shapiro, o uso da expressão "pré-formal" em lugar de "intuitivo":

> "Muitas classes de funções da teoria dos números têm sido delimitadas por definições formais: as funções aritméticas, as funções de Fibonacci, as funções recursivas, etc. O termo computável, porém, foi aplicado a funções *pré-formalmente* – quer dizer, antes de qualquer definição formal". (grifo nosso, **Shapiro, 1981, p. 353**)

É também o caso de um lógico contemporâneo:

> "A clássica Tese de Church identifica a noção intuitiva de *função computável* sobre os inteiros com a precisa noção matemática de *função recursiva*. É um erro grave

> entendê-la como simplesmente uma definição de computabilidade." (grifo do autor) **(Moschovakis, 1984, p. 354)**

Preferimos a expressão "intuitivo" a "pré-matemático", porque acreditamos que o conceito de computabilidade intuitiva pertence à matemática. Preferimos a expressão "preciso" à "formal" porque aceitamos a existência de conceitos precisos mesmo *antes* do surgimento do conceito de sistema formal em **Frege (1879)**.

Até aqui só temos variações terminológicas. Porém, existe um lógico que *parece* ser contrário a (b), usando o conceito de recursão em lugar do conceito de função computável:

> "Os conceitos e suposições que dão fundamento à noção de função recursiva parcial são, de uma maneira essencial, não menos vagos e imprecisos que a noção de função efetivamente computável; os primeiros são simplesmente mais conhecidos e formam parte de uma teoria respeitável com conexões com outras partes da lógica e da matemática." **(Mendelson, 1990, p. 232)**

Podemos observar que Mendelson compara a noção de computabilidade intuitiva com *os conceitos que dão fundamento* à noção de função computável. A conclusão de Mendelson parece ser a de que o conceito de função computável é mais preciso só na aparência.

Não podemos concordar com Mendelson por uma razão muito simples. Pensamos que as definições de Church, Turing e outros foram contribuições importantes à matemática. E essa contribuição tem tudo a ver com a precisão de um conceito. O mesmo acontece em outros ramos da matemática. Na matemática existem conceitos *intuitivos* e conceitos *precisos*; o conceito de computabilidade intuitiva é do primeiro tipo e o conceito de computabilidade é do segundo tipo.

Folina efetua uma segunda interpretação do argumento de Mendelson, a qual não parece corresponder a este autor, mas que parece interessante (ver **Folina, 1998, p. 309-311**). A idéia é que, *com o tempo*, o conceito de computabilidade intuitiva poder-se-ia tornar mais preciso. Concordamos com Folina em que isto não resolveria nada, pois a nova definição de computabilidade teria problemas similares aos provocados por (TC).

Folina coloca esta questão como concebível. Mas, parece-nos que isto já ocorreu. Trata-se, por exemplo, da teoria do Moschovakis. Este autor apresenta uma definição mais geral de função computável, com a sua correspondente tese, que examinaremos no Capítulo 4 desta Tese. (ver **Moschovakis, 1984, p. 354**)

Em relação a (c), comecemos por considerar que, em 1936, Church, depois de apresentar o que ele chamou de "definição da noção de calculabilidade efetiva", fez o seguinte comentário:

> "Esta definição *justifica-se* pelas seguintes considerações, na medida em que se pode obter justificação positiva para a seleção de uma definição formal para que corresponda a uma noção intuitiva". (grifo nosso) (**Church, 1936, p. 356** e **Davis, 1965, p. 100**)

Quer dizer que a computabilidade começou com *dois* elementos:

1) Uma definição do conceito de função recursiva (ou lambda definível) (ambos os conceitos equivalentes à noção de computabilidade);
2) (TC).

As *justificativas* que Church menciona são justificativas para (TC). Claro, pois uma definição, como simples abreviação, não precisa de justificativas, as quais já foram analisadas em **Ertola Biraben (1996, p. 64-68)**. O ponto de vista *standard* da comunidade é que estão *erradas*, por serem *circulares*, como foi sugerido por **Thomas (1973)**, **Soare (1996, p. 289-291)** e **Sieg (1996, p. 165)**.

De qualquer modo, fica claro que Church não pensava que pudesse existir uma prova de (TC) no sentido estrito (a frase "na medida em que se pode obter justificação positiva" indica isso). Kleene tinha a mesma opinião:

> "Como a nossa noção original de calculabilidade efetiva de uma função é uma noção pouco intuitiva, (TC) não pode ser provada [...]" (**Kleene, 1952, p. 317**)

Citamos também o lógico Kalmár, que em alguns pontos discordava de Church e Kleene, mas que nesta questão tinha a mesma opinião:

> "A Tese de Church não é um teorema matemático que possa ser provado ou refutado no sentido matemático exato, porque estabelece a identidade de duas noções das quais só uma está definida matematicamente e a outra é usada pelos matemáticos sem definição exata." (**Kalmár, 1959, p. 72**)

Portanto, o ponto de vista tradicional é que (TC) não tem nem pode ter prova.

Porém, alguns autores pretendem que existem provas de (TC). É o caso de **Mendelson (1990)**, **Shapiro (1993, p. 74-75)**, **Gandy (1988, p. 80)** e **Sieg (1997)**.

Mendelson pretende ter uma prova da metade menos significativa de (TC): se uma função f é computável, então f é intuitivamente computável. Usando o conceito de μ-recursão em lugar de computável, ele argumenta que é claro que as funções iniciais são intuitivamente computáveis e que também é claro que as regras de composição, recursão primitiva e operação-μ conservam a computabilidade intuitiva:

> "Em cada caso, podemos descrever procedimentos para computar as funções correspondentes" (**Mendelson, 1990, p. 233**).

Concordamos com **Folina (1998, p. 311-319)** em que o argumento de Mendelson não é uma prova, no sentido habitual na comunidade matemática.

No caso de Shapiro, Gandy e Sieg, pode-se consultar o artigo de **Folina (1998)**.

Em relação a (c) resumimos, então, que embora (TC) não tenha prova, os argumentos de Mendelson apresentados ou quaisquer outros semelhantes podem ser considerados como *argumentação matemática intuitiva*, a qual não deveria ser desvalorizada por sua condição não precisa. Concordamos com Kreisel em criticar:

> "[...] o ponto de vista que nos diz que não devemos levar a sério a questão tradicional de se uma caracterização axiomática de um conceito informal é *correta* porque (neste ponto de vista) os conceitos informais são por sua natureza demasiado imprecisos para que isto tenha uma resposta." (**Kreisel, 1967, p. 176**)

Este autor **(Kreisel, 1967, p. 152-155)** pensava que, em alguns casos, as argumentações podem ser consideradas *provas*, por exemplo, no caso das argumentações para as *teses* seguintes:

(T1) $\forall\alpha(\ Val\ \alpha \leftrightarrow V\alpha\ )$

(T2) $\forall\alpha(\ Val\ \alpha \leftrightarrow D\alpha\ ),$

onde o domínio de quantificação é o conjunto de fórmulas de primeira ordem, $Val\ \alpha$ significa que α é intuitivamente válida, $V\alpha$ que α é válida em todas as estruturas conjuntistas (validez semântica) e $D\alpha$ que α é formalmente derivável por meio de um conjunto fixado de regras de inferência (por exemplo, dedução natural). Desta forma, *(T1)* é a tese que diz que uma fórmula de primeira ordem é intuitivamente válida se e somente se ela é válida em todas as estruturas conjuntistas.

As provas de *(T1)* e *(T2)* fornecidas por Kreisel são as seguintes. Em primeiro lugar, temos que

(F1) $\forall\alpha(\ Val\ \alpha \rightarrow V\alpha\ )$

é aceita pela comunidade lógica, mesmo que *Val* seja um conceito informal, pois refere-se a *todas* as estruturas, e *V* simplesmente refere-se a todas as estruturas *conjuntistas*, qualquer que seja a teoria de conjuntos envolvida.

Em segundo lugar, temos que

(F2) $\forall\alpha(D\alpha \rightarrow Val\ \alpha\ )$

também é aceita pela comunidade lógica, mesmo que *Val* seja um conceito informal, porque, por exemplo, ninguém jamais argumentou contra a correção das regras de dedução natural. Por meio de *(F1)* e *(F2)*, usando completude (i. e. $\forall\alpha(V\alpha \rightarrow D\alpha)$), é trivial provar *(T1)* e *(T2)*.

Em **Mendelson (1990, p. 231)** aparece uma variante da prova apresentada por Kreisel.

É interessante acrescentar que a partir de *(F1)* e *(F2)* pode-se concluir o Teorema de Correção, i.e. $\forall\alpha\,(\,D\alpha \rightarrow V\alpha\,)$, conseqüência que não contém nenhum conceito intuitivo.

Entretanto, pensamos que a argumentação de Kreisel também não constitui uma prova, mas tem caráter matemático intuitivo.

Além disso, na argumentação apresentada Kreisel supõe que ser válido implica ser verdadeiro em toda estrutura, o que parece questionável:

> "O problema é que Kreisel simplesmente identifica, sem argumento, a noção intuitiva com a noção de teoria de modelos de verdadeira em toda estrutura." **(Etchemendy, 1999, p. 147)**.

2. Analisamos, a seguir, a nossa segunda questão: por que existe ou existiu uma tese em computabilidade?

Em relação a isto, em primeiro lugar, é natural se perguntar se existem teses análogas em outros ramos da lógica ou da matemática. O ponto de vista tradicional em relação a esta questão é que sim: vários autores têm chamado a atenção sobre a existência de "teses" em várias teorias matemáticas (**Shapiro, 1981, p. 356-359**, **Mendelson, 1990, p. 230-232** e **Soare, 1996, p. 296-298**). Um exemplo clássico disto seria a tese que diz que uma função é intuitivamente contínua se, e somente se, ela é contínua no sentido da aritmetização da análise, i.e. com a notação ε-δ. Na lógica, temos o uso da tese que diz que uma fórmula de primeira ordem é intuitivamente válida se, e somente se, ela é válida no sentido conjuntista de ser verdadeira em toda estrutura. Outro exemplo interessante seria o conceito de longitude de uma curva, o qual, segundo Descartes, seria impossível de definir:

> "Até aproximadamente 1650 ninguém acreditava que a longitude de uma curva pudesse igualar exatamente a longitude de uma linha. De fato, no segundo livro de *La Geometrie*, Descartes diz que a relação entre as linhas curvas e as retas não é nem poderá jamais ser conhecida." **(Kline, 1972, p. 354)**.

Isto significa que em outros ramos da lógica e da matemática tem existido teses semelhantes à de Turing-Church. Porém, pensamos que no caso da Tese de

Turing-Church a situação é bem mais explícita. A seguir, apresentamos várias razões para tentar explicar esta diferença.

Em primeiro lugar, observamos que durante muitos anos existiu *ceticismo* no ambiente matemático em relação à possibilidade de se definir o conceito de função computável. Uma causa desse ceticismo foi o poder que se atribuía ao método de *diagonalização*, pensando que sempre se poderia diagonalizar *fora* do conjunto de funções presumivelmente computáveis. Por exemplo, antes de Ackermann provar o contrário, usando um argumento com o conceito de diagonalização, pensava-se que o conceito de computabilidade poderia ser definido por meio do conceito de função recursiva primitiva (ver também **Rogers, 1967, p. 10-11**).

Para se ver o tipo de dificuldades que apresenta o método de diagonalização, pode ser útil considerarmos a seguinte função (que aparece em **Cutland, 1992, p. 78**):

$$f(n) = \begin{cases} \phi_n(n)+1 & \textit{se } \phi_n(n) \textit{ está definido;} \\ 0 & \textit{caso contrário,} \end{cases}$$

onde $\phi_0, \phi_1, \ldots, \phi_n, \ldots$ é a enumeração das funções unárias parciais computáveis. É claro que f difere de todas as funções desta enumeração. Portanto, f é uma função unária total não computável.

Observe-se que a definição de f *não* contradiz os seguintes comentários de Gödel (em 1946) e Kleene:

> "Para o conceito de computabilidade, porém, apesar de que é meramente uma classe especial de demonstrabilidade ou definabilidade, a situação é diferente. Por um tipo de milagre não é necessário distinguir ordens, e o *procedimento* diagonal não conduz fora da noção definida." (grifo nosso) **(Gödel, 1990, p. 150)**

> "Quando Church propôs esta tese, tentei refutá-la diagonalizando a classe de funções λ-definíveis. Porém, compreendendo rapidamente que a diagonalização não pode ser feita *efetivamente*, tornei-me defensor da tese de um dia para outro." (grifo nosso) **(Kleene, 1981, p. 59)**

A função f foi definida por uma diagonalização *não* construtiva, pois a propriedade $\phi_n(n)\downarrow$ não é computável (ver, por exemplo, **Bridges, 1994, p. 48** ou **Cutland, 1992, p. 101**); mas, Gödel e Kleene referem-se às diagonalizações *construtivas*.

Portanto, não é verdade que não seja possível diagonalizar fora das funções parciais computáveis; o que não é possível, é fazer isso *construtivamente* (caso contrário, teríamos um contra-exemplo à Tese de Church).

Uma segunda causa do ceticismo é o uso dos quantificadores existenciais *não construtivos* na definição do conceito de função computável. Esta questão é discutida no Capítulo 2 desta tese. Neste lugar, basta colocar o comentário de Péter, ainda em 1981(!):

> "Concordo com a convicção de Kalmár no sentido que a computabilidade efetiva é uma dessas noções cuja definição nunca pode ser considerada completa no curso do desenvolvimento da matemática." **(Peter, 1981, p. 142)**

Uma terceira causa do ceticismo é a questão de que o conceito de computabilidade não seria um conceito matemático, mas só um conceito psíquico ou físico. Post, por exemplo, falava do "poder matematizante do Homo Sapiens". Em relação às máquinas de Turing fala-se de "estados mentais": Turing falava de um *computor*, no sentido de uma *pessoa*, e do que ele *podia* calcular com um número finito de estados mentais. Shapiro fez uma observação similar:

> "A computabilidade é uma propriedade relacionada à habilidades humanas ou dispositivos mecânicos, ambos os quais são pelo menos, *prima facie*, não matemáticos" **(Shapiro, 1981, p. 353)**

Apesar de aqui não analisarmos mais o assunto, o leitor interessado pode consultar as idéias de Gödel no terceiro volume dos seus artigos póstumos sobre fundamentos da Matemática **(Gödel, 1995)**.

Em segundo lugar, observamos o fato de Church ter usado *justificativas* as quais foram consideradas *incorretas*, tanto por filósofos **(Thomas, 1973)** como por lógicos **(Soare, 1996, p. 289-291)** e historiadores **(Sieg, 1997)**. Nesta questão, é

curioso que na bibliografia consultada não tenhamos encontrado críticas aos argumentos de Church por parte dos lógicos *anteriores* à citação de Soare, já mencionada, de 1996. Por outro lado, Thomas publicou a sua crítica já em 1973. É como se a comunidade lógica tivesse preferido manter *silêncio* nas questões filosóficas básicas da disciplina, conforme a expressão *maioria silenciosa* da qual falava Kreisel, parodiando Nixon.

Em terceiro lugar, observamos o fato de que as primeiras definições propostas (com os conceitos de recursão geral ou definibilidade-λ por parte de Church, ou a Formulação 1 de Post) não estavam acompanhadas por uma *análise* do conceito intuitivo de computabilidade. Esta questão foi tratada, por exemplo, em **Ertola Biraben (1996, p. 46-51)**. Post criticou Church e foi o primeiro a usar o termo *hipótese*. Depois da análise de Turing, a dúvida provocada pelo artigo de Post permaneceu no ambiente lógico. As primeiras definições simplesmente "caíram do céu". Só existia uma coincidência extensional entre os conceitos intuitivo e preciso de computabilidade.

Em quarto lugar, observamos a comodidade de *usar* a Tese de Church nas provas em teoria da computabilidade, com a qual os lógicos simplificam as suas apresentações. Este costume já apareceu em **Rogers (1967)** e também foi tratado em **Ertola Biraben (1996, p. 58-60)**.

Em quinto lugar, observamos o uso da Tese de Church como princípio em várias formas de matemática construtiva (ver **Troelstra & van Dalen, 1988, p. 192-195** e esta Tese, p. 47).

CAPÍTULO 2

COMPUTABILIDADE E CONSTRUTIVISMO

Neste capítulo faremos uma análise de uma objeção de Péter à definição de função computável. Na Seção 1 apresentamos a motivação da questão e na Seção 2 a sua origem histórica. Na Seção 3 examinamos a relação entre os conceitos de computabilidade intuitiva e construtividade. Na Seção 4 apresentamos o caso das funções totais. Na Seção 5 apresentamos alguns comentários históricos sobre o conceito de função computável *parcial*. Nas Seções 6 e 7 examinamos o princípio de Markov e a noção de realizabilidade, ambos relacionados com o construtivismo e o intuicionismo. Na Seção 8 apresentamos uma análise de Heyting em relação à questão. Na Seção 9 analisamos o primeiro quantificador existencial presente na definição de função computável. Na Seção 10 fornecemos alguns comentários relativos ao Princípio do Menor Número. Finalmente, na Seção 11, apresentamos as nossas conclusões em relação à objeção de Péter.

2.1 MOTIVAÇÃO DA QUESTÃO

Este capítulo foi motivado pelos três comentários seguintes, que apresentamos em ordem cronológica:

> "Existem alguns finitistas ou construtivistas que poderiam negar que todas as funções recursivas gerais são computáveis, ou inclusive afirmar que a classe de funções recursivas gerais não está bem definida. Porém, falando de funções recursivas parciais

evitamos esta diferença de opinião. Pois certamente não existe dúvida nenhuma de que as rotinas dadas aqui e em qualquer outro lugar computarão o valor de uma função recursiva dada para um dado argumento onde a função está definida, e seguirão computando para sempre se a função não está definida nesse argumento. Sem dúvida, agora pode existir uma diferença de opinião em relação a se uma função recursiva parcial dada é recursiva geral, isto é, definida para todos os argumentos; de fato, a questão se uma função tal está definida para um argumento particular pode ser tão difícil como a conjectura de Fermat. Mas, a não concordância com esta questão, ou com a questão equivalente se a correspondente rotina computacional termina ou não, não afeta a prova completamente finitista de que para argumentos para os quais a função está definida ela computará o seu valor." (**Shepherdson & Sturgis, 1963, p. 217**)

"Diferentemente do mais complexo conceito de procedimento mecânico, que sempre termina, agora se vê claramente que o conceito não qualificado, isto é, que pode ou não terminar, tem o mesmo significado para os intuicionistas e para os clássicos." (**Wang, 1974, p. 84**)

"Olhando para trás não é difícil ver que a dificuldade provém de considerar só funções totais. Foi Kleene (1938), em seu notável artigo, quem notou que todo conjunto de equações pode ser usado para computar uma função *parcial*. Não existe nenhum rastro de *circularidade*, inclusive para um construtivista, na definição de *função recursiva parcial*." (nossa ênfase) (**Davis, 1982, p. 17**)

Estes três autores referem-se a um problema na definição do conceito de função computável, que não existiria no caso das funções parciais. Nenhum dos três autores dá referências bibliográficas, mas as únicas instâncias publicadas de um argumento que corresponda a esta questão, que encontramos na bibliografia, são as seguintes, que também apresentamos em ordem cronológica:

"Mas o objetivo principal da introdução deste conceito [de função recursiva geral] foi justamente a concepção exata do conceito de construtividade. A chamada Tese de Church identifica o conceito de função calculável com este conceito. Aqui não quero

entrar na questão sobre a qual falará Kalmár, se de fato todas as funções calculáveis são recursivas gerais; quero apresentar exatamente a questão oposta: podem as funções recursivas gerais legitimamente serem chamadas de efetivamente calculáveis, isto é, construtivas?" (**Péter, 1959, p. 227-228**)

"Assim, definem-se propriamente dois conceitos de função recursiva geral: um com "existe" concebido classicamente, e outro com "existe" concebido intuicionisticamente. Seria interessante mostrar com um exemplo até que ponto o último conceito é mais restrito, através de uma função recursiva geral clássica mas não intuicionista; mas isto é difícil de esperar, pois nas considerações até esta data ainda não apareceu, em geral, nenhum exemplo de função recursiva geral mas não recursiva especial. Agora, o conceito clássico de função recursiva geral não é construtivo, e o intuicionista contém um círculo vicioso: aqui o "existe" que aparece na *definição* deve ser construtivo –mas se queria definir exatamente com esta *definição* de recursividade geral a construtividade." (grifos nossos) (**Péter, 1959, p. 228**)

"A Sra. Péter explicou na sua conferência neste colóquio que qualquer tentativa de *definir* a noção de teoria construtiva conduz a um círculo vicioso, porque a definição sempre contém um quantificador existencial, que por sua vez deve ser interpretado construtivamente." (grifo nosso) (**Heyting, 1959, p. 70**)

"Se *definimos* a noção de função calculável como significando função recursiva, então a *definição* da última noção requer para a sua interpretação a noção de calculabilidade. Portanto, é circular *definir* recursividade por calculabilidade." (grifo nosso) (**Heyting, 1962, p. 197**).

Arend Heyting (1898-1980) não requer apresentação, mas pode ser necessário lembrar que Rózsa Péter (1905-1977) foi uma especialista húngara em funções recursivas (mas *não* no sentido de computabilidade) reconhecida, por exemplo, por Gödel (ver **Soare, 1996, p. 307-308**). Entre outras coisas, ela introduz o termo "recursão primitiva" em 1934 (**Péter, 1934**).

A objeção de Péter foi apresentada no mesmo colóquio sobre construtivismo em matemática, realizado em Amsterdam em 1957, onde Kalmár (1905-1976) apresentou o seu argumento contrário à (TC) que comentamos em **Ertola Biraben (1996, p. 68-72)**. Desta forma, ampliamos aqui a nossa análise destas questões conceituais colocadas *pelos próprios lógicos*. Esse argumento de Kalmár, por exemplo, ainda é *citado* em alguns textos *recentes* de computabilidade (ver e.g. **van Dalen, 1983, p. 453** e **Bridges, 1994, p. 32**). É claro que como a primeira citação de Péter indica, o que está em jogo aqui é um dos dois condicionais de (TC): o que afirma que toda função computável é intuitivamente computável.

2.2 ORIGEM HISTÓRICA DA QUESTÃO

A questão apresentada na seção anterior já preocupou a um dos iniciadores da teoria da computabilidade. Consideraremos um comentário de Church, em 1936, quando a teoria começava a se desenvolver. Para isso, previamente, será preciso colocar o contexto das considerações de Church.

Church usou uma linguagem com os símbolos *1*, *S*, um número infinito de variáveis numéricas $x_1, x_2, \ldots, x_n, \ldots$ e, para cada natural positivo n, um conjunto infinito $f^1_n, f^2_n, \ldots$ de letras de função.

Os *termos* são definidos da maneira usual. *Numerais* são os termos *1*, *S(1)*, ... e os naturais positivos *correspondem* aos termos *1*, *S(1)*, ... Chamam-se *equações elementares* as equações da forma $t_1 = t_2$, onde t_1 e t_2 são termos.

Dado um conjunto E de equações elementares, definem-se por indução as *equações derivadas* de E: i) as equações elementares de E são equações derivadas de E; ii) Se $t_1 = t_2$ é uma equação derivada de E que contém a variável x, então o resultado de substituir todas as ocorrências de x por um numeral é uma equação derivada de E; iii) se $t_1 = t_2$ é uma equação derivada de E que contém um termo t e se $t = s$ ou $s = t$ é uma equação derivada de E, então o resultado de substituir t por uma ocorrência de s em $t_1 = t_2$ é uma equação derivada de E.

Suponhamos um conjunto finito E de equações elementares tal que 1) nenhuma equação derivada de E tem a forma $t_1 = t_2$, onde t_1 e t_2 são numerais diferentes, 2) as letras de função que ocorrem em E são $f^1{}_n1, f^2{}_n2, \ldots, f^r{}_nr$ e 3) para todo número i entre 1 e r inclusive, e para todo conjunto de numerais $k^i_1, k^i_2, \ldots, k^i_ni$, *existe* um único natural k^i tal que $f^i{}_{ni}(k^i_1, k^i_2, \ldots, k^i_ni) = k^i$ é uma equação derivada de E. Sejam $F^1, F^2, \ldots, F^r$ as funções de naturais positivos tais que $F^i(m^i_1, m^i_2, \ldots, m^i_ni)$ é igual a m^i, onde $m^i_1, m^i_2, \ldots, m^i_ni$ e m^i são os naturais positivos que correspondem aos numerais $k^i_1, k^i_2, \ldots, k^i_ni$ e k^i. Então diz-se que o conjunto de equações E é um conjunto de *equações de recursão* para qualquer das funções F^i. Uma função de naturais positivos chama-se *recursiva* se, e somente se, pode-se dar um conjunto de equações de recursão para ela.

Acreditamos conveniente apresentar a versão original de Church da noção de função recursiva, para que as citações que se seguem possam ser cabalmente entendidas. Depois de apresentar a sua definição, Church argumenta intuitivamente que para toda função recursiva existe um algoritmo:

> "É claro que para toda função recursiva de inteiros positivos existe um algoritmo, através do qual pode-se calcular efetivamente qualquer valor particular referido da função. Pois as equações derivadas do conjunto de equações recursivas E são efetivamente numeráveis, e o algoritmo para o cálculo dos valores particulares de uma função F^i, denotados por uma variavel funcional $f^n{}_i$, consiste em efetuar a enumeração das equações derivadas de E até que a requerida equação particular da forma $f^i{}_{ni}(k^i_1, k^i_2, \ldots, k^i_ni) = k^i$ seja encontrada." (**Church, 1936, p. 351** e **Davis, 1965, p. 95**)

Agora, apresentamos o comentário de Church, por nós já mencionado, que tem a ver com a objeção de Péter:

> "O leitor pode objetar que não se pode sustentar que este algoritmo proporciona um cálculo efetivo do valor particular requerido de F^i, a menos que tenhamos uma prova *construtiva* de que a requerida equação $f^i{}_{ni}(k^i_1, k^i_2, \ldots, k^i_ni) = k^i$ será encontrada em última instância. Mas então isto meramente significa que deveríamos tomar o

> quantificador existencial que aparece na nossa definição de um conjunto de equações recursivas em um sentido construtivo. Deixa-se para o leitor a questão de qual seja o critério de construtividade." (grifo nosso) (**Church, 1936, p. 351** e **Davis, 1965, p. 95**)

Lembramos que as funções recursivas mencionadas por Church são funções totais. Na bibliografia consultada não encontramos nenhum outro comentário de Church sobre esta questão.

2.3 OS CONCEITOS DE COMPUTABILIDADE INTUITIVA E CONSTRUTIVIDADE

A última citação de Church parece indicar que ele *não* estava interessado em definir o conceito de função *construtiva* de números naturais. Porém, o seu discípulo Kleene uma vez escreveu:

> "A noção resultante [de recursão geral] é de especial interesse, pois a noção intuitiva de uma função "*construtiva*" ou "efetivamente calculável" de números naturais pode se identificar com ela muito satisfatoriamente." (grifo nosso) (**Kleene, 1936b, p. 544**)

Este não é um comentário ocasional. Num estudo retrospectivo, referindo-se ao intuicionismo e às funções computáveis, escreveu:

> "Ambas as teorias pretendiam tratar, em diferentes arenas, com processos efetivos ou *construtivos*." (grifo nosso) (**Kleene, 1973, p. 95**)

Além disto, parece natural esperar que os valores de um algoritmo sejam encontrados por um procedimento *construtivo*. Isto é, parece natural considerar que o conceito de função intuitivamente computável é o conceito de função construtiva. Concordamos com van Dalen:

> "Existem enfoques construtivo e não construtivo da noção de recursividade. Parece existir pouca dúvida de que o contexto apropriado para uma teoria da computabilidade (abstrata) é o construtivo." (**van Dalen, 1983, p. 453**)

Esta questão sobre os dois enfoques da noção de recursividade (ou de computabilidade) já foi indicada por Heyting:

> "O hábito salutar de distinguir entre resultados sobre funções recursivas obtidos pela lógica intuicionista e aqueles para os quais a prova precisa de lógica clássica é abandonado em muitos artigos e livros recentes. Lamento isso, porque desse modo é obscurecida a conexão da teoria com a noção de calculabilidade efetiva." (**Heyting, 1962, p. 196**)

Esta diferença entre os conceitos de computabilidade intuitiva (construtiva, segundo Péter) e computabilidade invalidaria (TC), como queria Péter. Por isto, num artigo de 1963, Mendelson sugere que o conceito de computabilidade intuitiva fosse considerado não construtivo. Com efeito, considere-se a seguinte passagem:

> "As duas ocorrências do quantificador existencial "existe" são tomadas no sentido clássico não construtivo." (**Mendelson, 1963, p. 202**)

Desta forma, teríamos agora três conceitos, os conceitos (imprecisos) de função construtiva e de função intuitivamente computável (não construtivo) e o conceito (preciso) de função computável (também não construtivo).

Nossa conclusão é que este novo conceito de função intuitivamente computável não construtivo é simplesmente uma invenção que serve como respaldo intuitivo do conceito (preciso) de função computável, para justificar (TC), obtido a partir do conceito intuitivo e construtivo de função computável.

2. 4 FUNÇÕES TOTAIS

Na Definição 1.2.15. definimos o conceito de função computável *parcial.* Para isso, usamos o conceito de função (parcial) *n*-computada por um programa *P* (Definição 1.2.12). Nesta definição aparece a condição $\exists\, t \in N$ tal que $[mr^t{}_P(m_{\vec{x}})](0) = 0$. É a este quantificador que nos referimos quando falamos do "segundo quantificador existencial". Aquilo que denominamos "primeiro quantificador" é o quantificador existencial da Definição 1.2.15, que se refere à existência de um programa.

Parece-nos bem claro que tanto Church como Péter (nas citações da Seção 1) referem-se ao segundo quantificador. Este quantificador refere-se ao caráter finito de uma computação, caso ela termine.

Para entender bem a questão envolvida na objeção de Péter será conveniente considerarmos também o caso da definição de funções computáveis *totais*, em nossa variante.

Definição 2.4.1 Seja *P* um programa e *n* um número maior ou igual a *1*. Dizemos que *P* é *n-regular* se, e somente se, para todo $\vec{x} \in N^n$, $\exists\, t \in N$ tal que $[mr^t{}_P(m_{\vec{x}})](0) = 0$.

Nesta definição fica clara a situação $\forall\exists$, isto é, a quantificação universal de uma quantificação existencial.

Definição 2.4.2 Seja $f(\vec{x})$ uma função total, com $n \geq 1$. Dizemos que *f é computável* se, e somente se, existe um programa *P n*-regular tal que $f = \varphi^n{}_P$, isto é, $\forall \vec{x}, f(\vec{x}) = \varphi^n{}_P(\vec{x})$.

2.5 ORIGEM DAS FUNÇÕES PARCIAIS

Pelas citações de Shepherdson e Sturgis, Wang e Davis da Seção 1 parece claro que parte da comunidade lógica, com exceção, por exemplo, de **Mendelson (1963)**, aceitava as objeções de Péter à (TC), no caso das funções totais. Mas esses quatro

autores também indicam que a objeção de Péter perderia sentido no caso das funções *parciais*. Nesta seção faremos um histórico da introdução do conceito de função parcial computável.

Em 18 de setembro de 1938 o *Journal of Symbolic Logic* recebeu um artigo de Kleene intitulado "On Notation for Ordinal Numbers", que foi publicado em dezembro de 1938. Nesse artigo aparece pela primeira vez a noção de função recursiva parcial. Kleene "parcializa" a noção de função Herbrand-Gödel computável, quer dizer, a noção que se refere a um sistema de equações:

> "Se omitimos o requisito de que o processo de computação sempre termina, obtemos uma classe mais geral de funções, cada uma das quais está definida sobre um subconjunto (possivelmente vazio ou total) das uplas de números naturais, e *possui a propriedade de efetividade quando definida*. Estas funções chamamos de recursivas parciais." (grifo nosso) (**Kleene, 1938, p. 151**)

Kleene, em nenhuma parte do artigo, diz explicitamente que com esta generalização obtém-se um conceito tal que a objeção construtivista sobre a noção de função recursiva geral (total) desaparece. Também não menciona os problemas dos quantificadores existenciais em nenhuma parte do artigo. Porém, Péter, no artigo já comentado, indica que

> "Kleene pensa que quem não aceita [um existe geral] pode entender "existe" construtivamente" (**Péter, 1959, p. 228**),

mas não dá nenhuma referência. Deste modo, isto simplesmente *sugere* que Péter considerava que Kleene tinha um ponto de vista similar ao de Church, o qual já foi comentado no final da Seção 2. Além do mais, não conhecemos nenhum outro texto de Kleene onde se estabeleça uma relação entre o conceito de recursão parcial e o problema do quantificador existencial.

É interessante acrescentar que, segundo Kleene, Gödel, no outono de 1939, ainda não sabia o que era uma função parcial:

> "Em 1938, generalizei (ou deveria dizer "parcializei"?) a noção de Gödel de função recursiva geral para obter a noção de funções recursivas parciais.
> Pode ser interessante que, em conversação com Gödel no outono de 1939, mencionei "funções recursivas parciais" e ele veio me perguntar imediatamente "O que é uma função recursiva parcial?" Evidentemente não tinha visto meu artigo de 1938.
> Aparentemente, logo ele incorporou a idéia." (**Kleene, 1987, p. 57**).

Efetivamente, em 1974, Wang informou as idéias de Gödel em relação às funções recursivas parciais:

> "Gödel diz que a noção precisa de procedimentos mecânicos é dada mais claramente pelas máquinas de Turing que produzem funções recursivas parciais do que pelas máquinas que produzem funções totais." (**Wang, 1974, p. 84**)

Atualmente, num texto *standard* de teoria de computabilidade trabalha-se, basicamente, com as funções computáveis parciais, as quais também têm a propriedade da enumerabilidade efetiva, o que simplifica muitas provas, como tem sido observado, por exemplo, por **Richman (1983, p. 797)**.

2.6 A NOÇÃO DE REALIZABILIDADE

Kleene tentou encontrar uma conexão entre o conceito de função computável e o intuicionismo. Desta forma surgiu o conceito de realizabilidade em **Kleene (1945)**. Nesta seção faremos um histórico da introdução do conceito de realizabilidade.

Para entender em detalhe as motivações de Kleene, que começou a trabalhar neste assunto já em 1941, pode-se ver **Kleene (1945)** e **Kleene (1949)**. É bom lembrar ainda que a interpretação "dialética" de Gödel também se originou no mesmo ano; ele

deu uma palestra em 15/04/41 em Yale intitulada "In what sense is intuitionistic logic constructive?" (ver **Gödel, 1990, p. 217**).

Uma variante da definição do conceito de realizabilidade é a seguinte (tomada de **Kleene, 1973**), onde $(e)_i$ é o expoente do $i+1$-ésimo primo na decomposição de e em produto de fatores primos:

Cláusula 1: Seja E uma fórmula fechada da aritmética de Heyting (HA) e e um número natural arbitrário.

Se E é atômica, então e *realiza* E se, e somente se, E é verdadeira;

Se E é da forma $A \,\&\, B$, então e *realiza* E se, e somente se, $(e)_0$ realiza A e $(e)_1$ realiza B;

Se E é da forma $A \vee B$, então e *realiza* E se, e somente se, $(e)_0 = 0$ e $(e)_1$ realiza A, ou $(e)_0 \neq 0$ e $(e)_1$ realiza B;

Se E é da forma $A \rightarrow B$, então e *realiza* E se, e somente se, para todo a, se a realiza A, então $\{e\}(a)\downarrow$ e $\{e\}(a)$ realiza B, onde $\{e\}$ é a e-ésima função numa enumeração efetiva das funções parciais;

Se E é da forma $\neg A$, então e *realiza* E se, e somente se, para todo a, a não realiza A;

Se E é da forma $\forall x\, Ax$, então e *realiza* E se, e somente se, para todo x, $\{e\}(x)\downarrow$ e $\{e\}(x)$ realiza $A(x)$;

Se E é da forma $\exists x\, Ax$, então e *realiza* E se, e somente se, $(e)_1$ realiza $A((e)_0)$.

Claúsula 2. Seja $E[y_1,...,y_m]$, contendo livres só $y_1,...,y_m$ $(m \geq 0)$. Então, E *é realizável* se, e somente se, existe uma função recursiva geral φ tal que, para todos $y_1,...,y_m$, $\varphi(y_1,...,y_m)$ realiza $E(y_1,...,y_m)$.

A definição dada é para HA. Em 1947 David Nelson provou que toda fórmula demonstrável em HA é realizável (ver **Nelson, 1947**).

Que o recíproco não vale foi provado por **Rose (1953)**, um estudante de doutorado de Kleene, que acabou seu trabalho em 1952. A questão tinha sido proposta por Kleene em 1941 e Rose começou a investigação em 1947, obtendo um contra-exemplo em 1951, dez anos depois. Rose provou que a fórmula

$$[(\neg\neg D \rightarrow D) \rightarrow \neg\neg D \vee \neg D] \rightarrow \neg\neg D \vee \neg D,$$

onde D é a fórmula $\neg p \vee \neg q$, é realizável, mas não é um teorema do cálculo proposicional intuicionista. Não é trivial provar que esta fórmula é realizável (ver **Rose, 1953, p. 11-12**).

Porém, usando modelos de **Kripke (1965)**, é fácil ver que não é um teorema do cálculo proposicional intuicionista. Comecemos lembrando o seguintes fatos:

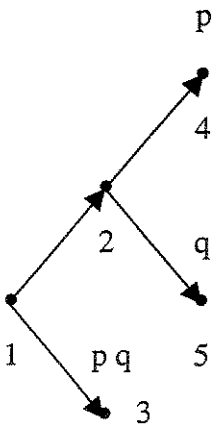
Proposição 2.6.1 Valem as seguintes relações de consequência intuicionista:

$$A \vee \neg A \nvdash_i \neg\neg A \rightarrow A$$
$$A \vee \neg A \nvdash_i \neg A \vee \neg\neg A$$
$$\neg\neg A \nvdash_i A$$
$$\nvdash_i \neg A \vee \neg\neg A$$
$$\neg\neg A \rightarrow A \nvdash_i \neg A \vee \neg\neg A.$$

Vejamos agora o caso da fórmula de Rose.

Proposição 2.6.2 A fórmula de Rose não é um teorema intuicionista.

Prova. Considere o seguinte contra-modelo de Kripke:



A fórmula de Rose não é satisfeita no nó *1*. Para ver isto, pode ser útil a seguinte tabela, onde, debaixo do número da cada nó, aparecem as colunas com a informação necessária, onde *Não F* significa que a fórmula *F* não é forçada no modelo acima:

1	2	3	4	5
Não D	*Não D*	*Não D*	*D*	*D*
Não $\neg D$	*Não* $\neg D$	$\neg D$	*Não* $\neg D$	*Não* $\neg D$
Não $\neg\neg D$	$\neg\neg D$	*Não* $\neg\neg D$	$\neg\neg D$	$\neg\neg D$
Não $\neg\neg D \vee \neg D$	$\neg\neg D \vee \neg D$	$\neg\neg D \vee \neg D$	$\neg\neg D \vee \neg D$	$\neg\neg D \vee \neg D$
Não $\neg\neg D \rightarrow D$	*Não* $\neg\neg D \rightarrow D$	$\neg\neg D \rightarrow D$	$\neg\neg D \rightarrow D$	$\neg\neg D \rightarrow D$

Basta provar que $1 \Vdash (\neg\neg D \rightarrow D) \rightarrow \neg\neg D \vee \neg D$, mas que não $1 \Vdash \neg\neg D \vee \neg D$. É claro que não $1 \Vdash \neg\neg D \vee \neg D$, pois não $1 \Vdash \neg\neg D$, pois $3 \Vdash \neg D$ e não $1 \Vdash \neg D$, pois $4 \Vdash D$.

Vejamos, agora, que para todo $k \geq 1$, se $k \Vdash \neg\neg D \rightarrow D$, então $k \Vdash \neg\neg D \vee \neg D$. Para $k > 1$, é fácil ver que $k \Vdash \neg\neg D \vee \neg D$, pois $2 \Vdash \neg\neg D$, $3 \Vdash \neg D$, $4 \Vdash \neg\neg D$ e $5 \Vdash \neg\neg D$. Resta ver que não $1 \Vdash \neg\neg D \rightarrow D$. Mas $1 \leq 2$ e $2 \Vdash \neg\neg D$, e não $2 \Vdash D$.

É interessante acrescentar que, segundo a prova do próprio Rose, o conectivo $\rightarrow$ aparece de modo essencial no contra-exemplo acima, pois ele mostrou que a interpretação da realizabilidade é completa para fórmulas sem $\rightarrow$ (ver **Rose, 1953, p. 12-14**).

Observe-se também que, na definição da realizabilidade para os conectivos proposicionais, os índices das funções computáveis *só* aparecem no caso do condicional, onde aparece a expressão $\{e\}(a)\downarrow$, que é um enunciado *existencial*.

2.7 O PRINCÍPIO DE MARKOV

Nesta seção apresentamos um princípio que tem a ver com a questão da objeção de Péter: o Princípio de Markov. O matemático russo Andrei A. Markov (1903-1979), que iniciou uma escola de matemática construtivista em Moscou, formulou ainda em 1952 (ver **Troelstra e van Dalen, 1988, p. 27**) o seguinte princípio:

> "Seja P uma propriedade, e suponhamos que existe um algoritmo que decide, para cada número natural n, se n tem ou não a propriedade P. Se a proposição que nenhum número tem a propriedade P leva a uma contradição, então existe um número natural com a propriedade P." (**Markov, 1971, p. 5**)

Formalmente, este princípio costuma se apresentar assim (ver e.g. **Troelstra & van Dalen, 1988, p. 203**):

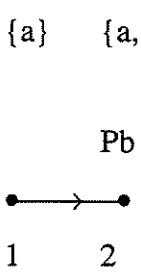
$$(MP) \quad \forall x (A \vee \neg A) \wedge \neg \neg \exists x A \rightarrow \exists x A.$$

Esta questão induz dois comentários. Em primeiro lugar, não conhecemos nenhum lugar da obra de Markov onde este autor tenha apresentado este princípio como um princípio lógico, como Troelstra e van Dalen, mas só como um princípio relativo aos números naturais, como acabamos de ver. Veremos que *(MP)* não é intuicionista. Então, daria lugar a uma lógica intermediária, entre a intuicionista e a lógica clássica.

Em segundo lugar, parece claro que, tanto em sua formulação aritmética como em sua generalização lógica, este princípio não tem sentido no contexto da lógica clássica, pois em tal contexto é trivialmente válido, mas sim, por exemplo, na lógica intuicionista ou na aritmética de Heyting.

Surge naturalmente a questão se (MP) é ou não um esquema intuicionista. Em **Troelstra (1973, p. 93-94)** aparece um argumento sintático de **Kreisel (1958)** que implica que (MP) não é um esquema intuicionista. Posteriormente John Myhill (1923-1987) deu outra prova desse resultado (ver **Myhill, 1963**). Estes argumentos são

anteriores à semântica de Kripke, com a qual pode-se dar o seguinte contra-modelo, que é sumamente simples:



onde {a} e {a, b} são os domínios de interpretação dos nós 1 e 2, respectivamente.

Para vermos que (MP) não é um esquema intuicionista, temos que $1 \Vdash \neg Pa$ e $2 \Vdash \neg Pa$, pois não $1 \Vdash Pa$ e não $2 \Vdash Pa$. Então, temos que $1 \Vdash Pa \vee \neg Pa$ e que $2 \Vdash Pa \vee \neg Pa$. Por outro lado, temos $2 \Vdash Pb \vee \neg Pb$, por ter $2 \Vdash Pb$. Isto significa que $1 \Vdash \forall x(Px \vee \neg Px)$.

Como $2 \Vdash \exists x\, Px$, então, intuicionisticamente, não se cumpre a conjunção de que não $1 \Vdash \exists x\, Px$ e que não $2 \Vdash \exists x\, Px$. Então, por definição de $\Vdash$ para negações, temos que não $1 \Vdash \neg \exists x\, Px$ e que não $2 \Vdash \neg \exists x\, Px$. Então, $1 \Vdash \neg \neg \exists x\, Px$.

Como o conjunto associado a P no nó 1 é vazio, é claro que não $1 \Vdash \exists x\, Px$.

Portanto, no modelo dado, não $1 \Vdash \forall x\, (Px \vee \neg Px) \wedge \neg \neg \exists x\, Px \rightarrow \exists x\, Px$. Pela correção intuicionista, resulta que a fórmula $\forall x(Px \vee \neg Px) \wedge \neg \neg \exists xPx \rightarrow \exists xPx$, um caso particular de (MP), não é um teorema intuicionista. Segue-se que (MP) não é um esquema intuicionista.

O Princípio de Markov não deve ser confundido com a seguinte regra, que é válida em HA:

(MR) Se $\vdash \forall x\, (A \vee \neg A)$ e $\vdash \neg \neg \exists x\, A$, então $\vdash \exists x\, A$.

Para uma análise de *(MP)* e *(MR)* ver **Troelstra & van Dalen, 1988, p. 203-206**.

Parece apropriado advertir que **Mendelson (1979, p. 238)**, quando fala do Princípio de Markov, refere-se a algo diferente, refere-se a uma forma da Tese de Turing-Church, que Markov formulou em 1951:

> "Agora é natural perguntar em que medida o conceito exato de algoritmo normal corresponde ao conceito geral, não inteiramente preciso, de um algoritmo em um alfabeto dado, que se formulou acima. Como uma resposta a esta questão pode-se propor o seguinte princípio de normalizacão de algoritmos: todo algoritmo sobre o alfabeto A é equivalente, em relação a A, a um certo algoritmo normal sobre A."(**Markov, 1960, p. 8**)

Para completar a nossa análise do Princípio de Markov, apresentamos a seguir a questão da validez de *(MP)* na interpretação da realizabilidade. No artigo de Myhill mencionado na Seção anterior, o autor indica que *(MP)* é valido na interpretação da realizabilidade (ver **Myhill, 1963, p. 359**). Em **Moschovakis (1980, p. 173)** aparece o mesmo comentário, acrescentando que a prova deste fato é *clássica*. Como não vimos uma tal prova em nenhuma parte e ela é simples, acreditamos conveniente apresentá-la, para completar o tratamento do Princípio de Markov.

Podemos definir a realizabilidade para a lógica de predicados de forma análoga à efetuada por **Rose (1953)** para a lógica proposicional: uma fórmula F do cálculo de predicados é *realizável* se, e somente se, toda fórmula de HA obtida a partir de F, substituindo fórmulas de HA com n variáveis livres por fórmulas atômicas com letras de predicado n-árias de F, é realizável.

Consideremos, a seguir, uma fórmula arbitrária de HA obtida da maneira indicada a partir de *(MP)*:

(I) $\forall x\,(A(x) \vee \neg A(x)) \wedge \neg\neg \exists x\, A(x) \rightarrow \exists x\, A(x)$.

Dada uma fórmula F qualquer, pelo terceiro excluído, temos que F é realizável ou não. Suponhamos que $\neg\neg \exists x\, A(x)$ seja realizável. Então, pelo seguinte resultado de **Kleene (1945, p. 114)**:

*(i)*Se A não contém variáveis livres e $\neg\neg A$ é realizável, então A é realizável,

temos que $\exists x\, A(x)$ é realizável. **Kleene (1945, p. 114)** também provou o seguinte:

(j) Se B é realizável, então $A \rightarrow B$ é realizável.

Por meio de *(j)* temos, então, que *(I)* é realizável.

Suponhamos que $\neg\neg\exists x\, A(x)$ não seja realizável. Então, pela definição de realizabilidade, para o caso da conjunção, temos que o antecedente de *(I)* não é realizável. Mas **Kleene (1945, p. 114)** também provou o seguinte:

(k) Se A não é realizável e não contém variáveis livres, então $A \rightarrow B$ é realizável.

Como o antecedente de *(I)* não contém variáveis livres, temos que *(I)* também é realizável neste caso. Como ficou claro no começo, esta é uma prova *clássica* da realizabilidade do princípio de Markov.

Finalmente, lembramos que *(MP)* também é válido na interpretação "dialética" de Gödel (ver **Troelstra e van Dalen, 1988, p. 204**).

2.8 A ANÁLISE DE HEYTING

A seguir, examinamos alguns comentários de Heyting. Como vimos na Seção 1, Heyting parece ter tido idéias semelhantes às de Péter. Nos artigos citados nessa seção, Heyting não analisou o assunto em detalhe, mas isto foi feito no artigo "Blick von der intuitionistischen Warte", publicado em 1958. Neste artigo Heyting comenta sobre a interpretação tanto clássica como intuicionista do segundo quantificador, considerando a nota de rodapé de Church já apresentada na Seção 2. A seguir, analisa a questão baseado na definição de recursividade em termos da existência de sistemas de equações, de acordo com as idéias de Herbrand e Gödel. Considera duas definições, fazendo

referência ao tratado de **Kleene (1952)**, mas *sem* indicar onde elas aparecem no mencionado livro. Trata-se de definir quando uma função φ é recursiva geral (**Kleene, 1952, #55, p. 274**) e quando um sistema de equações define uma função recursiva (**Kleene, 1952, # 55, p. 275**). A primeira definição é:

> Uma função $\varphi(\boldsymbol{x})$ é *recursiva* se, e somente se, existe um sistema de equações E tal que $\forall \boldsymbol{x}$ cumpre-se que $E \vdash f(\boldsymbol{x}) = \boldsymbol{y}$, onde f é a letra principal de E e $\boldsymbol{x}$ é uma n-upla de numerais se, e somente se, $\varphi(\boldsymbol{x}) = y$.

A segunda definição é:

> Um sistema de equações E define uma função recursiva de n variáveis se, e somente se, $\forall \boldsymbol{x} \exists! y$ tal que $E \vdash f(\boldsymbol{x}) = \boldsymbol{y}$, onde f é a letra principal de E.

Usando os conceitos do Capítulo *1*, estas definições podem ser traduzidas, respectivamente, da seguinte maneira:

Definição 2.7.1 Uma função $f(\boldsymbol{x})$ é *computável* se, e somente se, existe um programa P tal que $\forall \boldsymbol{x}\, P(\boldsymbol{x}) \downarrow f(\boldsymbol{x})$.

Definição 2.7.2 Um programa P *define uma função computável de n variáveis* se, e somente se, $\forall \boldsymbol{x} \in N^n$ existe um único y tal que $P(\boldsymbol{x}) \downarrow y$.

Heyting considera estas definições como *EA* e *AE* respectivamente, indicando *E* um quantificador existencial e *A* um quantificador universal. Isto significa que ele *não* está considerando o quantificador existencial implícito na notação $\vdash$, que era o quantificador comentado por Church na sua nota de rodapé. Com rigor, pode ser visto na Definição 2.7.1 que ela é da forma $EA(E \wedge P)$.

Heyting considera que ambas as definições supõem o conceito de função calculável. No primeiro caso, porque a função φ só pode estar dada, segundo ele (quer dizer, para um construtivista), se φ é calculável. No segundo caso, porque o valor da função só pode estar dado, para todo x, como valor de uma função calculável (neste segundo caso, Heyting argumenta como Péter) (ver **Heyting, 1958, p. 341**).

Mas Heyting examina a questão também no caso das funções parciais. Ele considera que, neste caso, a situação é diferente no caso da segunda definição, pois a existência e unicidade se transformam em simples unicidade. Heyting considera que a segunda definição de Kleene apresentada se transforma numa definição de tipo A, sem supor o conceito de função calculável.

Heyting não considera o caso das funções parciais para a primeira definição.

O comentário de Heyting, no sentido que as funções só podem estar dadas caso sejam calculáveis, aparece no tratamento atual da construtividade quando se inclui, como axioma agregado à Aritmética de Heyting *(HA)*, alguma formalização de *(TC)*, e. g.:

$$(CT0) \quad \forall n \exists m A(n,m) \rightarrow \exists k \forall n \exists m [A(n, U m) \wedge T knm],$$

onde U e T são os conhecidos predicados primitivos computáveis de Kleene (ver **Troelstra e van Dalen, 1988, p. 193**).

2.9 O PRIMEIRO QUANTIFICADOR EXISTENCIAL

Para entender a questão em detalhe parece necessário considerar os quantificadores existencias presentes na definicão de função computável.

A nossa definicão (Definição 1.2.15) exige a existência de um programa. Este primeiro quantificador existencial aparece em *todas* as variantes da definição de função computável (tanto parcial como total) mais ou menos explicitamente.

Do ponto de vista histórico, pode ser interessante observar que nas versões originais dessa definição o primeiro quantificador aparecia pouco explicitamente. Por exemplo, Kleene colocou a seguinte definição:

> Uma função é primitiva se *pode ser definida* a partir das funções... (grifo nosso) (**Kleene, 1936a, p. 729**)

Quer dizer, em lugar de usar o quantificador existencial usava-se a frase *pode ser definida.*

É fácil encontrar casos de funções que são computáveis (e.g. segundo a Definição 1.2.15) mas que não são construtivas:

$$f(x) = \begin{cases} 0 & \textit{se a Conjectura de Goldbach é verdadeira;} \\ 1 & \textit{caso contrário.} \end{cases}$$

A Conjectura de Goldbach diz que todo número natural maior que *2* é a soma de dois primos. É uma questão ainda não resolvida. Mas supondo o princípio do terceiro excluído, a Conjectura de Goldbach é verdadeira ou não. Se for verdadeira, a função f é a função constante *0*; caso contrário, a função f é a função constante *1*. Então, a função f é a função constante *0* ou a função constante *1*. Logo, como ambas essas funções são computáveis, resulta que a função f é computável, mesmo que ninguém saiba qual é exatamente esta função.

O exemplo dado é comum nos textos de teoria de computabilidade, para mostrar o caráter não necessariamente construtivo das definições de função computável. Porém, em geral, os lógicos clássicos não se preocupam com isto, mesmo que esteja sendo definido um conceito que tem algo a ver com a noção de construtividade. Estes lógicos costumam desenvolver a teoria da computabilidade usando métodos não construtivos: usam-se os princípios habituais da lógica clássica e a teoria dos conjuntos, incluindo o Axioma da Escolha (ver, por exemplo, **Rogers, 1967, p. 10**).

Outro exemplo interessante é dado pelo mesmo autor (**Rogers, 1967, p. 9**):

$$g(x) = \begin{cases} 1 & \text{se a expansão de } \pi \text{ tem uma sucessão consecutiva de pelo menos } x \text{ cincos;} \\ 0 & \text{caso contrário.} \end{cases}$$

A função g, mesmo que ninguém possa dizer que função é exatamente, usando o princípio do terceiro excluído, é a função constante *0*, ou a função constante *1*, ou existe um número i tal que $g(x) = 1$, para $x \leq i$ e $g(x) = 0$, para $x > i$. Como qualquer destas funções é computável, a função g também é computável.

É interessante acrescentar que nos artigos clássicos da década de 30 não aparecem exemplos semelhantes aos que acabamos de apresentar e que os lógicos dessa época não pareciam estar muito preocupados com esta questão, em relação ao *primeiro* quantificador existencial.

Também é interessante observar que van Dalen deu a seguinte resposta à objeção de Péter (os exemplos que ele menciona são similares aos que acabamos de apresentar):

> "Vistos construtivamente, os exemplos acima são defeituosos, pois o princípio do terceiro excluído é construtivamente falso (cfr. o capítulo sobre lógica intuicionista [Capítulo III.5]). O significado construtivo de "existe uma função recursiva parcial φ é: podemos computar efetivamente um índice e. Rósza Péter usou a leitura construtiva da teoria da recursão como um argumento para a circularidade da noção de recursividade, quando baseada na Tese de Church, Péter [1959]. A circularidade é, porém, enganosa. As funções recursivas não se usam para computar números isolados, mas para dar saídas para entradas dadas. A computação de objetos discretos isolados precede a manipulação de funções recursivas, é uma das atividades básicas do construtivismo. Então, a computação de um índice de uma função recursiva parcial não precisa ela mesma de funções recursivas." (**van Dalen, 1983, p. 453-454**)

Acreditamos que na Seção 1 ficou claro na citação de Péter que ela não estava se referindo ao que chamamos "o primeiro quantificador", nesta seção. Porém, como

acabamos de ver, van Dalen parece responder à objeção como se esta estivesse dirigida ao primeiro quantificador.

Como comentário final desta seção, temos a dizer que uma razão pela qual a questão do primeiro quantificador não é tão importante é que provavelmente a teoria da computabilidade possa se desenvolver sem necessidade da definição de função computável, usando só o conceito de função n-computada por um programa P, conforme a Definição 1.2.12.

2.10 O PRINCÍPIO DO NÚMERO MÍNIMO

Péter também apresentou uma versão de sua objeção para as funções computáveis colocadas na forma normal de Kleene (ver **Péter, 1959, p. 228**). Esta forma normal tem a ver com o operador μ, que é usado para definir as funções μ-recursivas, conceito equivalente ao conceito de função computável. No caso das funções *totais*, a definição do operador μ é a seguinte.

Definição 2.10.2 Seja $f(x, y)$ uma função tal que $\forall x \exists y\, f(x, y) = 0$. Então, $\mu y[f(x, y)=0]$ = *o mínimo y tal que* $f(x, y) = 0$.

Observamos que, como o Princípio do Número Mínimo:

$$(LNP)\ \exists x\, A(x) \rightarrow \exists x\, [A(x) \wedge \forall y{<}x \neg A(y)]$$

não vale no intuicionismo (ver **Troelstra & van Dalen, 1988, p. 129**), a definição acima não pode ser introduzida no intuicionismo.

De fato, no lugar indicado, Troelstra e van Dalen provam que se *(LNP)* for acrescentado à *(HA)*, então demonstra-se $A \vee \neg A$, para toda fórmula A, isto é, a lógica se torna clássica e, consequentemente, *(HA)* se transforma na aritmética de Peano. Nesta prova usa-se a regra *MTP* (Modus Tollendo Ponens). Um problema aparentemente em

aberto é o que acontece com *(LNP)* se a lógica utilizada for, simplesmente, a lógica minimal de Johansson, que é igual à intuicionista tirando-se *MTP*.

É natural se perguntar sobre o relacionamento entre *(MP)* e *(LNP)*. A questão parece ter sentido no contexto da aritmética de Heyting. Em primeiro lugar, é fácil ver, em *(HA)*, que *(LNP)* implica *(MP)*, pois sabemos que *(LNP)* implica a lógica clássica, e no contexto da lógica clássica, *(MP)* vale trivialmente.

Em relação à questão inversa, isto é, se *(MP)* implica ou não *(LNP)*, a resposta natural parece ser que não, pois no caso afirmativo não teria sentido estudar o comportamento de *(MP)* no contexto de *(HA)*, pois a lógica se tornaria clássica. Porém, não conhecemos nenhuma prova desta questão.

Em conclusão, no contexto de *(HA)*, *(MP)* parece ser mais débil que *(LNP)*.

2.11 CONSIDERAÇÕES FINAIS SOBRE A OBJEÇÃO DE PÉTER

Parece-nos evidente que a computação de um programa é um processo construtivo, quer dizer, quando um programa terminou, atingiu seu valor construtivamente. Mais especificamente, um programa atinge seu valor e *depois* pára. Por outro lado, em geral, quando tentamos definir a função computada por um programa, tendemos a começar pensando em termos da dicotomia pára - não pára. Somente depois disto definimos o valor da função para aqueles casos em que o programa pára. Disso resulta que, embora o processo de computação seja construtivo, a definição da função computada pelo programa não é construtiva, pelo menos em sentido intuicionista. Parece-nos que na discussão sobre a objeção de Péter há uma tendência para não diferenciar estes dois aspectos.

Não descartamos a possibilidade de dar uma definição intuicionista do conceito de função computável parcial. Em todo caso, tal caracterização deveria ser apresentada explicitamente. A esse respeito, pode-se consultar o terceiro capítulo de **Troelstra e van Dalen (1988)**.

No caso do construtivismo de Markov, diferenciamos duas situações: uma delas envolvendo funções parciais e a outra envolvendo funções totais. Se a função é parcial,

então a condição de existência na Definição 1.2.12, dado que a função *mr* é recursiva primitiva e portanto o predicado envolvido é decidível, assegura que o valor é definido construtivamente no sentido de Markov. Por outro lado, se a função é total, conforme a Definição 2.4.1 de programa *n*-regular, temos que considerar uma fórmula da forma $\forall x \exists y f(x,y) = 0$, onde f é uma função recursiva primitiva. Considerando os escrúpulos construtivistas, só podemos partir da fórmula $\forall x \neg \forall y \neg f(x,y) = 0$. Como f é recursiva primitiva, também dispomos de $\forall x \forall y [f(x,y) = 0 \vee f(x,y) \neq 0]$. A partir destas fórmulas, do (MP) e do fato de que na lógica minimal de Johansson deduzimos $\neg \neg \exists x A$ a partir de $\neg \forall x \neg A$, podemos provar facilmente que $\forall x \exists y f(x,y) = 0$, onde o quantificador existencial é construtivo no sentido de Markov.

Como foi indicado, a objeção de Péter foi apresentada apenas em 1957. Ela não disse nada, naquela ocasião, em relação às funções computáveis parciais. Mas seria estranho que ela ignorasse este conceito em 1957, introduzido, como vimos, por Kleene, quase vinte anos antes! De fato, em relação à primeira edição de seu livro, de 1950, Péter expressa o seguinte:

> "Introduzi o conceito de função recursiva parcial mais cedo e com maior ênfase." (**Péter, 1967, p. 9**).

Isto significa claramente que, em 1950, *sete anos antes* da apresentação de seu argumento contrário à (TC), Péter *não* ignorava o conceito de função parcial computável. Além disso, no artigo onde aparece a sua objeção, Péter cita, com outros propósitos, o artigo onde Kleene introduz a noção de função computável parcial (**Kleene, 1938**).

Quanto aos comentários iniciais deste capítulo de Shepherdson e Sturgis, Wang e Davis, não fica claro que a objeção de Péter só se aplica ao caso das funções totais, salvo seja omitido o quantificador existencial que aparece na condição da Definição 1.2.12 definindo o conceito de função computável de maneira intuitiva ou definindo-o, de maneira complexa, ao estilo do terceiro capítulo de Troelstra e van Dalen, assunto que teria merecido uma explicitação detalhada por parte de Shepherdson e Sturgis, Wang e

Davis. Em qualquer caso, pelo menos de um ponto de vista didático, também deveria explicitar-se que o que está em jogo aqui é um exemplo do paradoxo do inventor: o caso das funções totais dá origem a um problema em relação ao construtivismo, então prefere-se considerar o caso das funções parciais, mais abrangente, onde tal problema não aparece, pelo menos se se toma o cuidado de não usar quantificadores existenciais ao estilo da Definição 1.2.12.

CAPÍTULO 3

COMPUTABILIDADE SEM FORMALISMO

Neste capítulo nos propusemos a apresentar uma versão das funções computáveis onde todos os conceitos estivessem diretamente definidos em termos de números naturais e onde o programa estivesse na memória, como é o caso dos computadores reais, com o objetivo de desenvolver sob esse enfoque a teoria da computabilidade. Na Seção 1 apresentamos alguns textos que motivaram a introdução dos conceitos deste capítulo. Nas Seções 2 e 3 apresentamos os conceitos básicos necessários para a definição do conceito de função computável da Seção 4, na qual provamos que as funções computáveis são μ–recursivas. Na Seção 5 provamos o inverso. Nas Seções 6 a 8 analisamos algumas questões típicas da área. Nas Seções 9 e 10 apresentamos os conceitos de computabilidade auto-referencial e automodificante e provamos que a complexidade não varia em relação à noção padrão. No final da Seção 10 esboçamos algumas definições para uma teoria dos vírus. Na Seção 11 analisamos a questão da concorrência e na Seção 12 apresentamos uma generalização da nossa definição de computabilidade.

3.1 CONSIDERAÇÕES HISTÓRICAS

É comum se encontrar afirmações como a seguinte, quando se comentam as diferentes noções que se usam para definir o conceito de função computável:

> "A definição original de função recursiva [...] depende da escolha de um *P*-simbolismo [...]" **(Hartley Rogers, 1969, p. 146)**

Porém, consideremos a seguinte definição do conceito de função μ-recursiva.

Definição 3.1.1 As funções *iniciais* são as funções totais:

$Z(x)=0$, para todo x,
$S(x) = x+1$, para todo x,
$U_i^n(\vec{x}) = x_i$, para todos $\vec{x}$, $n \geq 1$ e $1 \leq i \leq n$.

Definição 3.1.2 Uma função parcial $f(\vec{x})$ é *μ-recursiva* se é uma função inicial; ou existem funções parciais μ-recursivas $g_1(\vec{x}),\ldots,g_k(\vec{x})$ e $h(\vec{x})$ tais que $f(\vec{x}) \simeq h(g_1(\vec{x}),\ldots,g_k(\vec{x}))$; ou existem funções parciais μ-recursivas $g(x_1, \ldots, x_{n-1})$ e $h(x_1, \ldots, x_{n+1})$ tais que $f(x_1, \ldots, x_{n-1}, 0) \simeq g(x_1, \ldots, x_{n-1})$ e $f(x_1, \ldots, x_{n-1}, y+1) \simeq h(x_1, \ldots, x_{n-1}, f(x_1, \ldots, x_{n-1}, y), y)$, para todo y; ou existe uma função parcial μ-recursiva $g(x_1, \ldots, x_{n+1})$ tal que $f(\vec{x}) \simeq \mu y[g(\vec{x}, y) = 0]$.

Não é claro que exista um simbolismo envolvido, ou, no caso que exista, qual seria esse simbolismo.

A situação é similar ao caso das funções recursivas primitivas. Gödel estava bem consciente de que para definir as funções recursivas primitivas não era preciso fazer referência a nenhum formalismo:

> "Agora inserimos entre parêntesis uma consideração que por enquanto *não tem nada a ver* com o sistema formal *P*, e damos em primeiro lugar a seguinte definição: uma função da teoria dos números [...]"(grifo nosso) **(Gödel, 1986, p. 156)**.

Webb ressaltou este caráter da μ-recursão:

> "Deste modo, de maneira diferente das funções recursivas gerais de Gödel, as funções μ-recursivas não são definidas fazendo referência a qualquer *formalismo* para expressar equações cujos cálculos procedem por regras fixas que governam o uso daquele formalismo: [...]" (**Webb, 1980, p. 206-207**).

Os demais autores que conhecemos parecem ignorar este fato. Considere-se, por exemplo, o seguinte comentário recente:

> "As funções μ-recursivas são uma classe matematicamente definível de funções *quase* independentes da sintaxe e do formalismo" (grifo nosso) (**Soare, 1996, p. 229**).

Neste capítulo, apresentamos um conceito preciso de função computável que, como é o caso da noção de função μ-recursiva, não faz referência a nenhum formalismo.

Como é sabido, a primeira definição precisa do conceito de computabilidade intuitiva foi dada por Gödel em 1931. Aparece implicitamente no enunciado do seguinte teorema:

> "Teorema V. Para toda relação recursiva $R(x_1, \ldots, x_n)$ existe um SIGNO DE RELAÇÃO r de n-lugares (com as VARIÁVEIS LIVRES $u_1, u_2, \ldots, u_n$) tal que para todas as n-uplas de números $(x_1, \ldots, x_n)$ temos que
>
> $$R(x_1, \ldots, x_n) \rightarrow Bew[Sb(r^{u_1 \ldots u_n}_{Z(x_1)\ldots Z(x_n)})],$$
>
> $$\neg R(x_1, \ldots, x_n) \rightarrow Bew[Neg(Sb(r^{u_1 \ldots u_n}_{Z(x_1)\ldots Z(x_n)}))]$$"(**Gödel, 1986, p. 170**).

Observamos que Gödel usa a relação *numérica Bew* ao invés de dizer que as fórmulas correspondentes são demonstráveis no sistema formal que está usando. Kleene destacou esta atitude de Gödel:

> "O Teorema V ilustra a tendência de Gödel em falar em termos de *seus números*, em lugar de diretamente em termos dos objetos formais (que nós preferimos como mais compreensíveis)" (grifo nosso) (**Kleene, apud Gödel, 1986, p. 130-131**).

Em geral, não se costuma dar as definições de funções computáveis em termos exclusivamente numéricos: os programas são *sucessões* finitas de *instruções*, as instruções são *sucessões* finitas de *símbolos*, etc. Isto faz com que, quando se quer provar que todas as funções computáveis são, por exemplo, μ-recursivas, seja necessário uma aritmetização do formalismo, processo que costuma requerer bastante trabalho.

Em geral, as noções de computabilidade também não costumam colocar os programas na memória. Os programas costumam ser sucessões finitas de instruções que estão "fora" da memória do computador, como ocorre na versão original de Turing para suas máquinas. Algumas exceções a estão em **Kaphengst (1959)** e o conceito de computador digital infinito que aparece em **Monk (1976, p. 67)**.

3.2 O PASSO DE COMPUTAÇÃO

Nossa idéia consiste em ver o computador, a memória, as instruções e os programas com olhos aritméticos. A memória será um número qualquer diferente de zero, interpretado como decomposto em todos os fatores primos. O computador será uma função de N^* em N^* (onde $N^* = N - \{0\}$), quer dizer, uma função que a uma memória qualquer associa uma outra memória, e as instruções e programas também serão números (≥ 0) com certas formas.

No caso da memória, a notação habitual para o i-ésimo número primo e seu expoente, quer dizer, p , com $i, n \geq 0$, é sugestiva, no sentido que pode ser lida como que o i-ésimo endereço da memória contém o número n.

Normalmente, as expressões $Z(n)$, $S(n)$, $T(m,n)$ e $J(m,n,q)$ são usadas para denotar, respectivamente, as instruções de colocar um zero no endereço n, colocar o sucessor no endereço n, transferir o número do endereço m para o endereço n e, caso os

conteúdos dos endereços m e n sejam o mesmo, pular para a instrução q. Porém, usaremos estas expressões como abreviações dos números das formas que indicaremos no começo da Seção *3.3*, mas será útil levar em conta o significado em termos de instruções.

Agora definimos o conceito de passo de computação.

Definição 3.2.1 Um *passo de computação* é, simplesmente, uma função $c: N^* \rightarrow N^*$ definida por:

$$c(d) = \begin{cases} p_0^0 \prod_{j=1}^{\infty} p_j^{(d)_j} & se\ (d)_{(d)_0} = 0, \\ p_0^{(d)_0+2} \prod_{j=1}^{a \dot{-} 1} p_j^{(d)_j}\ p_a^0 \prod_{j=a+1}^{\infty} p_j^{(d)_j} & se\ (d)_{(d)_0} = 1, \\ p_0^{(d)_0+2} \prod_{j=1}^{a \dot{-} 1} p_j^{(d)_j}\ p_a^{(d)_a+1} \prod_{j=a+1}^{\infty} p_j^{(d)_j} & se\ (d)_{(d)_0} = 2, \\ p_0^{(d)_0+3} \prod_{j=1}^{b \dot{-} 1} p_j^{(d)_j}\ p_b^{(d)_a} \prod_{j=b+1}^{\infty} p_j^{(d)_j} & se\ (d)_{(d)_0} = 3, \\ p_0^{(d)_{[(d)_0+3]}} \prod_{j=1}^{\infty} p_j^{(d)_j} & se\ (d)_{(d)_0} = 4\ e\ a = b, \\ p_0^{(d)_0+4} \prod_{j=1}^{\infty} p_j^{(d)_j} & se\ (d)_{(d)_0} = 4\ e\ a \neq b, \\ d & caso\ contrário, \end{cases}$$

onde $a = (d)_{[(d)_0+1]}$, $b = (d)_{[(d)_0+2]}$ e a notação $x \dot{-} y$ indica a diferença truncada.

Para entender rapidamente esta definição é suficiente esquecer as expressões com o símbolo Π e observar que as diferenças com relação a d correspondem apenas aos expoentes dos primos p_0 (=2), p_a e p_b, dependendo do expoente do primo $p_{(d)_0}$ de d, onde $(n)_i$ é definido como o expoente do i-ésimo primo na decomposição de n enquanto

produto de fatores primos e considerando que o *0*-ésimo primo é *2*, o *1*-ésimo primo é *3* etc.

Intuitivamente, o número *0* corresponde à instrução de *parar*, o *1* à instrução de *colocar um zero* no lugar do número que se encontra no endereço seguinte da memória, o *2* à instrução de *colocar o sucessor* no lugar do número que se encontra no endereço seguinte da memória, o *3* à instrução de *transferir* o conteúdo da célula com o número que aparece no endereço seguinte para a célula com o número que aparece no endereço seguinte do seguinte, e *4* à instrução de *pular*. Quer dizer que a instrução *0* não tem argumento, a *1* e a *2* têm um argumento, a *3* tem dois argumentos e a *4* tem três argumentos.

Aritmeticamente, não há problema em se considerar simultaneamente as modificações na memória. Mas, num computador real elas deveriam ser feitas no registro contador do programa (o registro p_0) no último momento.

Propriedade 3.2.2 Se $(d)_0 = 0$, então $c(d) = d$.

Agora, interessa-nos compor a função *c* consigo mesma, um número arbitrário de vezes, isto é, interessam-nos as operações seguintes:

$c(d)$,
$c(c(d))$,
$\vdots$
$c(c(\ldots c(d)\ldots))$.

Para esse fim, definimos a seguinte função:

Definição 3.2.3 $c^*(d, 0) = d$;
$c^*(d, t+1) = c(c^*(d, t))$.

É fácil provar a seguinte propriedade, útil para o que vem a seguir.

Propriedade 3.2.4 (a) Se $(d)_0 = 0$, então $c^*(d,t) = d$, para todo t;
(b) $c^*(d, t + s) = c^*(c^*(d, t), s)$.

Lema 3.2.5 Se $[c^*(d, t)]_0 = [c^*(d, s)]_0 = 0$, então $c^*(d, t) = c^*(d, s)$.

Prova: Se $t = s$, não temos o que provar. Suponhamos que $s < t$. Então

$c^*(d, t) = c^*(d, s+(t-s))$, porque $t = s+ (t-s)$
$= c^*(c^*(d, s), t-s)$, pela Propriedade 3.2.4 (b)
$= c^*(d, s)$, porque $c^*(d, s)_0 = 0$.

Analogamente, para o caso que $t < s$. □

Comentário 3.2.6 A prova anterior já vale na aritmética recursiva primitiva (*PRA*), apesar de parecer usar o princípio do terceiro excluído, porque em *PRA* é possível decidir a lei de tricotomia. Para um estudo introdutório da *PRA*, o leitor pode consultar o capítulo terceiro de **Troelstra & van Dalen (1988)**.

3.3 AS FUNÇÕES $CONFIG_n$

Para definir as funções $Config_n$, que serão utilizadas posteriormente, é preciso usar as noções seguintes.

Um *programa* é qualquer número natural.

Seja P um programa. Pode-se escrever de um único modo da seguinte forma (e. g. ver **Cutland, 1992, p. 74**):

$$P = \pi(a_1, a_2, \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2+1} + \cdots + 2^{a_1+a_2+\ldots+a_k+(k-1)} \dot{-} 1.$$

Definimos as *instruções* de P como os números $a_1, a_2, \ldots, a_k$.

Seja i uma instrução. Então, pode ser provado que i tem uma *única* das seguintes formas:

0,

$Z(n) = 4n$,

$S(n) = 4(n-1) + 1$,

$T(m, n) = 4\pi(m - 1, n - 1) + 2$,

$J(m, n, q) = 4\zeta(m-1, n-1, q - 1) + 3$,

onde m, n e $q \geq 1$ e π e ζ são as seguintes funções recursivas primitivas:

$\pi(m, n) = 2^m(2n + 1) \dot{-} 1$,

$\zeta(m, n, q) = \pi(\pi(m, n), q)$.

Observamos que as funções π e ζ correspondem, intuitivamente, a uma enumeração recursiva primitiva de pares e ternas de números naturais, respetivamente.

Definimos os *nomes* e os *argumentos* de uma instrução i da maneira seguinte.

O *nome* de uma instrução da forma 0 é 0 e ela não tem *argumentos*. O *nome* de uma instrução de forma $Z(n)$ é 1 e seu único *argumento* é n. O *nome* de uma instrução de forma $S(n)$ é 2 e seu único *argumento* é n. O *nome* de uma instrução de forma $T(m,n)$ é 3 e os seus únicos *argumentos* são m e n. Finalmente, o *nome* de uma instrução de forma $J(m, n, q)$ é 4 e os seus únicos *argumentos* são m, n e q.

Denominamos *códigos* de um programa aos nomes e argumentos das instruções do programa. Usamos a expressões *i-ésimo código de um programa P* e *i-ésima instrução de um programa P* no sentido intuitivo, começando com $i = 1$.

Agora podemos definir as seguintes funções, motivados por **Cutland (1992)**:

Definição 3.3.1 $\rho(P)$ = *max*{argumentos das instruções de P excetuando os terceiros argumentos das instruções de forma $J(m, n, q)$},

$ln(P)$ = quantidade de instruções do programa P,

$lnc(P)$ = a quantidade de códigos do programa P,

$lnca(P,i)$ = a quantidade de códigos do programa P anteriores à i-ésima instrução,

$gn(P,i)$ = i-ésima instrução do programa P,

$d_n(P,i) = max[n, \rho(P)]+1+lnca(P, i)$ (i. e. a direção da i-ésima instrução do programa P aplicada a n argumentos),

$$recod_n(P,c) = \begin{cases} d_n(P,q) & \text{se c é o código do terceiro argumento de uma instrução } J(\overline{m},\overline{n},\overline{q}); \\ o\ c-ésimo\ código\ do\ programa\ P & \text{caso contrário.} \end{cases}$$

Observe-se que as funções gn e $recod_n$ podem ficar definidas inclusive para $i > ln(P)$ e $lnc(P)$, respectivamente.

Observamos ainda que os exemplos de instruções e programas introduzidos no Capítulo 1 podem facilmente ser expressos segundo as novas definições introduzidas nesta seção.

Pod ser provado que as funções assim definidas são recursivas primitivas (consultar e.g. **Cutland, 1992, p. 97**).

Agora podemos definir as funções seguintes, para $n \geq 1$:

$$Config_n(P, \vec{x}) = p_0^{1+max(\rho(P),n)} \prod_{i=1}^{n} p_i^{x_i} \prod_{i=1}^{lnc(P)} p_{i+max(\rho(P),n)}^{recod_n(P,i)}.$$

Intuitivamente, $1+ max(\rho(P), n)$ é um endereço de memória *seguro*, a partir do qual pode-se guardar os códigos de P sem perigo de que, quando se efetue a computação, eles se auto-modifiquem, deixando espaço suficiente de memória para os cálculos de P e para armazenar a n-upla a ser computada.

Intuitivamente $Config_n(P, \vec{x})$ corresponde ao conteúdo de memória de um computador, quando o contador de programa tem a direção do endereço onde encontra-se a primeira instrução do programa P; nos primeiros n endereços encontram-se números naturais $\vec{x}$; a partir do endereço com direção imediata seguinte ao número $max(\rho(P), n)$ encontram-se os códigos do programa P. A computação começa a sua tarefa com a "área de trabalho zerada". Supomos que os programas trabalham com a sua área "zerada", com exceção da n–upla a ser computada.

A prova da propriedade seguinte é trivial:

Propriedade 3.3.2 (a) $[Config_n(P, \vec{x})]_0 \neq 0$;

(b) Para todo numero t, se $c^*[Config_n(P, \vec{x}), t)]_0 \neq 0$, então

$1 + max(\rho(P), n) \leq c^*[(Config_n(P, \vec{x}), t)]_0$.

3.4 FUNÇÕES COMPUTÁVEIS

Notação 3.4.1 Sejam P um programa, $n \geq 1$ e $\vec{x}$ uma n-upla de naturais. A notação $P(\vec{x})\downarrow$ significa que existe um t tal que $[c^*(Config_n(P, \vec{x}), t)]_0 = 0$.

Notação 3.4.2 Sejam P um programa, $n \geq 1$ e $\vec{x}$ uma n-upla de naturais. A notação $P(\vec{x})\downarrow y$ significa que $P(\vec{x})\downarrow$ e que $[c^*(Config_n(P, \vec{x}), m)]_1 = y$, onde m é o mínimo t tal que $[c^*(Config_n(P, \vec{x}), t)]_0 = 0$.

Definição 3.4.3 Seja P um programa e $n \geq 1$. Então, a *função de aridade n definida por* P é φ^n_P: $N^n \rightharpoonup N$, com $\varphi^n_P(\vec{x}) \simeq y$ se, e somente se, $P(\vec{x})\downarrow y$.

Definição 3.4.4 Seja P um programa, $n \geq 1$ e f uma função n-ária. Dizemos que P n-computa f se $f = \varphi^n_P$.

Definição 3.4.5 A função f, de aridade n, é *computável* se existe um programa P tal que P n-computa f.

Como já dissemos, dada a nossa versão puramente numérica de computabilidade, é trivial provar que todas as funções computáveis são μ-recursivas. Trata-se de uma prova que normalmente precisa de várias páginas, onde se faz uma aritmetização do formalismo envolvido.

Proposição 3.4.6 Toda função computável é μ-recursiva.

Prova. Suponhamos que f, de aridade n, é computável. Então, pela Definicao 3.4.4 existe um programa P tal que f é igual à função de aridade n definida por P. Para provar que f é μ-recursiva, conforme a Definição 3.1.1, basta provar que f é a última função de alguma sucessão de funções que conservem a propriedade de ser μ-recursivas. Fixando P, consideremos a sucessão

1. $Config_n(P, \vec{x})$
2. $c^*[Config_n(P, \vec{x}), t]$
3. $(c^*[Config_n(P, \vec{x}), t])_0$
4. $\mu t[(c^*[Config_n(P, \vec{x}), t])_0 = 0]$
5. $c^*(Config_n(P, \vec{x}), \mu t[(c^*[Config_n(P, \vec{x}), t])_0 = 0])$
6. $F(\vec{x}) \simeq [c^*(Config_n(P, \vec{x}), \mu t[(c^*[Config_n(P, \vec{x}), t]_0 = 0])]_1$.

Já foi indicado que $Config_n$, c^* e $(n)_i$ são funções recursivas primitivas. Portanto, também são μ-recursivas. Mas as operações de composição e operação μ conservam a μ-recursividade. Portanto, a função F é μ-recursiva.

Resta provar que a função do passo 6 coincide com f. Em primeiro lugar, suponhamos que $F(\vec{x})\downarrow$. Então existe $\mu t[(c^*[Config_n(P, \vec{x}), t])_0 = 0]$. Pela Notação 3.4.1 e Definição 3.4.3, $f(\vec{x})\downarrow$. Suponhamos, reciprocamente, que $f(\vec{x})\downarrow$. Então, existe t tal que

$(c^*[Config_n(P, \vec{x}), t])_0 = 0$. Classicamente, pelo Princípio do Número Mínimo, existe $\mu t[(c^*[Config_n(P, \vec{x}), t])_0 = 0]$. Então, $F(\vec{x})\downarrow$. Portanto, $F(\vec{x})\downarrow$ se, e somente se, $f(\vec{x})\downarrow$.

Suponhamos, finalmente, que $F(\vec{x})$ e $f(\vec{x})$ estão definidas. É claro que

$F(\vec{x}) = [c^*(Config_n(P, \vec{x}), \mu t[(c^*[Config_n(P, \vec{x}), t]_0 = 0])]_1$.

Temos que $f(\vec{x}) = [c^*(Config_n(P, \vec{x}), m)]_1$, onde m é o mínimo t tal que $c^*[Config_n(P, \vec{x}), t]_0 = 0$. Portanto, $F(\vec{x}) = f(\vec{x})$. □

3.5 COMPUTABILIDADE DAS FUNÇÕES μ-RECURSIVAS

Nesta seção provamos o resultado recíproco da Proposição 3.4.7.

Proposição 3.5.1. Toda função μ-recursiva é computável.

Prova. Comecemos com a função $Z(x) = 0$, para todo x. Seja $P = \tau(Z(1))$. Intuitivamente, é o programa que colca um zero no endereço 1 da memória. Seja $x_1 \in N$.

$c^*(Config_1(P, x_1), 0) = Config_1(P, x_1) = p_0^2 p_1^{x_1} p_2^1 p_3^1$,

$c^*(Config_1(P, x_1), 1) = p_0^4 p_1^0 p_2^1 p_3^1$,

$c^*(Config_1(P, x_1), 2) = p_0^0 p_1^0 p_2^1 p_3^1$.

Portanto, $[c^*(Config_1(P, x_1), 2)]_0 = 0$ e $[c^*(Config_1(P, x_1), 2)]_1 = 0$.

Considere-se agora a função sucessor, definida por $S(x) = x+1$, para todo x. Seja $P = \tau(S(1))$. Intuitivamente, é o programa que coloca o sucessor no endereço 1 da memória. Seja $x_1 \in N$.

$c^*(Config_1(P, x_1), 0) = Config_1(P, x_1) = p_0^2 p_1^{x_1} p_2^2 p_3^1$,

$c^*(Config_1(P, x_1), 1) = p_0^4 p_1^{x_1+1} p_2^2 p_3^1$,

$c^*(Config_1(P, x_1), 2) = p_0^0 p_1^{x_1+1} p_2^2 p_3^1$.

Portanto, $[c^*(Config_1(P, x_1), 2)]_0 = 0$ e $[c^*(Config_1(P, x_1), 2)]_1 = x_1+1$.

Sejam consideradas agora as funções projeções dadas por $U_i^n(\vec{x}) = x_i$ para todos $\vec{x}$ com $1 \leq i \leq n$. Seja $P = \tau(T(i,1))$. Intuitivamente, é o programa que transfere o conteúdo do i-ésimo endereço para o endereço 1 da memória. Seja $\vec{x}$ uma n-upla de naturais:

$c^*(Config_n(P, \vec{x}),0) = Config_n(P,\vec{x}) = p_0^{n+1} p_1^{1} \ldots p_n^{x_n} p_{n+1}^{3} p_{n+2}^{i} p_{n+3}^{1}$,

$c^*(Config_n(P, \vec{x}),1) = p_0^{n+4} p_1^{i} p_2^{2} \ldots p_n\, p_{n+1}^{3} p_{n+2}^{i} p_{n+3}^{1}$,

$c^*(Config_n(P, \vec{x}),2) = p_0^{0}\, p_1^{i} p_2^{2} \ldots p_n\, p_{n+1}^{3} p_{n+2}^{i} p_{n+3}^{1}$,

Portanto, $[c^*(Config_n(P,\vec{x}), 2)]_0 = 0$ e $[c^*(Config_n(P, \vec{x}), 2)]_1 = x_i$.

Para provar que as regras de substituição, recursão e minimalização conservam a computabilidade, é útil usar a seguinte abreviação:

$$P[l_1,\ldots,l_n \rightarrow l] = \tau(T(l_1, 1),\ldots,T(l_n, n),Z(n+1),\ldots,Z(\rho(P)), gn(P, 1),\ldots,gn(P,ln(P)),T(1, l)).$$

É claro que a função $P[l_1,\ldots,l_n \rightarrow l]$ definida é recursiva primitiva, dado o caráter recursivo primitivo de todas as funções envolvidas. Intuitivamente, dado um programa P, o programa $P[l_1,\ldots,l_n \rightarrow l]$ computa o conteúdo de $l_1,\ldots,l_n$ como entrada, como se fosse o programa P, e deixa o resultado no endereço l. Observe-se que o número de instruções do programa $P[l_1,\ldots,l_n \rightarrow l]$ (isto é, $ln(P[l_1,\ldots,l_n \rightarrow l])$) é igual a $max(n, \rho(P)) + ln(P) + 1$, isto é, uma função recursiva primitiva de n e P.

Vejamos, agora, em primeiro lugar, o caso da regra de composição. Seja h obtida da seguinte forma:

$$h(\vec{x}) \simeq f[g_1(\vec{x}),\ldots, g_k(\vec{x})].$$

Suponhamos que F k-computa f e $G_1, \ldots, G_k$ n-computam, respectivamente, $g_1,\ldots, g_k$. Seja $m = max(n, k, \rho(F[m+n+1,\ldots,m+n+k \rightarrow 1]), \rho(G_1[m+1,\ldots,m+n \rightarrow m+n+1]), \ldots, \rho(G_k[m+1,\ldots,m+n \rightarrow m+n+k]))$. Intuitivamente, $m + 1$ resulta um endereço "seguro"

para guardar dados. Seja H o seguinte programa, que escreveremos verticalmente, para melhor compreensão:

$\tau[T(1, m + 1),$

$T(2, m + 2),$

$\vdots$

$T(n, m + n),$

$gn(G_1[m+1, \ldots, m + n \rightarrow m+n+1], 1),$

$\vdots$

$gn(G_1[m+1, \ldots, m + n \rightarrow m+n+1], ln(G_1[m+1, \ldots, m + n \rightarrow m+n+1]),$

$gn(G_2[m+1, \ldots, m + n \rightarrow m+n+2], 1),$

$\vdots$

$gn(G_2[m+1, \ldots, m + n \rightarrow m+n+2], ln(G_2[m+1, \ldots, m + n \rightarrow m+n+2]),$

$\vdots$

$gn(G_k[m+1, \ldots, m + n \rightarrow m + n + k], 1),$

$\vdots$

$gn(G_k[m + 1, \ldots, m + n \rightarrow m + n + k], ln(G_k[m + 1, \ldots, m + n \rightarrow m + n + k]),$

$gn(F[m + n + 1, \ldots, m + n + k \rightarrow 1], 1),$

$\vdots$

$gn(F[m + n + 1, \ldots, m + n + k \rightarrow 1], ln(F[m + n + 1, \ldots, m + n + k \rightarrow 1]).$

É intuitivamente claro que H computa a função h. Em qualquer caso, a prova detalhada não difere da prova habitual para uma máquina de registros: simplesmente requer uma indução trabalhosa.

Vejamos, em segundo lugar, o caso da regra de recursão. Seja h obtida da maneira seguinte:

$h(\vec{x}, 0) \simeq f(\vec{x}),$

$h(\vec{x}, y+1) \simeq g(\vec{x}, y, h(\vec{x}, y)).$

Suponhamos que F n-computa f e G $n+2$-computa g. Seja $m = max(n+2, \rho(F), \rho(G))$. Seja H o seguinte programa:

$$\tau[T(1, m + 1),$$
$$T(2, m + 2),$$
$$\vdots$$
$$T(n+1, m + n+1),$$
$$gn(F[1,2,\ldots, n \rightarrow m+n+3], 1),$$
$$\vdots$$
$$gn(F[1,2,\ldots, n \rightarrow m+n+3], ln(F[1,2,\ldots, n \rightarrow m+n+3]),$$
$$J(m+n+2, m+n+1,(n+1)+ln(F[1,2,\ldots, n \rightarrow m+n+3])+1+ln(G[m+1,\ldots, m + n, m+n+2,m+n+3 \rightarrow m+n+3])+3)$$
$$gn(G[m+1,\ldots, m + n, m+n+2,m+n+3 \rightarrow m+n+3], 1),$$
$$\vdots$$
$$gn(G[m+1,\ldots, m + n, m+n+2,m+n+3 \rightarrow m+n+3], ln(G[m+1,\ldots, m + n, m+n+2,m+n+3 \rightarrow m+n+3]),$$
$$S(m+n+2),$$
$$J(1,1,(n+1)+ ln(F[1,2,\ldots, n \rightarrow m+n+3])+1),$$
$$T(m+n+3,1)].$$

É intuitivamente claro que H computa a função h. Neste caso, a prova detalhada também é simplesmente uma indução trabalhosa.

Finalmente, consideremos o caso da regra de minimalização. Seja h obtida da seguinte maneira:

$$g(\vec{x}) \simeq \mu y\,[f(\vec{x}, y) = 0].$$

Suponhamos que F $(n+1)$-computa a função f. E seja $m = max(n+1, \rho(F))$. Seja G o programa seguinte:

$\tau[T(1, m + 1),$
$T(2, m + 2),$
$\vdots$
$T(n, m + n),$
$gn(F[m+1,m+2,\ldots,m+n+1 \rightarrow 1], 1),$
$\vdots$
$gn(F[m+1,m+2,\ldots,m+ n+1 \rightarrow 1], ln(F[m+1,m+2,\ldots,m+n+1 \rightarrow 1]),$
$J(1,m+n+2, n+ln(F[m+1,m+2,\ldots,m+n+1 \rightarrow 1])+4),$
$S(m+n+1),$
$J(1,1,n+1),$
$T(m+n+1,1)].$

É intuitivamente claro que G computa à função g. Neste caso, a prova detalhada é, novamente, uma indução trabalhosa. □

Com isto, fica completa a prova de que as funções recursivas são computáveis. Como terá sido observado, esta prova não difere essencialmente da prova corriqueira no caso, por exemplo, de uma máquina de registros.

3.6 COMPUTABILIDADE DAS FUNÇÕES UNIVERSAIS

Seja $n \geq 1$ e P um programa. Foram definidas as funções $f^n{}_P$. A seguir definimos as funções universais.

Definição 3.6.1 A função *universal* de aridade n é a função $U^n(P, \vec{x}) \cong \varphi^n{}_P(\vec{x})$.

É usual provar que as funções universais são computáveis. Como fica a prova em nossa teoria?

Proposição 3.6.2 As funções universais são computáveis.

Prova. Fixemos n. Seja a função universal n-ária $U^n(P, \vec{x})$. Então, da prova da μ-recursividade das funções computáveis temos que

$$U^n(P, \vec{x}) \simeq [c^*(Config_n(P, \vec{x}), \mu t\{[c^*(Config_n(P, \vec{x}), t)]_0 = 0\})]_1.$$

Mas as funções c^*, $Config_n$ e $(n)_i$ são recursivas (primitivas). Portanto, U^n é uma função recursiva. Mas foi provado que as funções recursivas são computáveis. Portanto, as U^n são computáveis. □

Não é difícil escrever o programa que computa as funções U^n. Usando programas para computar as funções $Config_n$, c e $(n)_i$ que chamamos, respectivamente, de $CONFIG_n$, C e EXP, podemos escrever o programa seguinte, onde $m = max(\rho(C), \rho(EXP))$:

```
        gn(CONFIG_n, 1)
        ⋮
        gn(CONFIG_n, ln(CONFIG_n))
        Z(m+2)              Escreve o segundo argumento para a função EXP
volta:  gn(C, 1)
        ⋮
        gn(C, ln(C))
        T(1,m+1)            Conserva o valor do primeiro endereço
        T(m+2, 2)
```

	gn(EXP, 1) ⋮ *gn(EXP, ln(EXP))*	Calcula o *0*-ésimo expoente
	J(1,m+2, sair)	
	T(m+1,1)	Recupera Informação
	J(1,1,volta)	
sair:	*S(m+2)*	
	T(m+1,1)	
	T(m+2,2)	
	gn(EXP, 1) ⋮ *gn(EXP, ln(EXP))*	Calcula o *1*-ésimo expoente (o valor *f(x)*).

O desenvolvimento ulterior de nossa teoria não difere do tratamento usual. Tomemos como exemplo a prova de que a função seguinte não é computável:

$$g(x) = \begin{cases} 1 & \text{se } f_x^1 \text{ é total;} \\ 0 & \text{se } f_x^1 \text{ não é total.} \end{cases}$$

Para isso, usando a função universal, definimos uma função h da maneira seguinte:

$$h(x) = \begin{cases} U^2(x,x)+1 & \text{se } g(x)=1; \\ 0 & \text{se } g(x)=0. \end{cases}$$

É claro que h é total. Mas difere de toda função computável f^1_y, pois suponhamos que $h = f^1_y$. Então, $h(y) = f^1_y(y)$. Mas, como h é total, $h(y) = U^2(y, y) + 1$. Mas $f^1_y(y) = U^2(y, y)$. Então, $h \neq f^1_y$.

Então, a função h não é computável. Mas, se g fosse computável, então h também o seria. Portanto, g não é computável.

3.7 O TEOREMA *s-m-n*

A função universal nos permite "passar" de funções de n variáveis para uma função de $n + 1$ variáveis. Na teoria da recursão usa-se também um teorema que permite "passar" de uma função de $m + n$ variáveis para uma de n variáveis, com $n \geq 1$. É habitual chamar esse Teorema *s-m-n*. A sua formulação é a seguinte.

Teorema 3.7.1 Se $m, n \geq 1$, então existe uma função $(m + 1)$-ária computável total $s_n(e, x_1, \ldots x_m)$ tal que

$$\varphi_e^{(m+n)}(x_1, \ldots x_m, y_1, \ldots, y_n) \simeq \varphi^{(n)}_{s_n^m(e,x_1,\ldots,x_m)}(y_1, \ldots, y_n).$$

Prova: Basta considerar que $s_n^m(e, x_1, \ldots, x_m) = \tau(T(n, m + n), \ldots, T(2, m + 2), T(1, m + 1), R(1, x_1), R(2, x_2), \ldots, R(m, x_m), gn(e, 1), \ldots, gn(e, ln(e)))$, onde $R(i, x)$ é a sucessão $Z(i), S(i), \ldots, S(i)$ (x vezes). Este programa é intuitivamente inteligível. A prova de que a função é total, apesar de trabalhosa, não difere das provas habituais em computabilidade (ver **Mendelson, 1997, p. 330**).

3.8 O PROBLEMA DA PARADA

Tomemos como outro exemplo o problema da parada. Para provar que não se pode decidir se uma função pára, ao computar um x dado, temos que provar que a seguinte função não é computável:

$$g(x,y) = \begin{cases} 1 & , se\ \varphi_x(y)\downarrow; \\ 0 & , caso\ contrário. \end{cases}$$

Para isso, em primeiro lugar, provaremos que a seguinte função não é computável:

$$f(x)=\begin{cases}1 & ,se\ \varphi_x(x)\downarrow;\\ 0 & ,caso\ contrário.\end{cases}$$

Consideremos a seguinte função:

$$h(x)=\begin{cases}\uparrow & ,se\ \varphi_x(x)\downarrow;\\ 0 & ,caso\ \ contrário.\end{cases}$$

Evidentemente, $h(x) \simeq \mu y[f(x) + y = 0]$. Então, h é computável se f é computável. Mas é claro que h não pode ser computável, porque difere de toda função φ_x no ponto x. Voltemos agora à nossa função g. É claro que $f(x) = g(x, x)$, para todo x. Mas vimos que f não é computável. Então, $g(x, x)$ não é computável. Mas $g(x, x)$ é um caso particular de $g(x, y)$. Então, g também não é computável.

3.9 COMPUTABILIDADE AUTO-REFERENCIAL

Na Seção 3 definimos as funções $Config_n$ de forma que o programa ficasse numa região da memória que não permitisse a sua auto-referência ou auto-modificação. No entanto, é interessante considerar estas duas possibilidades, visto que nas versões habituais da computabilidade isto não é possível, dado que os programas sempre estão "fora" da memória; nas versões usuais isto só pode ser *simulado*. Nesta seção definiremos a noção de função computável com auto-referência.

Definição 3.9.1 $\sigma(P) = max$\{argumentos das instruções de forma $Z(n)$, $S(n)$, e segundos argumentos das instruções da forma $T(m, n)$\}

Com esta noção se evitará que os programas tenham automodificação, permitindo a auto-referência. É claro que, para todo P, $\sigma(P) \leq \rho(P)$.

$$d_n^{ar}(P, i) = max[n, \sigma(P)]+1+lnca(P, i)$$

$$recod_n^{ar}(P,c)=\begin{cases}d_n^{ar}(P,q)\text{, se } c \text{ é o código do terceiro argumento de uma instrução } J(\overline{m},\overline{n},\overline{q});\\ o\ c-ésimo\ código\ \ do\ programa\ P & \text{, caso contrário.}\end{cases}$$

Conseqüentemente, podemos definir:

Definição 3.9.2 $\text{Config}_n^{ar}(P, \vec{x}) = p_0^{1+\ a\ (\sigma(P),n)} \prod_{i=1}^{n} p_i^{x_i} \prod_{i=1}^{\ln c(P)} p_{i+max(\sigma(P),n)}^{recod_n^{ar}(P,i)}$.

A seguir, podemos estabelecer quando uma função é computável com auto-referência.

Definição 3.9.3 A função f de aridade n é dita *computável com auto-referência*, o que é denotado por *computável* ^{ar} se existe um número P tal que para todo $\vec{x}$ cumpre-se que:

$$f(\vec{x}) \simeq [c^*(\text{Config}_n^{ar}(P, \vec{x}), \mu t\{[c^*(\text{Config}_n^{ar}(P, \vec{x}), t)]_0 = 0\})]_1.$$

É natural pensar que pode ser provada a proposição seguinte.

Proposição 3.9.4 f é computável ^{ar} *se, e somente se, f é computável.*

Prova. Suponhamos que $f(\vec{x})$ é computável ^{ar} com o programa P. Sejam $v_1, v_2, \ldots, v_q$ os endereços de memória que se encontram no intervalo $[1+max(\rho(P),n), lnc(P)+max(\sigma(P),n)]$, tais que são primeiros argumentos de instruções da forma $T(m, n)$ ou primeiros ou segundos argumentos de instruções da forma $J(m, n, q)$. Evidentemente, se não existe nenhum v_i com essa propriedade, então $\sigma(P) = \rho(P)$. Então, seja Q o programa seguinte:

$$\left[\prod_{i=1}^{\ln c(P)} p_{i+max(\sigma(P),n)}^{recod_n(P,i)}\right]_{v_1} vezes \left\{\begin{array}{l} \tau[\\ \vdots \\ \\ S(v_1) \\ \vdots \end{array}\right. \begin{array}{l} S(v_1) \\ \\ \vdots \end{array}$$

$$\left[\prod_{i=1}^{\ln c(P)} P_{i+max(\sigma(P),n)}^{recod_n(P,i)}\right]_{v_q} vezes \begin{cases} S(v_q) \\ \vdots \\ S(v_q) \\ gn(P,1) \\ \vdots \\ gn(P,ln(P))] \end{cases}.$$

Intuitivamente, é claro que o programa Q computa a função $f(\vec{x})$ no sentido original.

Reciprocamente, suponhamos agora que $f(\vec{x})$ é computável pelo programa P. Intuitivamente, o programa P, ao ficar "mais à esquerda", pode conter argumentos como os v_i mencionados na prova da condição necessária, que não estão inicializados em zero. Para resolver este inconveniente, basta substituir tais v_i por $v_i + lnc(P)$. Chamemos de Q ao programa que resulta de P fazendo esta substituição. Então, é claro que Q computa ^{ar} a $f(\vec{x})$. □

3.10 COMPUTABILIDADE AUTOMODIFICANTE

Nesta seção definiremos a noção de função computável com auto-modificação. O conceito de automodificação pode ser *simulado* nos conceitos habituais de funções computáveis (máquinas de Turing, máquinas de registros, etc.). Na nossa teoria, não precisa ser simulado. Pode ser definido diretamente. Para isso, definimos:

Definição 3.10.1 $d_n^{am}(P,i) = n + 1 + lnca(P,i)$

$$recod_n^{am}(P,c) = \begin{cases} d_n^{am}(P,q) & \text{, se } c \text{ é o código do terceiro argumento de uma instrução } J(\overline{m},\overline{n},\overline{q}); \\ o\ c-\acute{e}simo\ c\acute{o}digo\ do\ programa\ P & \text{, caso contrário.} \end{cases}$$

Definição 3.10.2 Dados um programa P e um número n,

$$Config \quad (P, \vec{x}) = p_0^{+1} \prod_{i=1}^{n} p_i^{x_i} \prod_{i=1}^{\ln c(P)} p_{n+i}^{recod_n^{am}(P,i)} .$$

A seguir, podemos estabelecer quando uma função é computável com auto-modificação.

Definição 3.10.3 A função f de aridade n é dita *computável com automodificação*, o que se denota por *computável* ^{am} se existe um número P tal que para todo $\vec{x}$ cumpre-se que:

$$f(\vec{x}) \simeq [c^*(Config_n^{am}(P, \vec{x}), \mu t\{[c^*(Config_n^{am}(P, \vec{x}), t)]_0 = 0\})]_1.$$

É natural pensar que pode ser provado o seguinte.

Proposição 3.10.4 f é computável ^{am} se, e somente se, f é computável.

Prova. Suponhamos que $f(\vec{x})$ é computável ^{am} com o programa P. Vamos provar que $f(\vec{x})$ é computável ^{ar}. Pelo resultado anterior segue-se que $f(\vec{x})$ é computável. Consideremos o seguinte programa, onde q é o número de códigos do programa P:

$T(max(\sigma(P),n) + 4 + 3q + 1, n + 1)$ ┐
⋮ ├ leitura dos códigos do programa P (como instrução) nos endereços $n+1, \ldots, n+q$
$T(max(\sigma(P),n) + 4 + 3q + q, n + q)$
$J(1,1, q+3)$
P
$gn(R,1)$ ┐
⋮ ├ O programa R lê os endereços $n+1, \ldots, n+q$ e deixa o valor original de P em $n+1$.
$gn(R, ln(R))$
$T(n+1,n+2)$
⋮
$T(1,2)$
$T(n+2,1)$

$gn(CONFIG_m^{am},1)$

$\vdots$

$gn(CONFIG_m^{am},ln(CONFIG_m^{am}))$

$Z(2)$

volta: $gn(C,\ 1)$

$\vdots$

$gn(C,\ ln(C))$

$T(1,3)$ Conserva o valor do primeiro endereço

$gn(EXP,\ 1)$

$\vdots$ } Calcula o 0-ésimo expoente

$gn(EXP,\ ln(EXP))$

$J(1,2,\ sair)$

$T(3,1)$ Recupera informação

$J(1,1,volta)$

sair: $T(3,1)$

$S(2)$

$gn(EXP,\ 1)$

$\vdots$ } Calcula o 1-ésimo expoente (o valor $f(x)$)

$gn(EXP,\ ln(EXP))$

Intuitivamente, é claro que o programa apresentado simula de maneira autoreferencial a execução do programa P sob a forma automodificável. □

Em lugar do programa apresentado, pode-se tomar o programa que resulta da substituição das instruções de $T(max(\sigma(P),n) + 4 + 3q + 1,\ n + 1)$ até $gn(R,\ ln(R))$ pelas instruções seguintes:

$$P\ vezes\begin{cases} S(n+1) \\ \vdots \\ S(n+1). \end{cases}$$

O novo programa, claramente, não tem auto-referência.

Além disso, intuitivamente, ambos os programas têm a mesma complexidade que o programa P, pois estes programas simulam a execução do programa P, tendo, para uma n-upla dada, o mesmo número de "voltas" que o programa P. Isto significa que por meio da computabilidade automodificante não é possível diminuir o grau de complexidade.

Reciprocamente, suponhamos que f é computável pelo programa P. Consideremos o programa Q que resulta de se fazer as seguintes modificações no programa P: substituir todas as instruções da forma $Z(p)$, $S(p)$, $T(p, q)$ ou $J(p, q, r)$ com $n+1 \leq p$ ou $q \leq max(\rho(P), n)$, pelas instruções $Z(1 + lnc(P) + p)$, $S(1 + lnc(P) + p)$, $T(1 + lnc(P) + p, 1 + lnc(P) + q)$ e $J(1 + lnc(P) + p, 1 + lnc(P) + q, r)$. Então, o novo programa Q computa^{am} a função $f(\vec{x})$. A adição do número 1 nas expressões $1 + lnc(P)+p$ e $1 + lnc(P)+q$ se deve a que precisamos deixar com zero o endereço seguinte ao último código do programa Q. Intuitivamente, a conseqüência da substituição é que a área de trabalho fica transposta para o final da memória utilizada. Por outro lado, no caso do programa P, a área de trabalho fica entre a n-upla computada e o programa P.

O conceito de computabilidade ^{am} pode ser útil para o desenvolvimento de uma teoria dos vírus, para a qual introduzimos as seguintes definições.

Definição 3.10.5 Sejam P um programa e $n \geq 1$. Dizemos que P é *n-débil* se existe $\vec{x} \in N^n$ tal que $P(\vec{x})$ escreve na área do programa P.

Definição 3.10.6 Seja P um programa, $n \geq 1$ e $\vec{x}$ uma n-upla de naturais. Dizemos que $\vec{x}$ *é vírus para* P se existe t tal que $(c^*[\, Config^{am}_{M}(P, \vec{x}), t])_0 = d$ e $(c^*[\, Config^{am}_{M}(P, \vec{x}), t])_d \neq [Config^{am}_{M}(P, \vec{x})]_d$.

Intuitivamente, o controle vai para alguma coisa que é resultado da ação de P sobre $\vec{x}$.

Definição 3.10.7 Seja P um programa, com $n \geq 1$. Um *anti-vírus para* P é um programa Q tal que para todo $\vec{x}$, $\vec{x}$ não é um vírus para PQ, onde PQ é a concatenação de P e Q definida de maneira habitual.

Definição 3.10.8 Seja $n \geq 1$. Um *anti-vírus universal de grau n* é um programa Q tal que, para todo P e todo $\vec{x}$, $\vec{x}$ não é vírus para PQ.

Nas definições acima no lugar da concatenação entre programas, parece ser possível optarnos pela aplicação entre programas.

Conjecturamos que é possível provar que não existem anti-vírus universais e que a n-debilidade não é computável.

3.11 COMPUTABILIDADE CONCORRENTE

Nesta seção tratamos a questão de dois programas agindo concorrentemente. A nossa interpretação é facilmente generalizável para o caso de n programas.

Uma primeira maneira de obter programas agindo concorrentemente é substituir a nossa definição de passo de computação pela seguinte definição.

Definição 3.11.1 Um *passo de computação concorrente* é, simplesmente, uma função con_2: $N^* \rightarrow N^*$ definida por :

$con_2(d) = d$, caso $(d)_0 = (d)_1 = (d)_2$;

caso contrário,

$$con_2(d) = \begin{cases} p_0^{(d)_2} p_1^0 \prod_{j=2}^{\infty} p_j^{(d)_j} & se (d)_0 = (d)_1 \ e \ se \ (d)_{(d)_0} = 0, \\ p_0^{(d)_2} p_1^{(d)_0+2} \prod_{i=2}^{a-1} p_i^{(d)_i} p_a^0 \prod_{i=a+1}^{\infty} p_j^{(d)j} & se \ (d)_0 = (d)_1 \ e \ (d)_{(d)_0} = 1, \\ p_0^{(d)_2} p_1^{(d)_0+2} \prod_{i=1}^{a-1} p_i^{(d)_i} p_a^{(d)_a+1} \prod_{i=a+1}^{\infty} p_i^{(d)_i} & se (d)_0 = (d)_1 \ e \ se \ (d)_{(d)_0} = 2, \\ p_0^{(d)_2} p_1^{(d)_0+3} \prod_{i=2}^{b-1} p_i^{(d)_i} p_b^{(d)_a} \prod_{i=b+1}^{\infty} p_i^{(d)_i} & se \ (d)_0 = (d)_1 \ e \ (d)_{(d)_0} = 3, \\ p_0^{(d)_2} p_1^{(d)_{[(d)_0+3]}} \prod_{i=2}^{\infty} p_i^{(d)_i} & se \ (d)_0 = (d)_1 \ , (d)_{(d)_0} = 4 \ e \ a = b, \\ p_0^{(d)_2} p_1^{(d)_0+4} \prod_{i=2}^{\infty} p_i^{(d)_i} & se \ (d)_0 = (d)_1, (d)_{(d)_0} = 4 \ e \ a \neq b, \\ p_0^{(d)_1} p_1^{(d)_1} p_2^0 \prod_{i=3}^{\infty} p_i^{(d)_i} & se (d)_0 = (d)_2 \ e \ (d)_{(d)_0} = 0, \\ p_0^{(d)_1} p_1^{(d)_1} p_2^{(d)_0+2} \prod_{i=3}^{a-1} p_i^{(d)_i} p_a^0 \prod_{i=a+1}^{\infty} p_i^{(d)_1} & se \ (d)_0 = (d)_2 \ e \ (d)_{(d)_0} = 1, \\ p_0^{(d)_1} p_1^{(d)_1} p_0^{(d)_0+2} \prod_{i=3}^{a-1} p_i^{(d)_i} p_a^{(d)_a+1} \prod_{i=a+1}^{\infty} p_i^{(d)_i} & se (d)_0 = (d)_2 \ e \ (d)_{(d)_0} = 2, \\ p_0^{(d)_1} p_1^{(d)_1} p_2^{(d)_0+3} \prod_{i=3}^{b-1} p_i^{(d)_i} p_b^{(d)_a} \prod_{i=b+1}^{\infty} p_i^{(d)_i} & se \ (d)_0 = (d)_2 \ e \ (d)_{(d)_0} = 3, \\ p_0^{(d)_1} p_1^{(d)_1} p_2^{(d)_{[(d)_0+3]}} \prod_{i=3}^{\infty} p_i^{(d)_i} & se \ (d)_0 = (d)_2, (d)_{(d)_0} = 4 \ e \ a = b, \\ p_0^{(d)_1} p_1^{(d)_1} p_2^{(d)_0+4} \prod_{i=3}^{\infty} p_i^{(d)i} & se \ (d)_0 = (d)_2, (d)_{(d)_0} = 4 \ e \ a \neq b, \\ d & caso \ contrário, \end{cases}$$

onde $a = (d)_{[(\ )_0+1]}$ e $b = (d)_{[(d)_0+2]}$.

A idéia desta nova definição é que a memória é vista assim: o primeiro endereço (o endereço 0) tem o número do endereço com a próxima instrução, os endereços *1* e *2* contêm os números de endereços das próximas instruções dos dois programas que concorrem. A idéia do que segue é igual ao caso do computador *c* definido na Seção 1, só que se duplica o número de condições, por existirem dois programas agindo concorrentemente.

Analogamente ao caso da função *c*, na Seção *2*, definimos a função:

$con_2^*(d, 0) = d,$

$con_2^*(d, t+1) = con_2(con_2^*(d, t)).$

Assim, podemos definir a função $Config_{\mathcal{M}}^{com_2}$. Intuitivamente, pensaremos que a *n*-upla a ser computada concorrentemente estaria localizada a partir do endereço *3*, até o endereço *n+2*. A seguir, deixamos espaço para a área de trabalho, colocamos os códigos do primeiro programa, o número *0*, para indicar o fim do primeiro programa, os códigos do segundo programa e, ao final, outro zero para marcar o fim do segundo programa. De acordo com isto, temos a seguintes definições:

$$Config_{\mathcal{M}}^{com_2}(P_1, P_2, \vec{x}) = p_0^{a+1} p_1^{a+1} p_2^{b+1} p_3^{x_1} \cdots p_{n+2}^{x_n} \prod_{i=1}^{lnc(P_1)} p_{a+i}^{recod_n^1(P_1,P_2,i)} p_b^0 \prod_{i=1}^{lnc(P_2)} p_{b+i}^{recod_n^2(P_1,P_2,i)},$$

onde $a = max(\rho(P_1), \rho(P_2), n+2)$ e $b = a + lnc(P_1) + 1$,

$recod_n^1(P_1,P_2,i)$ = o *i*-ésimo código do programa *P* ao qual soma-se o $max(\rho(P_1), \rho(P_2), n) + lnca(P, i) + 3$ caso o código seja o terceiro argumento de uma instrução da forma $J(m, r, q)$,

$recod_n^2(P_1, P_2, i)$ = o *i*-ésimo código do programa *P* ao qual soma-se o $max(\rho(P_1), \rho(P_2), n) + lnc(P_1) + lnca(P_2, i) + 3$ caso o código seja o terceiro argumento de uma instrução de forma $J(m, r, q)$, ou soma-se *2* caso o código seja o primeiro argumento das instruções de forma $Z(m)$, $S(m)$ ou o segundo argumento das instruções da forma $T(m,r)$.

É claro que as funções con_2, con_2* e $Config_n^{\omega^{n_2}}$ são recursivas primitivas. Então, pela Proposição *3.5.1*, todas estas funções são computáveis.

A seguir, podemos definir quando uma função é *computável* ^{con2}. Diferentemente do caso da computabilidade não concorrente, temos duas definições possíveis: uma considerando que as computações terminam quando *algum* dos dois programas envolvidos termina e a outra quando *ambos* os programas param. De acordo com isto, daremos as duas definições a seguir.

Definição 3.11.2 A função f de aridade n é dita *computável concorrentemente a dois programas*, o que se denota por *computável*$_1$ ^{con2}, se existem números P_1 *e* P_2 (nessa ordem) tais que, para todo $\vec{x}$, cumpre-se que:

$f(\vec{x}) \simeq [con_2*(Config_n^{\omega^{n_2}}(P_1, P_2, \vec{x}), \mu t\{[\, con_2*(Config_n^{\omega^{n_2}}(P_1, P_2, \vec{x}), t)]_1 = 0$ ou $[con_2*(Config_n^{\omega^{n_2}}(P_1, P_2, \vec{x}), t)]_2 = 0\})]_3$.

Definição 3.11.3. Definimos o conceito de função *computável*$_2$ ^{con2} de maneira análoga à Definição 3.11.2, só que em lugar de "ou" tem-se "e".

Da mesma forma que no caso da computabilidade referencial e da computabilidade automodificante, é natural tentar provar que as funções computáveis$_{1,2}$^{con2} são as mesmas que as funções computáveis.

Proposicao 3.11.4 A função f, de aridade n, é computável se, e somente se, f é computável$_{1,2}$^{con2}.

Prova. Suponhamos, em primeiro lugar, que a função f é computável pelo programa Q. Para o caso da computabilidade$_1$^{con2}, sejam $P_1 = Q$ e $P_2 = J(1, 1, 1)$ (um *loop* infinito). É claro que P_1 e P_2 (nessa ordem) computam$_1$^{con2} a função f.

Para o caso da computabilidade$_2^{con2}$, sejam $P_1 = Q$ e $P_2 = 0$ (isto é, P_2 tem como única instrução uma instrução de parada). É claro que P_1 e P_2 (nessa ordem) computam$_2^{con2}$ a função f.

Suponhamos, inversamente, que os programas P_1 e P_2 computam$_1^{con2}$ a função f. Então, seja Q o seguinte programa:

$T(n, n+2)$
$\vdots$
$T(1,3)$
} Transfere a n-upla que será computada dois endereços para frente

$Z(1)$
$S(1)$
P_1 vezes $\vdots$
$S(1)$
} Coloca P_1 no endereço 1

$Z(2)$
$S(2)$
P_2 vezes $\vdots$
$S(2)$
} Coloca P_2 no endereço 2

$gn(CONFIG_n^{\omega n_2}, 1)$
$\vdots$
$gn(CONFIG_n^{\omega n_2}, ln(CONFIG_n^{\omega n_2}))$
} No endereço 1 fica o código do estado completo da memória no início da computação concorrente

$T(1, m+1)$
$T(1, m+2)$
$Z(m+3)$
$Z(m+4)$
$S(m+4)$
} Coloca 1 no endereço $m+4$ para o programa EXP

$Z(m+5)$
$S(m+5)$
$S(m+5)$
} Coloca 2 no endereço m+5 para o programa EXP

	$Z(m+6)$	Inicializa o contador de voltas do *loop* principal
loop:	$T(m+4,2)$	
	$gn(EXP, 1)$	Calcula o 0-ésimo expoente
	$\vdots$	
	$gn(EXP, ln(EXP))$	
	$J(1, m+3, sair)$	
	$T(m+2,1)$	
	$T(m+5,2)$	
	$gn(EXP, 1)$	Calcula o 1-ésimo expoente
	$\vdots$	
	$gn(EXP, ln(EXP))$	
	$J(1, m+3, saída)$	
	$T(m+2,1)$	
	$gn(Con_2, 1)$	Calcula um passo
	$\vdots$	
	$gn(Con_2, ln(Con_2))$	
	$T(1, m+2)$	
	$S(m+6)$	Calcula o número de voltas
	$J(1, 1, loop)$	
saída:	$T(m+1, 1)$	
	$T(m+6, 2)$	
	$gn(Con_2^*, 1)$	Calcula o valor
	$\vdots$	
	$gn(Con_2^*, ln(Con_2^*))$	

onde $CONFIG_n^{com_2}$, Con_2 e Con_2^* são os programas que computam as funções $Config_n^{com_2}$, con_2 e con_2^* respetivamente, EXP é o programa que computa a função $[n]_i$ e $m = max(\rho(Con_2), \rho(EXP), \rho(Con_2^*))$.

Desta forma, o programa Q simula a ação do computador con_2. Então, Q computa a função f.

Para o caso da computabilidade$_1^{con2}$, em cada passo do *loop* do programa anterior teríamos que calcular os dois expoentes, somá-los e verificar se a soma é 0. Acreditamos não ser necessário apresentar os detalhes.

3.12 GENERALIZAÇÃO

É possível generalizar a nossa definição de computabilidade para conjuntos arbitrários e não simplesmente para o domínio N, da seguinte maneira, inspirados pela tese de **Camarão de Figueiredo (1997)**. Seja X um conjunto arbitrário e F um subconjunto arbitrário (de elementos "terminais") de X. Sejam a seguinte relação e as seguintes funções:

test x

$encode_n$: $X^{n+1} \rightarrow X$,

decode: $X \rightarrow X$,

step: $X \rightarrow X$,

$step^*$: $(X,N) \rightarrow X$,

que cumprem as igualdades seguintes:

test x *se, e somente se,* $x \in F$

$decode(encode(p, x)) = x_1$

$step^*(x, 0) = x$

$step^*(x, n+1) = step(step^*(x,n))$.

Agora, definimos que f é computável se $\exists p \in X$ tal que para todo x cumpre-se que $f(x) \simeq decode\{step^*[encode(p, x), \mu i(test[step^*(encode(p, x), i)])\}$.

Tomando $X = N$, $F = \{n \in N: [n]_0 = 0\}$ e $[\ ]_0 = 0$, $Config_n$, $[\ ]_1$, c e c^*, respectivamente, por *test*, $encode_n$, *decode*, *step* e $step^*$, é trivial ver que a Definição 3.4.5 é um caso particular desta generalização.

CAPÍTULO 4

GENERALIZAÇÕES DA NOÇÃO DE FUNÇÃO COMPUTÁVEL

A forma mais corriqueira de introduzir a noção de função computável consiste em analisar a noção de *algoritmo*, apresentando alguns exemplos. Porém, às vezes, rapidamente, sem muitas explicações passa-se para 1) a noção de função e para 2) funções na estrutura dos números naturais.

Em relação a 2), é claro que existem algoritmos que não parecem ter nada a ver com os números naturais. Foi assim que os lógicos pensaram em como estender a noção de função computável para estruturas arbitrárias. Nesta direção, nas Seções 1-5 deste Capítulo apresentamos alguns conceitos de Friedman.

Em relação à 1), nas Seções 6-15 apresentamos uma teoria de Moschovakis, na qual pode ser definido com precissão o conceito de algoritmo.

4.1 CONCEITOS BÁSICOS DE FRIEDMAN

No ano de 1969 o lógico norte-americano Harvey Friedman apresentou um artigo intitulado "Algorithmic Procedures, Generalized Turing Algorithms and Elementary Recursion Theory", numa confêrencia de lógica. Este artigo foi publicado no ano de 1971 (**Friedman, 1971**) mas, segundo Shepherdson, ficou pouco conhecido, por ter sido publicado nos *Proceedings* de uma Conferência de Lógica em lugar de um periódico matemático internacional (ver **Shepherdson, 1985, p. 287**).

O último autor mencionado publicou um *comentário* sobre o artigo de Friedman, com o mesmo título, em 1985 (**Shepherdson, 1985**). É bom lembrar que Shepherdson é

um dos autores do famoso artigo de 1963 (**Shepherdson & Sturgis, 1963**) que introduziu o conceito de computabilidade com máquinas de registros, conceito que apareceu no Capítulo 1 desta Tese. Pensamos que a publicação deste comentário de Shepherdson sobre o artigo de Friedman deve-se, em parte, ao estilo um pouco apressado e despreocupado de Friedman ao escrever seu artigo, reconhecido pelo própio autor (ver **Friedman, 1971, p. 364**).

Friedman (1971, p. 362) admite que o conceito de Shepherdson e Sturgis é uma espécie de precursor dos conceitos de Friedman, no sentido de se considerar *números naturais* como conteúdos dos registros, em lugar de se considerar *símbolos* como no caso das máquinas de Turing. Com efeito, em cada registro de uma máquina de registros pode-se armazenar um número natural arbitrariamente grande e sobre tal número pode-se aplicar a função sucessor e o teste de igualdade com o zero.

De fato, Friedman introduz vários conceitos: os conceitos de procedimento algorítmico formalizado (fap), procedimento algorítmico formalizado com contagem (fapc), algoritmo generalizado de Turing (gTa) e o esquema definicional efetivo (eds). Antes de apresentar estes conceitos, introduzimos algumas definições.

Friedman usa o conceito de estrutura relacional da maneira habitual na lógica:

Definição 4.1.1 Uma *estrutura relacional* é um sistema $(D, c_0, \ldots, c_k, f_0, \ldots, f_r, R_0, \ldots, R_m)$, onde $D \neq \varnothing$, $c_0, \ldots, c_k \in D$, $f_0, \ldots, f_r$ são funções *totais* sobre D, e $R_0, \ldots, R_m$ são relações (totais) sobre D, com k, r e $m \geq 0$.

Comentário 4.1.2 A relação de igualdade pode *não* aparecer entre as relações de uma estrutura. Portanto, na teoria de Friedman não será possível reduzir as funções parciais às relações parciais e vice-versa. Isto é interessante, pois intuitivamente é questionável reduzir as relações a funções, ou vice-versa.

Definição 4.1.3 O tipo relacional de uma estrutura $(D, c_0, \ldots, c_k, f_0, \ldots, f_r, R_0, \ldots, R_m)$ é a sucessão $((k), (a_0, \ldots, a_r), (b_0, \ldots, b_m))$, onde a_i é a aridade de f_i e b_j a aridade de R_j.

Intuitivamente, o tipo de uma estrutura diz quantos elementos distinguidos existem na estrutura e quantas e de que aridades são as funções e relações da estrutura.

Antes de apresentar os conceitos de Friedman, comentamos que uma diferença importante entre o conceito de máquina de Turing e alguns dos conceitos de Friedman (faps, fapcs e gTas) e, também, do conceito de máquina de registros de Shepherdson é que, com exceção do caso das máquinas de Turing, durante a computação está fixo, de antemão, o tamanho da memória a ser usada. Por exemplo, no caso das máquinas de registros, dado um programa P, podemos calcular o número $\rho(P)$, que diz qual é o primeiro registro a partir do qual os registros ficarão vazios durante qualquer computação do programa P.

4.2 OS PROCEDIMENTOS ALGORÍTMICOS FORMALIZADOS DE FRIEDMAN

Nesta seção apresentamos dois conceitos de Friedman: o conceito de procedimento algorítmico formalizado e o conceito de procedimento algorítmico formalizado com contagem.

Definição 4.2.1 Um tipo de procedimento é uma sequência (n, e, p, q, a, b, c) onde (a, b, c) é um tipo relacional e $1 \leq n, q$ e $0 \leq e, p$.

A interpretação intuitiva de um tipo de procedimento (n, e, p, q, a, b, c) é que n corresponde ao número de argumentos (de uma função ou relação), e ao número de registros auxiliares, p ao número de registros de contagem e q ao número de instruções do procedimento.

Parece interessante comentar que no caso do conceito de Turing o número de símbolos é finito e fixado de antemão. No caso das máquinas de registros os programas agem com um número *finito* de registros.

Definição 4.2.2 Considere-se um tipo de estrutura $\alpha = (a, b, c) = ((k), (a_0, \ldots, a_r), (b_0, \ldots, b_m))$ e um tipo de procedimento (n, e, p, q, a, b, c). Os *α-símbolos* são os símbolos $d_0, \ldots, d_k, F^0_{a0}, \ldots, F^r_{ar}, A^0_{b0}, \ldots, A^m_{bm}$. Os *$(n, e, p, q, a, b, c)$-átomos* são as expressões $A^i_j(r_{e1}, \ldots, r_{ej})$ e $\neg A^i_j(r_{e1}, \ldots, r_{ej})$, onde $1 \leq e_t \leq n+1+e$.

Intuitivamente, r_i corresponde ao conteúdo do i-ésimo registro. O limite superior $n+1+e$, na definição dos átomos, indica que interessam os registros que guardam os argumentos e os registros auxiliares, mas não interessam os registros de contagem, os quais terão um uso específico.

Definição 4.2.3 As *instruções* são expressões da forma "Se E, então fazer B, depois ir à instrução i" ou da forma "Se E, então fazer B, depois terminar", onde E é da forma

a) um (n, e, p, q, a, b, c)-átomo
b) "r_i é vazio" ou "r_i não é vazio"
c) "r_i tem 0" ou "r_i não tem 0" (r_i, registro de contagem)
d) $0 = 0$

e B é da forma

a) "nada"
b) "apagar r_i"
c) "transferir o conteúdo de r_j para r_i" (onde r_i é um registro de contagem se, e somente se, r_j é de contagem)
d) "somar 1 a r_i" ou "subtrair 1 de r_i" se o conteúdo for diferente de zero, com r_i registro de contagem
e) "colocar 0 em r_i" ou "colocar d_j em r_j" (onde r_i é de contagem e r_j não é de contagem)
f) "colocar em r_i o resultado de aplicar F^j_l aos conteúdos de $(r_{t1}, \ldots, r_{taj})$ se todos os l registros são não vazios", onde r_i e os r_t não são registros de contagem.

A condição para que os registros não estejam vazios em f) provém do fato que as funções da estrutura são totais, como foi colocado na Definição 4.2.1.

Definição 4.2.4 Um *fap* é uma sucessão finita de instruções.

A *computação* de um fap de tipo *(n, e, p, q, a, b, c)* sobre uma estrutura M com domínio D e tipo *(a, b, c)* nos argumentos $\vec{x}$ é, intuitivamente, definida assim: cada x_i é colocado no registro r_i e todos os outros registros ficam vazios; depois se consideram as instruções começando com a primeira, que também é considerada como a seguinte da última; os símbolos d_i, A^i_j e $F^{\,j}_i$ referem-se, respetivamente, ao objeto constante c_i, à relação j-ária R_i e à função f^j_i de M; se algum dos registros mencionados no antecedente de uma instrução (que é um átomo) é vazio, então a instrução não se aplica e tem-se que pular para a próxima instrução; a computação termina quando se chega a "terminar".

Se depois de terminar, o registro de saída está vazio, então dizemos que o procedimento aplicado a M em $\vec{x}$ fornece - (esta notação é introduzida por Friedman). Se tem y, dizemos que fornece y. Se a terminação não ocorre, dizemos que o fap aplicado a M em $\vec{x}$ está indefinido.

Cada fap aplicado a uma estrutura define uma função parcial n-ária.

Definição 4.2.5 Sejam β um fap com tipo de procedimento *(n, e, p, q, a, b, c)* e M uma estrutura de tipo *(a, b, c)* com domínio D. Então, PFCN(β, M) é a função n-ária sobre D dada por PFCN(β, M)($\vec{x}$) = y se, e somente se, β aplicado a M em $\vec{x}$ fornece y e $y \in D$.

Definição 4.2.6 O conceito de *fap sem contagem* obtém-se quando o número de registros de contagem é zero.

4.3 AS MÁQUINAS GENERALIZADAS DE FRIEDMAN

O nome deste conceito de Friedman sugere claramente que se trata de uma generalização do conhecido conceito de máquina de Turing de 1936. Intuitivamente, a

diferença consiste em que o que se chama de *situação* no novo conceito não depende só do estado e do símbolo lido, mas também do fato de algumas das relações dadas na estrutura verificarem-se ou não. Além disto, entre os chamados *comandos*, é possível imprimir o valor de alguma das funções dadas na estrutura.

Porém, vejamos o conceito em detalhe. Em primeiro lugar, dispõe-se de um dispositivo que tem 1) uma fita infinita em ambas direções, com infinitos endereços; 2) uma cabeça que em cada momento está lendo uma célula determinada e 3) um número infinito de estados possíveis que chamamos $q_0, q_1, \ldots$ Em segundo lugar, consideramos um conjunto infinito $a_0, a_1, \ldots$ de símbolos auxiliares.

Antes de definir o conceito de máquina generalizada de Turing, definimos o conceito de situação e comando.

Definição 4.3.1 Seja uma estrutura de tipo α, domínio D e relações $R_0, \ldots, R_m$ de aridades $b_0, \ldots, b_m$. Uma *situação-α* é uma *$m+3$-upla* $(q_i, x, e_0, \ldots, e_m)$, onde

1) q_i é o estado no qual se encontra o dispositivo
2) x é o conjunto de símbolos auxiliares da célula que está sendo lida (isto é, $x = \varnothing$ ou $x = \{a_k\}$)
3) para $0 \leq j \leq m$,

$$e_j = \begin{cases} 0 & \textit{se as } b_j \textit{ células à direita contém elementos de } D \textit{ que satisfazem a relação } R_j; \\ 1 & \textit{caso contrário.} \end{cases}$$

Definição 4.3.2 Seja uma estrutura de tipo α, domínio D, funções $f_0, \ldots, f_r$ de aridades $a_0, \ldots, a_r$, respectivamente. Os *comandos-α* são as expressões seguintes:

(1) Lq_i
(2) Rq_i
(3) Bq_i
(4) xq_i

(5) SLq_i

(6) SRq_i

(7) $Fa_j{}^j q_i$,

as quais interpretam-se intuitivamente assim:

(1) mover a cabeça leitora à esquerda e colocar-se no estado q_i

(2) idem à direita

(3) apagar a entidade impressa e colocar-se em estado q_i

(4) imprimir a entidade x (x é um símbolo auxiliar ou algum d_j) e colocar-se no estado q_i

(5) intercambiar o conteúdo da célula lida com o conteúdo da célula à esquerda e colocar-se no estado q_i

(6) idem à direita

(7) apagar a entidade impressa e imprimir o elemento $f_j(y_1, \ldots, y_{aj})$ e colocar-se no estado q_i, onde $y_1, \ldots, y_{aj}$ são os elementos de D que estão nas d_j células à direita da célula lida.

Definição 4.3.3 Dada uma estrutura de tipo α, uma *instrução-α* é uma expressão da forma $(s \rightarrow t)$, onde s é uma situação-α e t um comando-α.

Comentário 4.3.4 Uma instrução-α é interpretada, intuitivamente, assim: fazer t quando se cumpre s.

Comentário 4.3.5 É interessante observar que os comandos foram definidos como expressões linguísticas, mas as situações não. Consideramos que isto é simplesmente uma falta de rigor formal de Friedman.

Definição 4.3.6 Seja α um tipo relacional. Um *algoritmo generalizado de Turing-*α (gTa) é um conjunto finito de α-instruções tais que se $(s \rightarrow t)$ e $(s \rightarrow r)$ pertencem a gTa, então $t = r$.

Comentário 4.3.7 É claro que se trata do conceito *determinístico* de algoritmo. Friedman não se interessa pelo conceito não determinístico de algoritmo.

A *computação* de um gTa de tipo α sobre uma estrutura de tipo α e dominio D no argumento $\vec{x}$ pertencente a D é, intuitivamente, definida assim: os objetos $x_1, \ldots, x_n$ são colocados em n células adjacentes da fita (todas as outras células ficam vazias). A cabeça leitora coloca-se na célula mais à esquerda e a máquina coloca-se no estado q_0; a seguir, começam a se aplicar as α-instruções do *gTa*, até chegar a uma α-situação s tal que não existe uma instrução da forma $(s \rightarrow t)$ pertencente ao *gTa*, ou existe uma tal instrução, mas t é um comando de tipo (7) que não pode ser aplicado, isto é, alguma das d_j à direita não tem elemento de D, dado que na estrutura todas as funções são totais. Nesse momento, dizemos que o gTa tem como saída o conteúdo da célula que está sendo lida, que pode ser um elemento de D, um símbolo auxiliar ou um branco.

Com estes elementos, podemos definir o conceito de função computável por um *gTa*.

Definição 4.3.8 Seja β um gTa de tipo α e M uma estrutura de tipo α com domínio D e $1 \leq n$. Então, $TFCN(n,\beta,M)$ é a função n-ária parcial sobre D, tal que $TFCN(n,\beta,M)(\vec{x}) = y$ se, e somente se, a computação de β sobre M nos argumentos $\vec{x}$ fornece y, com y pertencente a D.

4.4 OS ESQUEMAS DEFINICIONAIS EFETIVOS DE FRIEDMAN

O quarto conceito é o conceito de *esquema definicional efetivo* (eds). Intuitivamente, como o próprio Friedman diz (**Friedman, 1971, p. 363**), os eds podem

ser vistos como definições por infinitos casos, dados de maneira efetivamente numerável, onde os casos estão dados por conjunções de fórmulas atômicas e suas negações.

Definição 4.4.1 Um *tipo definicional* é uma sequência (n,Fcn, a,b,c) ou (n, Rel, a,b,c), onde $1 \leq n$ e (a,b,c) é um tipo relacional. Intuitivamente, *Fcn* corresponde às funções e *Rel* às relações.

A seguir, damos uma definição indutiva da noção de termo-α de um tipo relacional $\alpha = (n,x, (k), (a_0, \ldots , a_r),(b_0, \ldots , b_r))$.

Definição 4.4.2 Caso Base 1. As variáveis v_i, $1 \leq i \leq n$, são *termos-*α.
Caso Base 2. Os símbolos d_i, $0 \leq i \leq k$, são *termos-*α.
Caso Indutivo. Se $s_1, \ldots , s_{aj}$ são termos-α, então $F^j_{aj}(s_1, \ldots , s_{aj})$, $0 \leq j \leq r$, é um *termo-*α.

Definição 4.4.3 As *fórmulas atômicas-*α são as expressões $A^i_{bi}(t_1, \ldots , t_{bi})$ e $\neg A^i_{bi}(t_1, \ldots , t_{bi})$, onde $t_1, \ldots , t_{bi}$ são termos-α.

Agora podemos definir as condições de um tipo definicional.

Definição 4.4.4 Seja α um tipo definicional. Então, as condições-α são expressões da forma $E1 \ \& \ E2 \ \& \ \ldots \ \& \ En$, $0 \leq n$, tais que cada E_i é uma fórmula atômica-α, $Ei \neq \neg Ej$, $i, j \leq n$.

Definição 4.4.5 Duas condições-α são *incompatíveis* se, e somente se, uma das fórmulas atômicas de uma delas é a negação de uma das fórmulas atômicas da outra.

Definição 4.4.6 Seja $\alpha = (n, Fcn, a, b,c)$ um tipo definicional (de uma função). Então, as *cláusulas-*α são as expressões $(c \rightarrow t)$, onde c é uma condição-α e t é um termo-α. Seja

$\alpha = (n, Fcn, a, b,c)$ um tipo relacional (de uma relação). Então, as *cláusulas-α* são as expressões $(c \rightarrow verdadeiro)$ ou $(c \rightarrow falso)$, onde c é uma condição-α.

A interpretação das cláusulas é a natural.

Definição 4.4.7 Seja α um tipo definicional. Então, uma *definição-α* é um conjunto de cláusulas α para o qual duas cláusulas quaisquer do conjunto têm antecedentes incompatíveis.

Isto significa, é claro, que não pode ocorrer que duas cláusulas do conjunto sejam simultaneamente verdadeiras, isto é, quaisquer duas cláusulas devem ser incompatíveis entre si.

Finalmente, podemos definir o conceito de esquema definicional efetivo.

Definição 4.4.8 Seja α um tipo definicional. Então, um *esquema definicional efetivo de tipo α* é uma definição-α que é efetivamente enumerável.

Na definição acima usa-se a noção de enumerabilidade *efetiva*. Friedman diz que faz este uso no sentido da teoria de recursão ordinária (**Friedman, 1971, p. 369**). Isto significa que este conceito, diferentemente dos conceitos de fap, fapc e gTa, requer um conceito da teoria da recursão ordinária.

Dados um eds $\beta = (n, Fcn, a, b, c)$ e uma estrutura $M = (D, c_0, \ldots, c_k, f_0, \ldots, f_r, R_0, \ldots, R_m)$ de tipo (a,b,c), podemos definir uma (única) função n-ária parcial sobre D^n que chamamos $DFCN(\beta,M)$ dada por $DFCN(\beta,M)(\vec{x}) = y$ se, e somente se, existe $(c \rightarrow t)$ em β tal que c é verdadeira e t vale y quando interpretamos v_i como x_i, d_i como c_i, F^j_i como f_j e cada A^j_i como R_j.

Analogamente, podemos definir o conceito *DREL* para o caso das relações.

4.5 RELAÇÕES ENTRE OS CONCEITOS DE FRIEDMAN

Friedman estudou as relações entre os conceitos dados. Para isso, definiu um conceito de equivalência que pode servir tanto para casos de conceitos do mesmo tipo, quanto para diferentes.

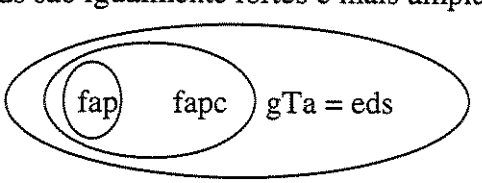
Definição 4.5.1 Sejam tanto β_1 quanto β_2 faps de tipo *(n, e, p, q, a, b, c)* ou *gTa* de tipo *(a,b,c)* ou eds de tipo *(n, Fcn, a, b, c)*. Então, β_1 *é n-Fcn equivalente a* β_2 se, e somente se, para todo M de tipo *(a, b, c)* temos que $XFCN(n, \beta_1, M) \simeq YFCN(n, \beta_2, M)$, onde X e Y são P, T ou D, e n é usado se X ou Y é T.

É trivial que para cada fap existe um fapc equivalente, que para cada fapc existe um gTa equivalente e que para cada gTa existe um eds equivalente.

Por outro lado, Friedman dá exemplos de fapc que não têm fap equivalente, e de gTa que não têm fapc equivalente.

Além disto, prova que para cada eds existe um gTa equivalente.

Concluindo, a noção de fapc é mais forte que a noção de fap e as noções de gTa e eds são igualmente fortes e mais amplas que a noção de fapc.

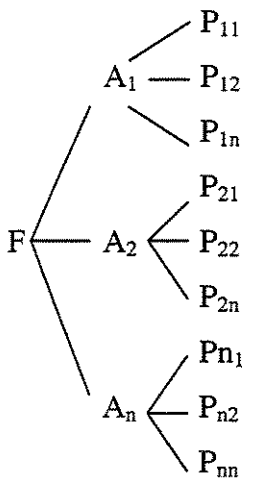


4.6 OUTRAS IDÉIAS

Vimos, então, como entender a noção de função computável para uma estrutura arbitrária de primeira ordem. Porém, dois autores, Feferman e Moschovakis apresentaram definições equivalentes para estruturas que incluem funcionais, isto é, de ordem superior. Além disto, o segundo autor apresentou uma *definição* da noção de algoritmo com uma nova versão da Tese de Turing-Church, a qual é usada também para

definir o conceito de *sentido* de uma fórmula. Por esta razão, apresentamos a seguir a teoria de Moschovakis.

Antes de darmos detalhes desta teoria, pode ser conveniente deixar claro que Moschovakis faz uma distinção explícita entre as noções de função computável, algoritmo e programa, que consideramos muito pertinente. A distinção está baseada no fato de que podemos ter diferentes algoritmos para computar a mesma função (por exemplo, no caso de dois algoritmos para ordenar listas) e também podemos ter diferentes programas para o mesmo algoritmo (por exemplo, quando programamos em diferentes linguagens formais). Pode-se representar a situação assim:



4.7 A LÓGICA DE MOSCHOVAKIS

No contexto desta Tese interessa-nos uma teoria de Moschovakis que proporciona uma definição de algoritmo com uma versão da Tese de Turing-Church para a mesma. Também interessa que Moschovakis explicitamente distingue entre programas, funções computáveis e algoritmos.

Moschovakis define uma linguagem de expressões que chama FLR (Linguagem Formal da Recursão). Existem dois tipos de expressões: os termos e os pf termos. A novidade é um operador *rec* que se pode interpretar intuitivamente como uma definição local.

Intuitivamente, a semântica *denotacional* consta de objetos de um universo para os termos, e funções parciais para os pf termos. No caso do operador *rec*, a denotação será definida por meio do conceito de ponto fixo.

Moschovakis também define uma semântica *intensional*, a qual é usada para definir o conceito de algoritmo como a *intensão* de um termo *especial* da FLR. A noção de intensão supõe dada uma estrutura de recursores e um conceito de redução com uma propriedade de normalização. Moschovakis só apresenta a versão *sintática* desta normalização.

Para a compreensão da lógica de Moschovakis estudamos diversos artigos. Aqui nos baseamos nos artigos seguintes: **(Moschovakis, 1984)**, que é o primeiro artigo de Moschovakis sobre o assunto e que apresenta numerosos exemplos; **(Moschovakis, 1989a)**, que é o artigo mais preciso, onde aparecem a sintaxe detalhada da lógica de Moschovakis, incluindo o cálculo de reduções mencionado, mas só a semântica denotacional; **(Moschovakis, 1989b)**, onde aparece a semântica intensional e **(Moschovakis, 1994)**, onde aparecem de forma menos precisa algumas vinculações com questões tradicionais, como a diferença entre sentido e denotação em Frege, e a teoria da verdade de **Kripke (1975)**. No último artigo, também aparecem alguns resultados de decidibilidade da teoria de Moschovakis.

Mesmo para artigos posteriores, o artigo de **Moschovakis (1989a)** é mencionado como fundamental.

4.8 OS TIPOS

Moschovakis baseia a sua lógica no conceito de tipo.

Definição 4.8.1 Um *conjunto de tipos básicos* é qualquer conjunto que contém a expressão "bool". *O conjunto T de tipos sobre um conjunto B de tipos básicos* se define indutivamente pelas seguintes cláusulas:

T1. Se $\overline{u_1}$, ... , $\overline{u_n}$ são tipos básicos de B, então a expressão $(\overline{u_1}, \ldots, \overline{u_n})$ é um *tipo individual* de T; $( )$ é um caso particular;

T2. Se $\overline{u}$ é um tipo individual e $\overline{w}$ é um tipo básico, então a expressão $(\overline{u} \rightharpoonup \overline{w})$ é um *tipo pf* de *T*;

T3. Se $\overline{x_1}, \ldots, \overline{x_n}$ são tipos básicos ou *pf*, então a expressão $(\overline{x_1}, \ldots, \overline{x_n})$ é um *tipo produto* de *T*.

T4. Se *x* é um tipo produto e *w* é um tipo básico, então a expressão $(\overline{x} \rightharpoonup \overline{w})$ é um *tipo de função* de *T*.

Observar que a notação $\rightharpoonup$ se justifica pelo fato que na semântica tomam-se funções parciais e a notação "bool" se justifica pelo fato que na semântica a seguir, "bool" é interpretado como um conjunto binário.

4.9 A SEMÂNTICA DOS TIPOS

A seguir podem-se definir os objetos dos distintos tipos. Simultaneamente, Moschovakis define uma ordem parcial entre os objetos, a qual será útil para definir os objetos de tipo função.

Definição 4.9.1 Um *universo* sobre um conjunto *B* de tipos básicos é uma família de conjuntos $U=\{U(i);\ i \in B\}$ tal que $U(bool)=\{0,1\}$. Os *objetos básicos* de *U* são os elementos dos conjuntos de *U*.

A partir destes objetos podemos definir os objetos de tipos restantes de maneira indutiva:

O1. Um *indivíduo* de tipo $u = (\overline{u_1}, \ldots, \overline{u_n})$, onde os u_i são tipos básicos, é qualquer elemento do produto cartesiano $U = U(\overline{u_1}) \times \ldots \times U(\overline{u_n})$. Vamos considerar cada espaço *U* com a ordem parcial definida por =. Está incluído *I*, o produto cartesiano

sem fatores, que tem como único elemento a seqüência vazia e satisfaz, para cada X, $X \times I = I \times X = X$.

O2. Um *objeto de tipo pf* $(\overline{u} \rightharpoonup \overline{w})$ é qualquer função parcial no espaço $\{p:U \rightharpoonup W\}$, onde U e W são os espaços individual e básico, respectivamente, de tipo $\overline{u}$ e $\overline{w}$. Vamos considerar cada espaço de funções parciais com a ordem definida por

$p \leq q$ se, e somente se, para todo $u\ [p(u)\downarrow \Rightarrow q(u) \simeq p(u)]$.

O3. Um *ponto* de tipo produto $(\overline{x_1}, \ldots, \overline{x_n})$ é qualquer elemento do espaço produto $X = X_1 \times \ldots \times X_n$, onde os X_i são conjuntos básicos ou pf de tipo $\overline{x_i}$.

Todo espaço produto vem ordenado por $(x_1, \ldots, x_n) \leq (x'_1, \ldots, x'_n)$ se, e somente se, $x_1 \leq x'_1 \ \& \ \ldots \ \& \ x_n \leq x'_n$.

O4 Um *objeto de tipo função* $(\overline{x} \rightharpoonup \overline{w})$ é um funcional $f:X \rightharpoonup W$ do espaço produto de tipo $\overline{x}$ no conjunto básico de tipo $\overline{w}$ que é monótono relativamente à ordem parcial em X:

$f[x] = w \ \& \ x \leq y \Rightarrow f[y] = w$.

Esta semântica encontra-se em **Moschovakis (1989a, p. 1219)**.

4.10 A LINGUAGEM FORMAL DA RECURSÃO

Definimos a linguagem $FLR(\tau)$ para uma assinatura τ.

Definição 4.10.1 Uma *assinatura* é uma tripla (B, S, d), onde B é um conjunto de tipos básicos que inclui pelo menos o tipo bool, S é um conjunto de constantes para funcionais e d é uma função que associa, a cada $f \in S$, um tipo de função $d(f)$ sobre B.

Em primeiro lugar, apresentamos o *alfabeto* de $FLR(\tau)$, que consta de:

Variáveis: lista infinita de variáveis básicas de tipo i, para cada $i \in B$;
Lista infinita de variáveis *pf* de tipo $(\overline{u} \rightharpoonup \overline{w})$, para cada tipo pf $(\overline{u} \rightharpoonup \overline{w})$.
Constantes: são as constantes que pertencem a S; $\underline{1}$ e $\underline{0}$ são constantes de tipo $(( ) \rightharpoonup bool)$; $cond_w$ é uma constante de tipo $[bool, (( ) \rightharpoonup \overline{w}), (( ) \rightharpoonup \overline{w})] \rightharpoonup \overline{w}$.

Intuitivamente, $\underline{1}$ e $\underline{0}$ correspondem à verdade e a falsidade, respectivamente, e $cond_w$ corresponde ao condicional.

Notação. Por abuso de notação, chamamos de *variáveis individuais* de tipo $(u^1, \ldots, u^m)$ às expressões $u_1, \ldots, u_m$, onde as u_m são variáveis básicas *distintas*. Está incluída a expressão vazia $\varnothing$ que tem tipo $()$.

A seguir, definimos por indução o conceito de *expressão*, que corresponde à noção habitual de fórmula de uma lógica. Haverá dois tipos de expressões: os termos e os termos *pf*.

Definição 4.10.2

T1. Toda variável básica é um *termo*. Toda variável pf é um *termo pf*. Os tipos destes termos são iguais aos tipos das variáveis.

T2. Se p é uma variável pf de tipo $((t_1, \ldots, t_n) \rightharpoonup w)$ e $t_1, \ldots, t_n$ são termos de tipo $\overline{t_1}, \ldots, \overline{t_n}$, então $p(t_1, \ldots, t_n)$ é um *termo* de tipo $\overline{w}$. Inclui-se o termo $p( )$ de tipo $\overline{w}$, se p for de tipo $(() \rightharpoonup \overline{w})$.

T3. Se t é um termo de tipo $\overline{w}$ e $u_1, \ldots, u_n$ são variáveis básicas distintas, então $\lambda(u)t$ é um *termo pf* de tipo $(\overline{u_1}, \ldots, \overline{u_n} \rightharpoonup \overline{w})$. Inclui-se o termo $\lambda( )t$, de tipo $(( ) \rightharpoonup \overline{w})$.

T4. Se $t_1, \ldots, t_n$ são expresões de tipo $\overline{t_i}$ e f é uma constante de função de tipo $((\overline{t_1}, \ldots, \overline{t_n}) \rightharpoonup \overline{w})$, então $f[t_1, \ldots, t_n]$ é um *termo* de tipo $\overline{w}$. Inclui-se o termo $f[ ]$, se f for de

tipo $((\,) \rightharpoonup \overline{w})$. Abreviamos $f = f[\,]$. Por exemplo, $\underline{1} = \underline{1}[\,]$. No caso de $cond_w$, usamos a notação "if s then t_1 else t_2"em lugar de $cond_w[s, \lambda(\,)t_1, \lambda(\,)t_2]$.

Esta forma de definir uma função é corriqueira em matemática. Pode-se lembrar, por exemplo, das definições da função valor absoluto ou da função fatorial:

$$|x| = \begin{cases} x & se\ x \geq 0; \\ -x & se\ x < 0, \end{cases}$$

$$n! = \begin{cases} 1 & se\ n = 0; \\ n(n-1)! & se\ n > 0. \end{cases}$$

Na lógica anterior à década de sessenta não existia uma maneira direta de formalizar este tipo de equações. McCarthy (1960) formalizou estas equações da seguinte maneira:

$|x| = \text{if } x \geq 0 \text{ then } x \text{ else } -x;$
$n! = \text{if } n = 0 \text{ then } 1 \text{ else } n(n\text{-}1)!$

Nesta equação a expressão *if... then ... else ...* fornece um *termo*, não uma fórmula. O argumento que segue *if* é uma expressão que pode ser verdadeira ou falsa. Por outro lado, os outros argumentos podem ser expressões que fazem referência a objetos arbitrários. É interessante explicitar que esse tipo de fórmula preenche um buraco nas formalizações habituais de primeira ordem, pois 1) os símbolos lógicos como $\wedge$ e $\vee$ podem ser vistos como funções de verdade/falsidade, 2) as fórmulas da forma $Rx_1 \ldots x_n$ podem ser vistas como funções que dão valores de verdade a objetos arbitrários e 3) os termos da forma $f(x)$ como funções que associam objetos arbitrários a objetos arbitrários. A quarta possibilidade é o caso de associar objetos arbitrários aos valores de verdade, que é o caso das fórmulas da forma *if... then... else*.

É interessante acrescentar que a partir do *if... then... else* podem-se definir os símbolos lógicos habituais $\wedge$, $\vee$, $\rightarrow$ e $\neg$ da seguinte maneira:

$A \wedge B = if\ A\ then\ B\ else\ \bot$

$A \vee B = if\ A\ then\ T\ else\ B$

$A \rightarrow B = if\ A\ then\ B\ else\ T$

$\neg A = if\ A\ then\ \bot\ else\ T.$

Esta forma de fazer lógica conduz a uma linguagem de termos e não de fórmulas, e dá primazia ao conceito de função sobre o conceito de *relação*. De um ponto de vista histórico é pertinente dizer que este já era o ponto de vista de Frege, o qual na sua ontologia privilegiava as categorias de função e objeto, e que também aparece no cálculo-λ de Church.

Definição 4.10.3 Uma expressão é *explícita* se, e somente se, pode ser construída usando apenas T1-T4. Para definir as *expressões recursivas* acrescentamos a cláusula seguinte que define os *termos recursivos*:

T5. Se $u_1, \ldots, u_n$ são variáveis individuais, $p_1, \ldots, p_n$ são variáveis *pf diferentes* de tipos $(\overline{u_1} \rightarrow \overline{w_1}), \ldots, (\overline{u_n} \rightarrow \overline{w_n})$ e $t_0, t_1, \ldots, t_n$ são termos de tipos $\overline{w}, \overline{w_1}, \ldots, \overline{w_n}$, então:

$$rec(u_1 p_1, \ldots, u_n, p_n)[\, t_0, t_1, \ldots, t_n]$$

é um termo de tipo $\overline{w}$.

Inclui-se o caso $n = 0$. Portanto, *rec ( )[t]*, que abreviamos *rec ( )t*, é um termo. As sequências $(u_1, \ldots, u_n)$ podem ser vazias. Portanto, rec*(p, x, q)[t_0, t_1, t_2]* é um termo.

Parece conveniente fornecer alguma intuição para T5. Moschovakis dá a notação alternativa:

$$t_0\ onde\ \{p_1(u_1) \simeq t_1, \ldots, p_n(u_n) \simeq t_n\}.$$

A interpretação intuitiva mais adequada parece ser considerar as cláusulas da forma $p_i(u_i) \simeq t_i$ como *definições locais* das p_i que aparecem em t_0. Mas não encontramos este comentário em nenhum dos artigos de Moschovakis. Este operador parece já ter história na área de computação teórica (ver, e.g. **Landin, 1964**). É claro que é um operador de segunda ordem, com variáveis ligadas de funções parciais.

A seguir, Moschovakis define os conceitos habituais de variáveis livres e ligadas e de substituições, adaptando-os à sua teoria (ver **Moschovakis, 1989a, p. 1221-2**).

4.11 A SEMÂNTICA DENOTACIONAL

Moschovakis considera várias semânticas, mas aqui basta apresentar a que ele considera mais natural.

Em primeiro lugar, definimos o conceito de estrutura funcional.

Definição 4.11.1 Uma *estrutura funcional* de assinatura $\tau = (B, S, d)$ é um par $A = (U, F)$, onde U é um universo sobre B e $F = \{F(f): f \in S\}$ é uma família de funcionais *monótonos* sobre U, tal que $d(f)$ é o tipo de $F(f)$.

A seguir, é necessário definir a noção habitual de interpretação numa lógica.

Definição 4.11.2 Uma *atribuição* numa estrutura funcional A é qualquer função *parcial* definida para *todas* as variáveis pf, mas só em *algumas* variáveis básicas de *FLR*, que associa (quando está definida) a cada variável x um objeto em A do mesmo tipo que x.

Baseados nesta definição, podemos definir uma função *val* para todas as expressões de *FLR*:

val: Expressões $\times$ *Atribuição* $\rightharpoonup$ *Objetos de A.*

Esta definição é indutiva e baseada na definição de expressão. Só interessa a cláusula que corresponde aos termos gerados com o operador *rec*:

Se t é $rec(u_1,p_1, \ldots, u_n, p_n)[\, t_0, t_1, \ldots, t_n]$, então seja

$$h_i[u_i, p_1, \ldots, p_n] \simeq val(t_i, \pi\,[v_i/u_i, p_1/p_1, \ldots, p_n/p_n]),$$

para $i = 0,1, \ldots, n$, onde $\pi[\ldots]$ é obtido a partir de π trocando os valores só nas variáveis $u_i,p_1, \ldots, p_n$. Pode-se provar por indução que cada funcional h_i é monótono. Então, por métodos conhecidos da teoria de conjuntos, temos que as equações

$$h_i[u_i, p_1, \ldots, p_n] \simeq p_i(u_i)\ (i= 0, \ldots, n)$$

têm uma sucessão $p_0, p_1, \ldots, p_n$ de menores soluções simultâneas. Então, definimos

$$val(t, \pi) \simeq p_0(\,)$$
$$\simeq val(t_0, \pi\,[p_1/p_1, \ldots, p_n/p_n]).$$

Definição 4.11.3 Seja x uma sucessão de variáveis básicas *pf* que inclui todas as variáveis livres de um termo t. Então, a *denotação de t na estrutura funcional A relativa a x* é o funcional

$den(x,t)$: $X \rightharpoonup W$, $den\ (x,\ t)\ (x) \simeq val(t, \pi_x)$, onde X é o espaço produto com o tipo de x, W é o conjunto básico com o tipo de t e, para cada $x = x_1, \ldots, x_n$, π_x é *qualquer* atribuição que associa x_i a x_i, para $i = 1, \ldots, n$.

Definição 4.11.4 Dois termos t e s são *denotacionalmente equivalentes*, o que é denotado por $t \sim s$, se, e somente se, para toda estrutura A e para toda atribuição π em A, $val(t, \pi) \simeq val(s, \pi)$.

Interessam os funcionais recursivos de uma estrutura. Para definí-los, introduzimos os termos especiais.

Definição 4.11.5 Um termo t é *especial* se é da forma $p(t_1, \ldots, t_n)$, ou $f(t_1, \ldots, t_n)$, ou $rec(u_1,p_1, \ldots, u_n, p_n)[s_0, s_1, \ldots, s_n]$, onde $s_0, s_1, \ldots, s_n$ são todos especiais.

Definição 4.11.6 Um funcional $f:X \rightharpoonup W$ no universo de uma estrutura funcional A é *A-recursivo* se f é a denotação de um termo especial relativo a alguma seqüência de variáveis.

4.12 CONGRUÊNCIA, CÁLCULO DE REDUÇÕES E FORMA NORMAL

Antes de definir a intensão de um termo é preciso dispor de algumas noções, por exemplo, a noção de expressão imediata, a qual requer uma relação de congruência entre termos.

A linguagem *FLR* tem os operadores λ e *rec* que ligam variáveis. Em geral, nas linguagens que contêm operadores que ligam variáveis costuma-se não distinguir entre as expressões que só diferem por suas variáveis ligadas. Por exemplo, na lógica de primeira ordem são congruentes $\forall xPx$ e $\forall yPy$. Desta forma, define-se uma relação de congruência para *FLR*, a qual será um pouco mais complicada que a habitual, dado que se deseja considerar também a relação entre as variáveis pf e o operador λ, ignorar a operação vácua *rec( )* e identificar termos recursivos que só diferem na ordem de suas partes distintas da primeira parte.

Definição 4.12.1 A *congruência* é a menor relação de equivalência $=_c$ no conjunto de expressões que satisfazem as seguintes cláusulas:

C1. Para cada variável pf p, $p =_c \lambda(u)p(u)$, para toda sucessão u de variáveis;

C2. Se $t_1 =_c t'_1,..., t_n =_c t'_n$, então $p(t_1, \ldots, t_n) =_c p(t'_1,...,t'_n)$;

C3. Se $z^1, \ldots, z^m$ são variáveis que não ocorrem em t nem em t', então se $subst(t, u'/z', \ldots, u_m/z_m) =_c subst(t', v'/z', \ldots, v^m/z^m)$, então $\lambda(u_1, \ldots, u_m)\, t =_c \lambda(v^1, \ldots, v^m)\, t'$;

C4. Se $t_1 =_c t'_1, \ldots, t_n =_c t'_n$, então $f(t_1, \ldots, t_n) =_c f(t'_1, \ldots, t'_n)$;

C5a. $rec(\ )[t] =_c t$;

C5b. $rec(u_1,p_1, \ldots, u_n,p_n)[\ t_0, t_1, \ldots, t_n] =_c rec(v_1, q_1, \ldots, v_n, q_n)[\ t'_0, t'_1, \ldots, t'_n]$ se existe uma permutação σ sobre $\{0, \ldots, n\}$ com inversa ρ tal que $\sigma(0) = 0$ e tal que:

(a) para $i = 1, \ldots, n$ as variáveis pf $p_{\sigma(i)}$ e q_i têm o mesmo tipo;

e

(b) Se $r_1, \ldots, r_n$ são variáveis pf e $z_1, \ldots, z_n$ são seqüências de variáveis básicas que não aparecem em $t_0, \ldots, t_n$ nem em $t'_0, \ldots, t'_n$, então o termo $subst(t'_{\rho(i)}, v_{\rho(i)}/z_{\rho(i)}, q_1/r_1, \ldots, q_n/r_n) =_c subst(t_i, u_i/z_{\rho(i)}, p_{\sigma(1)}/r_1, \ldots, p_{\sigma(n)}/r_n)$.

Para entender a necessidade das próximas noções, pode ser muito útil considerar o seguinte exemplo: para definir no "contexto vazio" o algoritmo definido por um termo recursivo $rec(u_1,p_1, \ldots, u_n,p_n)[\ t_0, t_1, \ldots, t_n]$ será necessário ter os algoritmos definidos por cada t_i no "contexto" $\{u_i, p_1, \ldots, p_n\}$.

Definição 4.12.2 Um *contexto* é um conjunto de variáveis básicas ou *pf*.

Definição 4.12.3 Seja E um contexto. Uma expressão de *FLR* é *imediata em E* se é congruente a uma variável básica $p(v_1, \ldots, v_n)$, com $p, v_1, \ldots, v_n \in E$ ou $\lambda(u^1, \ldots, u^n)\, p(v_1, \ldots, v_n)$, com $p, v_1, \ldots, v_n \in E \cup \{u^1, \ldots, u^n\}$.

Moschovakis define indutivamente uma relação de *redução* entre termos que depende do contexto E e que se denota por $s \rightarrow_E t$. **(Moschovakis, 1989a, p. 1230)**

Definição 4.12.4 Seja t um termo e E um contexto. Então, t é *irredutível* em E se, para todo s, se $t \rightarrow_E s$, então $s =_c t$.

Definição 4.12.5 Uma expressão t é *simplificada* se a operação vácua *rec ( )* não ocorre em t ou t é da forma *rec ( )s*, e *rec( )* não ocorre em s.

Seja t um termo e E um contexto. A *forma normal nf(t, E)* é um termo da forma $rec(u_1,p_1, \ldots, u_n, p_n)[\, t_0, t_1, \ldots, t_n]$, simplificado, irredutível em E e tal que $t \rightarrow_E nf(t, E)$. A definição por indução, que dá a existência da forma normal, encontra-se na p. 1233 de **(Moschovakis, 1989a)**. Prova-se a unicidade a menos de congruência (*idem*, p. 1236).

4.13 A SEMÂNTICA INTENSIONAL

Definição 4.13.1 Um *recursor* sobre um universo U é uma n-upla de funcionais $f = [f_0, \ldots, f_n]: X \rightarrow W$, para o qual existem espaços individuais $U_0=I$, $W_0=W$ e $U_1, W_1, \ldots, U_n, W_n$ tais que

$$f_i: U_i \times P(U_1,W_1) \times \ldots \times P(U_n,W_n) \times X \rightarrow W_i.$$

Definição 4.13.2 A *denotação* de um recursor $f = [f_0, \ldots, f_n]: X \rightarrow W$ é o funcional $\mathbf{f}: X \rightarrow W$ definido por

$$\mathbf{f}(x) = f_0(p_{1,x}, \ldots, p_{n,x}, x),$$

onde para cada x, $p_{i,x}$ são os pontos fixos simultâneos das equações

$$p_{i,x}(u_i) = f_i(u_i, p_1, \ldots, p_n, x), \ (1 \leq i \leq n).$$

Definição 4.13.3 Dois recursores $f = [f_0, \ldots, f_n]$ e $g[g_0, \ldots, g_n]: X \rightarrow W$ são *fortemente equivalentes* se, e somente se, $n = m$ e existe uma permutação σ sobre $\{0,1, \ldots, n\}$ com

inversa ρ, tais que $\sigma(0) = 0$ e, para todo $i = 0, \ldots, n$ e todo $u_i, x, p_1, \ldots, p_n$, cumpre-se que

$$f_i(u_i p_1, \ldots, p_n, x) \simeq g_{\rho(i)}(u_i p_{\sigma(1)}, \ldots, p_{\sigma(n)}, x).$$

Definição 4.13.4 Uma *estrutura de recursores de assinatura* $\tau = (B, S, d)$ é um par $A = (U, F)$, onde U é um universo sobre B e $F = \{F(f) \mid f \in S\}$ é uma família de recursores, tal que, para cada $f \in S$, o tipo da *denotação* $F(f)$ é $d(f)$.

Definição 4.13.5 Seja A uma estrutura de recursores de assinatura τ. Então, a *expansão funcional* A^o é a estrutura que resulta de substituir cada recursor $f = [f_0, \ldots, f_n]$ de A pelos funcionais $f_0, f_1, \ldots, f_n$. A assinatura τ^o de A^o resulta de uma expansão da assinatura de A: mantemos a constante f que nomeava o recursor f para nomear f_0 e introduzimos novos símbolos $f_1, \ldots, f_n$ para os outros funcionais de f.

Definição 4.13.6 Seja A uma estrutura de assinatura τ. Com cada termo ou λ-termo t de $FLR(\tau)$ e cada contexto E define-se, indutivamente, a *tradução tr (t, E) de t no contexto E*, uma expressão da mesma categoria que t em $FLR(\tau^o)$.

Como é de se esperar, esta definição é trivial, com exceção do caso dos termos que são da forma $f(t_1, \ldots, t_n)$. Podem-se ver os detalhes em **Moschovakis, 1989b, p. 220**.

Definição 4.13.7 Sejam A uma estrutura de recursores, A^o a sua expansão funcional e t um termo. Suponhamos que x é uma lista de variáveis que inclui todas as variáveis livres de t e seja E um contexto. Seja $nf(tr(t, E), E) = rec(u_1, p_1, \ldots, u_n, p_n)[t_0, t_1, \ldots, t_n]$ a forma normal da tradução de t no contexto E. Então, definimos a *intensão de t no contexto E segundo a estrutura A*, da seguinte forma:

$int(\aleph, E)\ t = [den(\aleph, t_0), den(\aleph, t_1), \ldots, den(\aleph, t_n)]$.

Definição 4.13.8 Um *algoritmo* de uma estrutura de recursores A é qualquer recursor $f:X \rightarrow W$ sobre o universo de A tal que, para alguma lista de variáveis x com o tipo de X e para algum termo especial $t(\aleph)$ tal que suas variáveis livres estão contidas em x,

$f = int\ (\aleph, \varnothing)\ t(\aleph)$.

Em relação à definição de algoritmo, parece natural que exista um interesse na comunidade lógica por esta noção. Mas, também, existe um interesse *legal*, pois nos Estados Unidos algumas pessoas tentaram *patentear* algoritmos, por exemplo, um algoritmo de multiplicação rápida e, para que isso seja de utilidade, seria conveniente ter um procedimento para decidir se dados dois algoritmos eles são iguais ou não. Os resultados de Moschovakis podem ser úteis neste sentido (ver **Moschovakis, 1994, p. 226**).

4.14 A TESE DE MOSCHOVAKIS

A definição de algoritmo que acabamos de dar sugere uma tese análoga à tese de Church: um algoritmo no sentido intuitivo corresponde a um algoritmo no sentido da Definição 4.13.8. A Tese de Turing-Church corresponde ao conceito de *função* computável, mas a Tese de Moschovakis corresponde à noção de *algoritmo*. Neste sentido, é mais abrangente que a tradicional Tese de Turing-Church.

Um argumento a favor da Tese de Turing-Church usa o primeiro Teorema da Recursão (**Kleene, 1952, #66**). Analogamente, Moschovakis apresenta um Teorema Intensional da Recursão (**Moschovakis, 1989b**) para argumentar a favor de sua tese. O teorema diz que se f é um algoritmo de uma estrutura A de recursores, então a expansão (A, f) tem os mesmos algoritmos que A. Isto significa que não será fácil escapar do conjunto de algoritmos associados a uma estrutura por alguma forma de definição explícita ou implícita do tipo da teoria de Moschovakis.

Como último comentário, acrescentamos que a opção de definir a noção de algoritmo como a classe de equivalência de programas que computam a mesma função obviamente, não está disponível, pois essas classes são demasiado povoadas, pois incluem programas que correspondem a algoritmos diferentes, no sentido de Moschovakis.

4.15 COMPARAÇÃO COM O CONCEITO HABITUAL DE COMPUTABILIDADE

Moschovakis considera óbvio que todos os algoritmos definidos em N pelos sistemas de equações de Herbrand-Gödel podem ser obtidos numa estrutura de recursores da sua teoria (ver **Moschovakis, 1989b**).

Por outro lado, parece que não é possível definir uma máquina de Turing que represente com fidelidade, por exemplo, o algoritmo sugerido pela expressão seguinte:

Quadrado(x) = produto (x,x), onde produto (u, v) = if zero (v) then 0 else soma (produto(u,pred(v),u)).

Moschovakis pensa que qualquer máquina de Turing ficará comprometida com os detalhes da implementação da recursão, em lugar de simplesmente representar tal algoritmo (paradigmas iterativo e recursivo, respectivamente).

Moschovakis gosta de apresentar a sua teoria como uma forma de superar o que às vezes chama-se de estilo von Neumann de programação (ver **Backus, 1978**). Talvez esta seja, de fato, uma das questões mais importantes da informática contemporânea.

Fechamos esta tese com um comentário de Moschovakis a nós enviado, em correspondência particular, sobre o ponto de vista de Soare, comentado na Seção 1 do Capítulo 1:

> "Robert Soare tem argumentado que aquilo que os teóricos da recursão estudam é a classe das funções computáveis em números naturais; bem, esta é exatamente a

classe das funções recursivas (por teoremas clássicos de Turing e Kleene), e, portanto, o ponto de vista de Soare não é filosófico ou fundacional, mas trata-se de um conselho político: ele acredita que os teóricos da recursão seriam melhor apreciados se consistentemente denominassem as funções que estudam de "computáveis" ao invés de "recursivas". Vindo de um país que praticamente se ridicularizou no cenário internacional ao rejeitar chamar à ex República Iugoslávia de Macedônia, como os cidadãos daquele país querem chamá-la, não estou na situação de argumentar que o ponto de vista de Bob é ridículo, e poderia ser que ele fosse ridículo e estivesse certo.

A meu respeito, não estudo simplesmente as funções recursivas (ou computáveis) em números naturais, mas "recursivo" como um processo fundamental de definição implícita em estruturas abstratas de primeira ordem (e algumas de segunda ordem). As funções definidas recursivamente nestas estruturas abstratas nem sempre podem ser identificadas com as funções que são computáveis por máquinas de algum tipo razoável; e mesmo quando podem, suas definições recursivas têm aspectos "intensionais" interessantes que não estão refletidos no simples fato que, ao final de contas, as funções definidas são "computáveis". (Por exemplo, eu sustento que as definições recursivas expressam diretamente "algoritmos", e "mais algoritmos" que aqueles expressados por máquinas). Portanto, meu uso do termo "recursão" é necessário, porque o que eu faço com recursões nem sempre é "equivalente" a algo que pode ser feito com máquinas, e, portanto, não tenho o luxo de dispor de dois termos sinônimos." **(Moschovakis, 2000)**

CONSIDERAÇÕES FINAIS

As contribuições originais desta Tese são: 1) a análise do *status* da Tese de Turing-Church; 2) a análise detalhada de uma objeção de Péter, em particular a Proposição 2.6.2, os vínculos dessa objeção com o Princípio de Markov e o Princípio do Número Mínimo, e as conclusões constantes na Seção 2.11; 3) a teoria desenvolvida no Capítulo 3, em particular a distinção entre a computabilidade padrão, auto-referencial e automodificante, a prova da não variação da complexidade dessas diferentes noções, e a definição do conceito de vírus.

Entre os possíveis desenvolvimentos futuros estão: 1) uma avaliação da definição de função computável parcial de Troelstra e van Dalen, questão de interesse didático-pedagógico e vinculada com a questão do Capítulo 2; 2) conseqüências didáticas das questões analisadas na Tese; 3) avaliação da conveniência de um maior desenvolvimento da teoria esboçada no Capítulo 3; 4) aprofundamento de questões relativas ao construtivismo; 5) uma avaliação mais pormenorizada da teoria de Moschovakis apresentada no Capítulo 4.

ANEXO

CITAÇÕES DO CAPÍTULO 1

"I am extremely careful with the choice of those (*i. e.* new notions), as I take the position that the development and propagation in no small degree depends on a fortunate and properly fitting terminology." **(Soare, 1996, p. 313)**

"*Computability theory* might be a better name for recursion theory, since it is the mathematical study of what is computable." **(Philips, 1992, p.80)**

"...it is maintained that the notion of an effectively calculable function of positive integers should be identified with that of a recursive function, since other plausible definitions of effective calculability turn out to yield notions which are either equivalent to or weaker than recursiveness." **(Church, 1935, p. 333)**

"The resulting notion is of especial interest since the intuitive notion of a "constructive" or "effectively calculable" function of natural numbers can be identified with it very satisfactorily." **(Kleene, 1936c, p. 544)**

"Turing´s computability is intrinsically persuasive in the sense that the ideas embodied in it directly support the thesis that the functions encompassed are all for which there are algorithms; lambda definability is not intrinsically persuasive (the thesis using it was supported not by the concept itself but rather by results established about it) and general recursiveness scarcely so (its author Gödel being at the time not al all persuaded). **(Kleene, 1981b, p. 49)**

"Church, Kleene, and I each thought that general recursivity seemed to embody the idea of effective calculability, and so each wished to show it equivalent to λ-definability." **(Rosser, 1984, p 345)**

"I myself, perhaps unduly influenced by rather chilly receptions from audiences around 1933-35 to disquisitions on λ-definability, choose, after general recursiveness had appeared, to put my work in that format." **(Kleene, 1981a, p. 62)**

"...it is maintained that the notion of an effectively calculable function of positive integers should be identified with that of a recursive function..." **(Church, 1935, p. 333)**.

"[TC] states the identity of two notions..." **(Kalmár, 1959, p. 72)**

"The identity known as Church´s Thesis..." **(Folina, 1998, p. 302)**

"...the selection of a formal definition to correspond to an intuitive notion." **(Church, 1936, p. 356, Davis, 1965, p. 100)**

"...our original notion of effective calculability of a function is a somewhat vague intuitive one, ..." **(Kleene, 1952, p. 317)**

"All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically." **(Turing, 1937, p. 249** e **Davis, 1965, p. 135)**

"There are pre-mathematical concepts which must remain pre-mathematical ones, for they cannot permit any restriction imposed by an exact mathematical definition. Among these belong, I am convinced, such concepts as that of effective calculability..." **(Kalmár, 1959, p. 79)**

"Many classes of number-theoretic functions have been delimited by formal definitions: the arithmetic functions, the Fibonacci functions, the recursive functions, etc. The term "computable", however, was applied to functions pre-formally – that is, in advance of any formal definition." **(Shapiro, 1981, p. 353)**

"The classical Church's Thesis identifies the intuitive notion of *computable function* on the integers with the precise, mathematical notion of *recursive function*. It is a serious mistake to understand it as just a definition of computability. **(Moschovakis, 1984, p. 354)**

"The concepts and assumptions that support the notion of partial-recursive function are, in an essential way, no less vague and imprecise that the notion of effective computable function; the former are just more familiar and are part of a respectable theory with connections to other parts of logic and mathematics." **(Mendelson, 1990, p. 232)**

"This definition is thought to be justified by the considerations which follow, so far as positive justification can ever be obtained for the selection of a formal definition to correspond to an intuitive notion". **(Church, 1936, p. 356** e **Davis, 1965, p. 100)**

"Since our original notion of effective calculability of a function is a somewhat vague intuitive one, (CT) cannot be proved..."**(Kleene, 1952, p. 317)**

"Church´s Thesis is not a mathematical theorem which can be proved or disproved in the exact mathematical sense, for it states the identity of two notions only one of which is mathematically defined while the other is used by mathematicians without exact definition." **(Kalmár, 1959, p. 72)**

"In each case, we can describe procedures that will compute the new funcions." **(Mendelson, 1990, p. 233)**

"...the view which tells us not to take seriously the traditional question whether an axiomatic characterization of an informal concept is *correct* because (on this view) informal concepts are by their nature too imprecise for this to have an answer." **(Kreisel, 1967, p. 176)**

"The problem is that Kreisel simply identifies, without argument, the intuitive notion with the model-theoretic notion of truth in all structures." **(Etchemendy, 1999, p. 147)**

"Up to about 1650 no one believed that the length of a curve could equal exactly the length of a line. In fact, in the second book of *La Geometrie*, Descartes says the relation between curved lines and straight lines is not nor ever can be known." **(Kline, 1972, p. 354)**.

"For the concept of computability, however, although it is merely a special kind of demonstrability or definability, the situation is different. By a kind of miracle it is not necessary to distinguish orders, and the diagonal procedure does not lead outside the defined notion." **(Gödel, 1990, p. 150)**

"When Church proposed this thesis, I sat down to disprove it by diagonalizing out of the class of the lambda-definable functions. But, quickly realizing that the diagonalization cannot be done effectively, I became overnight a supporter of the thesis." **(Kleene, 1981a, p. 59)**

"I myself agree with Kalmár´s conviction that effective computability is one of those notions the definition of which can never be considered complete in the course of the development of mathematics." **(Péter, 1981, p. 142)**

"Computability is a property related to either human abilities or mechanical devices, both of which are at least prima facie non-mathematical." **(Shapiro, 1981, p. 353)**

CITAÇÕES DO CAPÍTULO 2

"There are some finitists or intuitionists who might deny that all general recursive functions are computable, or even assert that the class of general recursive functions is not well-defined. However, by speaking of partial recursive functions we avoid this difference of opinion. For there is surely no doubt that the routines given here and elsewhere will actually compute the value of a given recursive function for a given argument at which the function is defined, and will go on computing forever if the function is not defined at that argument. Of course, there may now be a difference of opinion as to whether a given partial recursive function is general recursive, i.e. defined for all arguments; in fact, the question of whether such a function is defined for one particular argument can be as difficult as the Fermat conjecture. But disagreement on this or on the equivalent question of whether the corresponding computational routine terminates or not does not affect the completely finitist proof that for arguments for which the function *is* defined the routine will compute its value." **(Shepherdson & Sturgis, 1963, p. 217)**

"Unlike the more complex concept of always-terminating mechanical procedures, the unqualified concept, seen clearly now, has the same meaning for the intuitionists as for the classicists." **(Wang, 1974, p. 84)**

"With hindsight, it is not hard to see that the difficulty comes from considering only total functions. It was Kleene (1938) in his remarkable paper who noted that every set of equations can be used to compute a *partial* function. There is no trace of circularity, even for a constructivist, in the definition of *partial recursive function.*" **(Davis, 1982, p. 17)**

"Aber das Hauptziel bei der Einführung dieses Begriffes war eben die exacte Fassung des Konstruktivitätsbegriffes. Die sogenannte Churchsche Thesis identifiziert den Begriff der berechenbaren Funktion mit diesem Begriff. Hier möchte ich nicht darauf eingehen, worüber Kalmár sprechen wird, nämlich ob tatsächlich alle berechenbaren Funktionen allgemein-rekursiv sind; ich möchte gerade die

entgegengesetze Frage aufwerfen: können die allgemein-rekursiven Funktionen sämtlich mit Recht "effecktiv-berechenbar", d.h. "konstruktiv" gennant werden?" **(Péter, 1959, p. 227-228)**

"So werden eigentlich zwei Begriffe der allgemein-rekursiven Funktion definiert: einer mit klassisch aufgefasstem, und einer mit intutionistisch aufgefasstem "es gibt". Es wäre interessant durch ein Beispiel zu zeigen, inwiefern der letztere Begriff enger ist, nämlich durch eine Funktion, welche klassisch allgemein-rekursiv ist und intuitionistisch nicht; das ist aber kaum zu hoffen, da in den bisherigen Betrachtungen noch überhaupt kein Beispiel für eine allgemein- und nicht speziell-rekursive Funktion vorgekommen ist. Nun, der klassische Begriff der allgemeinen-rekursiven Funktion ist nicht konstruktiv, und die intuitionistische enthält ein Circulus vitiosus: hier soll das in der Definition auftretende "es gibt" konstruktiv sein - man wollte aber gerade mit dieser Definition der Allgemein-Rekursivität die Konstruktivität exakt definieren." **(Péter, 1959, p. 228)**

It has been explained by Miss Péter in her conference in this colloquium that any attempt to define the notion of a constructive theory leads into a vicious circle, because the definition always contains an existential quantifier, which in its turn must be interpreted constructively." **(Heyting, 1959, p. 70)**

"If we define the notion of a calculable function as meaning a recursive function, then the definition of the latter notion requires for its interpretation the notion of calculability. Therefore, it is circular to define recursiveness by calculability." **(Heyting, 1962, p. 197)**

"It is clear that for any recursive function of positive integers there exists an algorithm using which any required particular value of the function can be effectively calculated. For the derived equations of the set of recursion equations E are effectively enumerable, and the algorithm for the calculation of particular values of a function F^i, denoted by a functional variable $f^i_{n_i}$, consists in carrying out the enumeration of the derived equations of E until the required particular equation of

the form $f^i_{n_i}(k^i_1, k^i_2, \ldots, k^i_{n_i}) = k^i$ is found." (**Church, 1936, p. 351** e **Davis, 1965, p. 95**)

"The reader may object that this algorithm cannot be held to provide an effective calculation of the required particular value of F^i unless the proof is constructive that the required equation $f^i_{n_i}(k^i_1, k^i_2, \ldots, k^i_{n_i}) = k^i$ will ultimately be found. But if so this merely means that he should take the existential quantifier which appears in our definition of a set of recursion equations in a constructive sense. What the criterion of constructiveness shall be is left to the reader." (**Church, 1936, p. 351** e **Davis, 1965, p. 95**)

"The resulting notion is of especial interest, since the intuitive notion of a 'constructive' or 'effectively calculable' function of natural numbers can be identified with it very satisfactorily." (**Kleene, 1936b, p. 544**)

"Both theories claimed to deal, in different senses, with effective or constructive processes." (**Kleene, 1973, p. 95**)

"There is a constructive and a non-constructive approach to the notion of recursiveness. There seems little doubt that the proper framework for a theory of (abstract) computability is the constructive one." (**van Dalen, 1983, p. 453**)

"The good habit of distinguishing between results on recursive functions obtained by intuitionistic logic and those which for their proof need classical logic is abandoned in many recent papers and books. I regret this, because thereby the connection of the theory with the notion of effective calculability is obscured." (**Heyting, 1962, p. 196**)

"Both ocurrences of the existential quantifier "there exists" are meant [...] in the non-constructive classical sense." (**Mendelson, 1963, p. 202**)

"If we omit the requirement that the computation process always terminate, we obtain a more general class of functions, each function of which is defined over a subset (possibly null or total) of the u-tuples of natural numbers, and posesses the property of effectiveness when defined. These functions we call partial recursive." (**Kleene, 1938, p. 151**)

"Kleene meint, wer das in dieser Allgemeinheit nicht annimmt, mag dieses "es gibt" Konstruktiv auffassen." (**Péter, 1959, p. 228**)

"In 1938, I generalized (or should I say "partialized"?) Gödel's notion of "general recursive functions" to get the notion of "partial recursive functions".

It may amuse you that, in conversation with Gödel in the fall of 1939, I mentioned "partial recursive functions"; and he came right back at me with the question, "What is a partial recursive function?". He evindently had not looked at my 1938 paper.

Apparently, he then embraced the idea." (**Kleene, 1987, p. 57**)

"Gödel points out that the precise notion of mechanical procedures is brought out clearly by Turing machines producing partial rather than general recursive functions." (**Wang, 1974, p. 84**)

"Let P be a property, and let there be an algorithm that decides for every natural number n whether or not n has the property P. If the proposition that no number has the property P leads to a contradiction, then there is a natural number with the property P." (**Markov, 1971, p. 5**)

"It is now natural to ask to what extent the exact concept of a normal algorithm corresponds to the general, not entirely precise, concept of an algorithm in a given alphabet, formulated above. As an answer to this question one can advance the following principle of normalization of algorithms: every algorithm over the alphabet A is equivalent with respect to A to a certain normal algorithm over A." (**Markov, 1960, p. 8**)

"A function is primitive recursive if it can be defined from the functions ..." **(Kleene, 1936, p. 729)**

"Constructively viewed, the above examples are defective, since the principle of the excluded third is constructively false (cf. the Chapter on intuitionistic logic ´Chapter III.5). The constructive reading of "there exists a partial recursive function φ" is: we can effectively compute an index e. Rósza Péter has used the constructive reading of recursion theory as an argument for the circularity of the notion of recursiveness, when based on Church's Thesis, Péter [1959]. The circularity is , however, specious. Recursive functions are not used for computing single numbers, but to yield outputs for given inputs. The computation of isolated discrete objects precedes the manipulations of recursive functions, it is one of the basic activities of constructivism. So the computation of an index of a partial recursive function does itself not need recursive functions." **(van Dalen, 1983, p. 453-454)**

"I have introduced the concept of partial recursive function earlier and with greater emphasis." **(Péter, 1967, p. 9)**.

CITAÇÕES DO CAPÍTULO 3

"The original definition of recursive function ... depends upon the choice of a *P*-symbolism..." **(Hartley Rogers, 1969, p. 146)**

"Wir shalten nun eine Zwischenbetrachtung ein, die mit dem formalen System *P* vorderhand nichts zu tun hat, und geben zunächst folgende Definition: Eine zahlentheoretische Funktion..." **(Gödel, 1986, p. 156)**

"Thus, unlike Gödel´s general recursive functions, the μ-recursive functions are not defined by reference to any *formalism* for expressing equations whose calculations proceed by fixed rules governing the use of that formalism..." **(Webb, 1980, p. 206-207)**

"The μ-recursive functions are a mathematically definable class of functions *almost* independent of syntax and formalism." **(Soare, 1996, p. 229)** (grifo nosso)

"Satz V: Zu jeden rekursiven Relation $R(x_1, \ldots, x_n)$ gibt es ein n-stelligen RELATIONSZEICHEN r (mit den FREIEN VARIABLEN $u_1, \ldots, u_n$) so dass für alle Zahlen-n-tupel $(x_1, \ldots, x_n)$ gilt:

$$R(x_1, \ldots, x_n) \rightarrow Bew[Sb(r\,{}^{u1, \ldots, un}_{Z(x1), \ldots, Z(xn)})]$$

$$\neg R(x_1, \ldots, x_n) \rightarrow Bew[Neg(Sb(r\,{}^{u1, \ldots, un}_{Z(x1), \ldots, Z(xn)}))]$$

" **(Gödel, 1986, p. 170)**.

"Theorem V illustrates Gödel's propensity for speaking in terms of his numbers, rather than directly in terms of the formal objects (which we prefer as being more understandable)." **(Kleene, apud Gödel, 1986, p. 130-131)**

CITAÇÕES DO CAPÍTULO 4

"Robert Soare has been making the point that what recursion theorists study is the class of computable functions on the natural numbers; now this is exactly the class of recursive functions (by classical theorems of Turing and Kleene), and so Soare's point is not a philosophical or foundational position but political advice: he believes that recursion theorists would be appreciated more if they consistently called the functions they study "computable" rather than "recursive". Coming from a country which has practically ridiculed itself in international relations by refusing to call the "Former Yugoslav Republic of Macedonia" what the citizens of that country want to call it, I am not in a position to argue that Bob's point is ridiculous; and he may well be both ridiculous and right.

About me, I do not just study the recursive (or computable) functions on the natural numbers, but "recursion", as a fundamental process of implicit definition, on abstract first-order (and some second-order) structures. The functions defined recursively in these abstract structures cannot always be identified with the functions

which are computable by machines of some reasonable sort; and even when they can, their recursive definitions have interesting "intensional" aspects which are not reflected in the plain fact that, in the end, the function defined is "computable". (For example: I claim that recursive definitions directly express "algorithms", and "more algorithms" than those expressed by machines.) Thus, my use of the term "recursion" is necessary, because what I do with recursions is not always "equivalent" to something which can be done with machines, and so I do not have the luxury of two synonymous terms." **(Moschovakis, 2000)**

REFERÊNCIAS BIBLIOGRÁFICAS

BACKUS, J. Can Programming be liberated from the von Neuman style? A functional style and its algebra of programs, *Comm. of the ACM*, 21, p. 613-641, 1978.

BRIDGES, Douglas S. *Computability*. New York: Springer, 1994.

CAMARÃO DE FIGUEIREDO, Lucília. *λ_Ω-calculus. Um Modelo Para Não-determinismo em Linguagens Reflexivas*. Tese apresentada ao Departamento de Ciência de Computação do Instituto de Ciências Exatas da Universidade de Minas Gerais, como requisito parcial à obtenção do Título de Doutor em Ciência da Computação. Belo Horizonte: Universidade Federal de Minas Gerais, Junho de 1997.

CHURCH, Alonzo. An Unsolvable Problem of Elementary Number Theory, Preliminary Report (Abstract 205). *Bulletin of the American Mathematical Society*, v. 41, p. 332-333, 1935.

——. An Unsolvable Problem of Elementary Number Theory. *The American Journal of Mathematics*, v. 58, p. 345-363, 1936.

CUTLAND, Nigel. *Computability*. Cambridge: Cambridge University Press, 1992.

DAVIS, Martin. *The Undecidable*. New York: Raven Press, 1965.

——. Why Gödel didn't have Church's Thesis. *Information and Control*, v. 54, p. 3-24, 1982.

DAVIS, Martin D., SIGAL, Ron e WEYUKER, Elaine J. *Computability, Complexity and Languages*. Boston: Academic Press, 1994.

EPSTEIN, Richard, CARNIELLI, Walter. *Computability, Computable Functions, Logic and the Foundations of Mathematics*. Pacific Grove: Wadsworth, 1989.

ERTOLA BIRABEN, Rodolfo Cristian. *Tese de Church: Questões Histórico-Conceituais*. Campinas: UNICAMP, CLE, 1996.

ETCHEMENDY, John. *The Concept of Logical Consequence*, Stanford: CSLI Publications, 1999.

FEFERMAN, Solomon. Computation on Abstract Data Types. The extensional approach, with an Application to Streams. *Annals of Pure and Applied Logic*, v. 81, p. 75-113, 1996.

FRIEDMAN, Harvey. Algorithmic Procedures, Generalized Turing Algorithms, and Elementary Recursion Theory. In: R. O. Gandy & C.E.M. Yates (Eds.) *Logic Colloquium '69*, Elsevier: North-Holland, p. 361-389, 1971.

FOLINA, Janet. Church's Thesis. Prelude to a Proof. *Philosophia Mathematica*, v.6, p. 302-323, 1998.

GANDY, Robin. The Confluence of ideas in 1936, in R. Herken (ed.). *The Universal Turing Machine*. Oxford: Oxford University Press.

GÖDEL, Kurt. *Collected Works*. Editado por Solomon Feferman e outros. Oxford: Oxford University Press, vol. I, 1986.

——. Remarks before the Princeton Bicentennial Conference on Problems in Mathematics. In: --. *Collected Works*, vol. II, p. 150, 1990.

——. *Collected Works*. Editado por Solomon Feferman e outros. Oxford: Oxford University Press, vol. III, 1995.

HEYTING, Arend. Blick von der intuitionistischen Warte, *Dialectica*, v. 12, p. 332-345, 1958.

——. After Thirty Years. In: NAGEL, Ernst, SUPPES, Patrick, TARSKI, Alfred (Eds.) *Logic, Methodology and Philosophy of Science*, Proceedings of the 1960 International Congress. Stanford: Stanford University Press, p. 194-197, 1962.

——. Some Remarks on Intuitionism. In: HEYTING, Arend (Ed.) *Constructivity in Mathematics*: Proceedings of the Colloquium held at Amsterdam, 1957. Amsterdam: North Holland, 1959, p. 69-71.

KAPHENGST, Heinz. Eine abstrakte Programmgesteurte Rechenmaschine. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, v. 5, p. 366-379, 1959.

KLEENE, Stephen Cole. General Recursive Functions of Natural Numbers. *Mathematische Annalen*, v. 112, p. 727-742, 1936a.

——. λ-Definability and Recursiveness. *Duke Mathematical Journal*, v. 2, p. 340-353, 1936b.

——. A Note on Recursive Functions. *Bulletin of the American Mathematical Society*, v. 42, p. 544-6, 1936c.

——. On notation for Ordinal Numbers. *The Journal of Symbolic Logic*, v.3, p. 150-155, 1938.

——. On the Interpretation of Intuitionistic Number Theory. *The Journal of Symbolic Logic*, v. 10, p. 109-124, 1945.

——. On the Intuitionistic Logic. *Proceedings of the Tenth International Congress of Philosophy*. Amsterdam: North-Holland, p. 741-743, 1949.

——. *Introduction to Metamathematics*. New York: D. Van Nostrand, 1952.

——. Realizability: A Retrospective Survey. In: DOLD, A. e ECKMANN, B. *Lecture Notes in Mathematics. Cambridge Summer School in Mathematical Logic*, v. 337, p. 95-112, 1973.

——. Origins of Recursive Function Theory. *Annals of the History of Computing*, v. 3, p. 52-67, 1981a.

——. The Theory of Recursive Functions approaching its Centennial. *Bulletin of the American Mathematical Society*, v. 5, p. 43-61, 1981b.

——. Gödel's Impression on Students of Logic in the 1930s. In: WEINGARTNER, P., SCHMETTERER, L. (eds.) *Gödel Remembered*. Nápoli: Bibliopolis, p. 51-64, 1987.

KLINE, Morris. *Mathematical Thought from Ancient to Modern Times*. New York: Oxford University Press, v. 1, 1972.

KREISEL, Georg. The non-derivability of $\neg(x)A(x)\rightarrow\exists x\neg A(x)$, A primitive recursive, in intuitionistic formal systems (abstract). *The Journal of Symbolic Logic*, v.23, p. 456-457, 1958.

KREISEL, Georg. Informal Rigour and Completeness Proofs. In: LAKATOS, Imre (Ed.) *Problems in the Philosophy of Mathematics*. Amsterdam: North-Holland, 1967, p. 138-186.

KRIPKE, Saul. A. Semantical Analysis of Intuitionistic Logic. In: CROSSLEY, J., DUMMETT, M. A. E. (Eds.) *Formal Systems of Recursive Functions*. Amsterdam: North-Holland, 1965. p. 92-130.

——. Outline of a Theory of Truth. *The Journal of Philosophy*, v. 72, p. 690-716, 1975.

LANDIN, . The Mechanical Evaluation of Expressions. *Computer Journal*, v. 6, p. 308-320, 1964.

MARKOV, Andrei A. The Theory of Algorithms (Ruso). *Trudy Mat. Inst. Steklov.*, v. 38, p. 176-189, 1951. Tradução em *American Mathematical Society Translations*, Series 2, v. 15, p. 1-14, 1960.

——. On Constructive Mathematics (Ruso). *Trudy Mat. Inst. Steklov.*, v. 67, p. 8-14, 1962. Tradução em *American Mathematical Society Translations*, Series 2, v. 98, p. 1-9, 1971.

MCCARTHY, John. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I, *Communications of the ACM*, v. 3, p. 184-195, 1960.

MENDELSON, Elliott. On Some Recent Criticism of Church's Thesis. *Notre Dame Journal of Formal Logic*, v. 4, p. 201-205, 1963.

——. *Introduction to Mathematical Logic*, New York: D. Van Nostrand, 1979.

——. Second Thoughts about Church´s Thesis and Mathematical Proofs. *The Journal of Philosophy*, v. 87, p. 225-233, 1990.

——. *Introduction to Mathematical Logic*, London: Chapman & Hall, 1997.

MONK, James Donald. *Mathematical Logic*. New York: Springer, 1976.

MOSCHOVAKIS, Joan R. Kleene's Realizability and "Divides" Notions for Formalized Intuitionistic Mathematics. In: BARWISE, J., KEISLER, H. e KUNEN, K. (eds.)

MOSCHOVAKIS, Yiannis N. Abstract Recursion as a Foundation for the Theory of Algorithms. In: *Lecture Notes in Mathematics*, Eds.: A. Dold & B. Eckmann, v. 1104, p. 289-364. Berlin: Springer, 1984.

MOSCHOVAKIS, Yiannis N. The Formal Language of Recursion. *The Journal of Symbolic Logic*, v. 54, p. 1216-1252, 1989a.

——. A Mathematical Modeling of Pure Recursive Algorithms. In: MEYER, A.R. & TAITSLIN, M.A. (Eds.) *Logic at Botik'89, Lecture Notes in Computer Science*, v. 363 , p. 208-229. Berlin: Springer, 1989b.

——. Sense and Denotation as Algorithm and Value. In: J. Oikkonen e J. Vaananen (Eds.) *Logic Colloquium '90*, p. 210-249, Berlin:Springer, 1993

——. *Notes on Set Theory*. Berlin: Springer, 1996.

—— (UCLA). Comunicação Pessoal, 2000.

MYHILL, John. The Invalidity of Markoff's Schema. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, v. 9, p. 359-360, 1963.

NELSON, David. Recursive Functions and Intuitionistic Number Theory. *Transactions of the American Mathematical Society*, v. 61, p. 307-368, 1947.

PÉTER, Rózsa. Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktion. *Mathematische Annalen*, v. 110 (1934), p. 612-632.

——. Graphschemata und Rekursive Funktionen. *Dialectica*, v. 12, 1958, p. 373-393.

——. Rekusivität und Konstruktivität. In: HEYTING, Arend (Ed.) *Constructivity in Mathematics*: Proceedings of the Colloquium held at Amsterdam, 1957. Amsterdam: North Holland, 1959, p. 226-233.

——. *Recursive Functions in Computer Theory*. Chichester: Ellis Norwood Limited, 1981.

PHILIPS, I. C. C. Recursion Theory. In: Handbook of Logic in Computer Science, v. 1, ed. por Samson Abramsky, Dov. Gabbay e T. S. E. Maibaum. Oxford: Clarendon Press, 1992.

RICHMAN, Fred. Church's Thesis without Tears. *The Journal of Symbolic Logic*, v. 48, p. 797-803, 1983.

ROGERS Jr., Hartley. *Theory of Recursive Functions and Effective Calculability*, New York: McGraw Hill, 1967.

ROGERS Jr., Hartley. The Present Theory of Turing Machine Computability. Em Hintikka, Jaakko. *The Philosophy of Mathematics*, London: Oxford University Press, 1969.

ROSE, Gene F. Propositional Calculus and Realizability. *Transactions of the American Mathematical Society*, v. 75, p. 1-19, 1953.

ROSSER, John B. *Highlights of the History of lambda-calculus*. ACM Symposium on Lisp and Functional Programming (Pennsylvania), ACM Press, p. 216-225, 1982.

SHAPIRO, Stewart. Understanding Church´s Thesis. *Journal of Philosophical Logic*, v. 10, p. 353-365, 1981.

——. Understanding Church's Thesis again. *Acta Analytica*, 11, p. 59-77, 1993.

SHEPHERDSON, John Cedric & STURGIS, H. E. Computability of Recursive Functions. *Journal of the Association of Computing Machinery*, v. 10, p. 217-255, 1963.

SHEPHERDSON, John Cedric. Algorithmic Procedures, Generalized Turing Algorithms, and Elementary Recursion Theory. In: L. A. Harrington et al. (eds.), *Harvey Friedman´s Research on the Foundations of Mathematics*. Elsevier: North-Holland, p. 285-308, 1985.

SIEG, Wilfried. Step by Recursive Step: Church's Analysis of Effective Calculability. *The Bulletin of Symbolic Logic*, v. 3, p. 154-180, 1997.

SOARE, Robert I. Computability and Recursion. *The Bulletin of Symbolic Logic*, v. 2, p. 284-321, 1996.

THOMAS, William J. Doubts About Some Standard Arguments for Church´s Thesis. In: BOGDEN, Radu, NIINILUOTO, Ilkka (Eds.) *Logic, Language and Probability*. Dordrecht: D. Reidel, 1973. P. 55-65

TROELSTRA, A. S. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Berlin: Springer, 1973.

TROELSTRA, A. S., VAN DALEN, D. *Constructivism in Mathematics*. Amsterdam: North-Holland, v. 1, 1988.

TURING, Alan Mathison. On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Second Series, v. 42, 1937, p. 230-265.

VAN DALEN, Dirk. "Algorithms and Decision Problems: A Crash Course in Recursion Theory" (p. 409-478) in *Handbook of Philosophical Logic*, v. I: Dordrecht, 1983.

WANG, Hao. F*rom Mathematics to Philosophy*. New York: Humanities Press, 1974.

WEBB, Judson C. *Mechanism, Mentalism and Metamathematics*. Dordrecht: D. Reidel, 1980.