UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS HUMANAS, LETRAS E ARTES
DEPARTAMENTO DE FILOSOFIA
PROGRAMA DE PÓS-GRADUAÇÃO EM FILOSOFIA

GUSTAVO CAVALCANTI DE MELO

FUNÇÕES PARCIAIS RECURSIVAS E FUNÇÕES PARCIALMENTE TURING-COMPUTÁVEIS: UMA PROVA DE EQUIVALÊNCIA

JOÃO PESSOA – PB
2016

GUSTAVO CAVALCANTI DE MELO

FUNÇÕES PARCIAIS RECURSIVAS E FUNÇÕES PARCIALMENTE TURING-COMPUTÁVEIS: UMA PROVA DE EQUIVALÊNCIA

Dissertação apresentada ao Programa de Pós-Graduação em Filosofia da Universidade Federal da Paraíba como requisito para obtenção do título de mestre em Filosofia.

Orientador: Prof. Dr. Matias Francisco Dias

JOÃO PESSOA – PB
2016

M528f Melo, Gustavo Cavalcanti de.

Funções parciais recursivas e funções parcialmente Turing-computáveis: uma prova de equivalência / Gustavo Cavalcanti de Melo. – João Pessoa, 2016.

83f.

Orientador: Matias Francisco Dias

Dissertação (Mestrado) – UFPB/CCHLA

1. Ciências da computação. 2. Função parcial recursiva. 3. Função parcialmente Turing-computável. 4. Teorema de Rice. 5. Problema de decisão.

UFPB/BC CDU: 681.3(043)

GUSTAVO CAVALCANTI DE MELO

FUNÇÕES PARCIAIS RECURSIVAS E FUNÇÕES PARCIALMENTE TURING-COMPUTÁVEIS: UMA PROVA DE EQUIVALÊNCIA

Dissertação apresentada ao Programa de Pós-Graduação em Filosofia da Universidade Federal da Paraíba como requisito para obtenção do título de mestre em Filosofia.

Aprovada em 24 de outubro de 2016

BANCA EXAMINADORA

Matias Francisco Dias

Prof. Dr. Matias Francisco Dias - UFPB

(Orientador)

Prof. Drª. Ana Lêda de Araújo - UFPB

Prof. Dr. Garibaldi Monteiro Sarmento - UFPB

Prof. Dr. Rodrigo Costa Ferreira - UEPB

AGRADECIMENTOS

A Cristo, por sua constante providência na minha vida e por colocar em meu caminho pessoas tão especiais.

Aos meus alicerces, Francisco de Assis, Graciete Melo e Marcel Melo, pelo cuidado, incentivo e pela sólida formação que proporcionou a continuidade dos meus estudos.

À minha namorada, Denise Pereira, por compartilhar comigo as alegrias e dificuldades da vida acadêmica.

Ao meu amigo Paulo Fernando, pela convivência durante alguns dias da semana e pela divisão dos trabalhos domésticos que, com certeza, me rendeu horas a mais de pesquisa.

À minha amiga Lucemar Gomes. Sem medir esforços, ela sempre foi para mim sinônimo de companheirismo e dedicação.

Ao meu amigo Victor Pereira, pelas horas de estudo e pelos momentos indescritíveis que passamos juntos durante estes anos de pós-graduação.

À professora Ana Lêda, com quem tive os primeiros contatos com a Teoria da Computabilidade. Sob seu olhar, aperfeiçoei a arte de ensinar e, com suas atitudes, aprendi, na prática, o significado do termo “ética profissional”.

Ao meu orientador, Matias Francisco, que, com muita paciência e disponibilidade, transmitiu o seu saber, colaborando diretamente na construção deste trabalho.

Ao meu amigo Leonardo Weber, pelo seu constante empenho em esclarecer as minhas dúvidas e pelas discussões, às terças-feiras à tarde, que me fizeram vislumbrar novas possibilidades de estudo.

Aos professores Garibaldi Monteiro Sarmento e Rodrigo Costa Ferreira, por aceitarem, de bom grado, compor a banca examinadora.

Para finalizar, meus sinceros agradecimentos a todas as pessoas que, de alguma maneira, contribuíram com a minha formação acadêmica.

RESUMO

Na década de 30 do século passado, foram oferecidas várias versões formais para a noção intuitiva de função algorítmica. Dentre elas, a versão das funções recursivas e a versão das funções Turing-computáveis. Posteriormente, tais versões foram estendidas a fim de abranger também as funções parciais algorítmicas, dando origem, deste modo, à versão das funções parciais recursivas e à versão das funções parcialmente Turing-computáveis. Nesse contexto, esta pesquisa, situada dentro do domínio da Teoria da Computabilidade e construída à luz dos pressupostos teóricos de Davis (1982), Mendelson (2009), Dias e Weber (2010), Rogers (1987), Soare (1987), Cooper (2004), entre outros, destina-se a reconstruir a prova de que as referidas versões formais dadas para a noção intuitiva de função parcial algorítmica, apesar de conceitualmente distintas, são extensionalmente equivalentes no sentido de que elas determinam o mesmo conjunto de funções numéricas. Como parte desta reconstrução, provaremos, de modo inédito, mediante o uso de quíntuplas, que toda função parcial recursiva é parcialmente Turing-computável. Na literatura especializada, esse teorema é provado por meio de um conjunto de quádruplas. Porém, definindo um conjunto de menor cardinalidade constituído por quíntuplas, é possível prová-lo em um intervalo menor de tempo, o que representa um ganho do ponto de vista computacional. Além de apresentar essa prova alternativa, posto pela Tese de Church-Turing que o conjunto das funções parciais recursivas contém todas as funções parciais algorítmicas, investigaremos se ele próprio e os seus infinitos subconjuntos são ou não algorítmicos. Nesta investigação, demonstraremos, em termos aritméticos, com o auxílio do Teorema de Rice, que embora o conjunto das funções parciais recursivas seja algorítmico, todos os seus subconjuntos diferentes do conjunto vazio não o são, dentre os quais estão o conjunto das funções recursivas e o conjunto das funções recursivas primitivas.

Palavras-chave: função parcial recursiva, função parcialmente Turing-computável, teorema de Rice, problema de decisão.

ABSTRACT

In the thirties of the last century, several formal versions for the intuitive notion of algorithmic function were offered. Among them, the version of the recursive functions and the version of the Turing-computable functions. Posteriorly, such versions were extended in order to also include the partial algorithmic functions, giving rise, in this way, to the version of the partial recursive functions and to the version of the partially Turing-computable functions. In this context, this research, located into Computability Theory domain and built in the light of theoretical assumptions of Davis (1982), Mendelson (2009), Dias & Weber (2010), Rogers (1987), Soare (1987), Cooper (2004), among others, is intended to rebuild the proof that the given formal versions referred to the intuitive notion of partial algorithmic function, despite being conceptually distinct, they are extensionally equivalents in the sense that they determine the same set of theoretical-numerical functions. As a part of this rebuilding, we shall prove, in na unprecedented way, using quintuples, that every partial recursive function is partially Turing-computable. In the literature, this theorem is proved by means of a set of quadruples. However, defining a lower cardinality set constructed by quintuples, it is possible to prove it in a smaller time interval, which representes a gain from the computational point of view. Besides presenting this alternative proof, posed by the Church-Turing thesis that the set of partial recursive functions includes all the partial algorithmic functions, we shall investigate if this set itself and its infinite subsets are or are not algorithmic. In this survey, we shall demonstrate, in arithmetical terms, with the aid of Rice's theorem, that although the set of partial recursive functions is algorithmic, all its subsets which are different from the empty set are not, among which are the set of recursive functions and the set of primitive recursive functions.

Keywords: partial recursive functions, partially Turing-computable functions, Rice's theorem, decision problem.

SUMÁRIO

INTRODUÇÃO

Nesta dissertação, pretendemos explicitar a equivalência entre duas versões formais oferecidas para a noção intuitiva de função parcial algorítmica: a versão das funções parciais recursivas sistematizada por Kleene a partir dos trabalhos de Herbrand e Gödel e a versão das funções parcialmente Turing-computáveis concebida por Alan Turing. Em outras palavras, pretendemos reconstruir a prova segundo a qual o conjunto das funções parciais recursivas e o conjunto das funções parcialmente Turing-computáveis, apesar de conceitualmente distintos, possuem os mesmos elementos. Em seguida, buscaremos demonstrar, em termos aritméticos, com o auxílio do Teorema de Rice, que embora estes conjuntos sejam decidíveis, eles contêm infinitos subconjuntos indecidíveis, dentre os quais, destacam-se o conjunto das funções recursivas e o conjunto das funções recursivas primitivas.

Em Teoria da Computabilidade, ramo da Lógica no qual estará concentrada nossa pesquisa, noções como algoritmo, função algorítmica e decidibilidade mantêm entre si uma estreita relação. Intuitivamente, entende-se por algoritmo um procedimento mecânico para computar uma função. Na prática, nós o identificamos como um conjunto finito de regras inequívocas (em linguagem natural ou simbólica) que devem ser aplicadas, sem nenhum recurso à criatividade, a um dado *input* finito, fornecendo, após a execução de um número finito de operações elementares, um possível *output* também finito. Uma função, por sua vez, é algorítmica se, e somente se, existe um algoritmo para computá-la. E um conjunto é decidível (ou algorítmico) se, e somente se, existe um algoritmo que nos permita identificar, dado um objeto qualquer, se ele pertence ou não a este conjunto. Caso contrário, o conjunto é indecidível. A noção de decidibilidade é, analogamente, aplicável a teorias formais. Neste caso, dizemos que uma teoria formal é decidível se, e somente se, o conjunto de seus teoremas for decidível.

Nos primeiros anos do século XX, o surgimento de alguns paradoxos na recente Teoria dos Conjuntos de Cantor abalou profundamente os fundamentos da Matemática. Diante deste fato, o analista alemão David Hilbert (apud SOBRINHO, 1987, p. 4) proferiu as seguintes palavras:

> O atual estado de coisas, em que estamos nos defrontando com paradoxos, é, de fato, absolutamente intolerável. Imagine se as definições e métodos dedutivos que todos aprendemos, ensinamos e utilizamos em Matemática nos conduzirem a absurdos! Se o próprio pensamento matemático já for defeituoso, onde é que iremos encontrar a verdade e a certeza?

Entre as muitas tentativas de solucionar o problema, Hilbert propôs o programa formalista de fundamentação da Matemática. Este programa era assim chamado por defender a reconstrução do edifício matemático a partir de métodos axiomáticos formais, cujo rigor característico, segundo Hilbert, impediria a ocorrência de contradições. Em última análise, a Matemática, na visão dos formalistas, seria redutível a sistemas axiomáticos constituídos exclusivamente por símbolos isentos de interpretação, manipulados através de regras precisas e mecanismos finitários. Em termos mais elementares, eles encaravam a Matemática como um mero jogo formal. Os passos permitidos pelas regras de inferência em uma demonstração seriam, por exemplo, os lances possíveis de um jogo de tabuleiro, os axiomas corresponderiam à configuração inicial do tabuleiro e as fórmulas, às peças do jogo. Além disso, o ato de 'jogar' seria análogo ao de 'executar operações matemáticas' e as declarações *sobre* o jogo equivaleriam a declarações *sobre* a Matemática.

Na tentativa de estabelecer bases sólidas para o pensamento matemático de modo a evitar resultados contraditórios, além de propor o uso de métodos axiomáticos formais, Hilbert convida a comunidade acadêmica a oferecer uma solução positiva para os três problemas referentes aos fundamentos da Matemática. O primeiro deles, o problema da consistência. Hilbert pretendia demonstrar que as diversas teorias matemáticas eram consistentes, ou seja, não admitiriam contradições. O segundo, o problema da completude, para o qual uma solução positiva implicaria afirmar que, dado um enunciado de uma teoria matemática qualquer, ele ou a sua negação seriam demonstráveis nesta teoria. Por fim, o terceiro problema, conhecido como *Entscheidunsproblem*, que investiga se o Cálculo de Predicados de Primeira Ordem é decidível. Hilbert caracterizou o *Entscheidunsproblem* como o problema fundamental da Lógica Matemática, pois ele acreditava que a solução deste problema permitiria, pelo menos em princípio, decidir, sem nenhum recurso à criatividade, se, dada uma teoria matemática qualquer, uma fórmula pertencente a esta teoria seria ou não um de seus teoremas. Essa crença de Hilbert foi confirmada em 1929, quando Gödel demonstrou a completude do Cálculo de Predicados de Primeira Ordem, apresentando-o como "uma linguagem e uma lógica completa servindo de embasamento para a formalização das teorias matemáticas" (SOBRINHO, 1987, p. 7).

Uma solução positiva para os três problemas hilbertianos estabeleceria a Matemática como um grandioso cálculo axiomático formal - consistente, completo e decidível. A Matemática seria, incontestavelmente, segura (livre de contradições) e responderia a todos os problemas a ela referentes de modo efetivo. No entanto, as pretensões do programa formalista

fracassaram por duas vezes. Em 1931, Kurt Gödel publicou os famosos teoremas da incompletude. De acordo com o primeiro teorema, toda axiomática consistente da aritmética de Peano é incompleta. Por contraposição, toda axiomática completa da aritmética de Peano é inconsistente. Sendo assim, ao contrário do que Hilbert acreditava, a Matemática, não poderia ser completa e consistente ao mesmo tempo. Já o segundo teorema de Gödel garantiu que a consistência de uma axiomática da aritmética de Peano não é demonstrável somente com os recursos dessa axiomática. Seria então necessário, lançar mão de uma axiomática mais forte na qual esta demonstração fosse possível. No entanto, a prova da consistência desta última axiomática demandaria uma outra axiomática ainda mais forte e assim sucessivamente. Este resultado, portanto, destruiu a esperança de Hilbert de encontrar uma prova finitária da consistência da aritmética e, consequentemente, da consistência da Matemática. Em 1936, o programa hilbertiano, mais uma vez, fracassou: Alonzo Church prova, formalmente, a indecidibilidade do Cálculo de Predicados de Primeira Ordem, obtendo, desta maneira, uma resposta negativa para o *Entscheidunsproblem*. Na mesma época, provou-se também, de modo semelhante, a indecidibilidade de diversas teorias matemáticas, entre elas a Aritmética de Peano.

Essas provas formais de indecidibilidade só foram possíveis a partir da década de 30 do século passado, com o advento da Teoria da Computabilidade, quando lógicos e matemáticos propuseram várias caracterizações precisas para as noções intuitivas de algoritmo e, consequentemente, de função algorítmica, visando à obtenção de uma resposta em termos matemáticos para o *Entscheidunsproblem*. Neste contexto, podemos afirmar que o surgimento da Teoria da Computabilidade foi motivado, em última análise, pela hipótese de Hilbert segundo a qual as diversas teorias matemáticas eram decidíveis. Ainda na década de 30, provou-se que todas as versões formais oferecidas para as referidas noções intuitivas eram equivalentes, gerando, desde então, a crença segundo a qual se tinha captado, de uma vez por todas, de forma precisa, o que se entendia intuitivamente por algoritmo e função algorítmica. Sobre este acontecimento, Hao Wang (apud SOBRINHO, 1987, p. 1) escreve:

> Uma das grandes conquistas da Lógica desde os anos 30 foi o sucesso experimentado ao ter sido dada uma definição absoluta (i.e., independente do particular formalismo adotado) da interessante noção epistemológica de processo mecânico (ou procedimento efetivo, computabilidade, algoritmo, método finitista). Com efeito, pode-se afirmar que tenha sido o único conceito epistemológico básico relacionado com a Matemática que tenhamos sido capazes de iluminar até agora.

A fim de lograrmos êxito na realização dos objetivos mencionados no início desta introdução, dividiremos o nosso trabalho em três capítulos, contando sempre com o suporte teórico de vários autores importantes da Lógica e da Teoria da Computabilidade, entres eles: Davis, Mendelson, Dias e Weber, Rogers, Soare e Cooper.

No primeiro capítulo, estabeleceremos indutivamente o conjunto das funções parciais recursivas. De modo análogo, definiremos, em ordem decrescente de generalidade, outros dois conjuntos: o das funções recursivas e o das funções recursivas primitivas. Explicitaremos as diferenças e semelhanças entre eles, analisando detalhadamente as suas respectivas definições. Para cada conjunto estabelecido, apresentaremos um modelo de algoritmo capaz de computar suas funções. Por fim, investigaremos, em termos intuitivos, se os conjuntos de algoritmos apresentados são ou não decidíveis.

No segundo capítulo, apresentaremos a teoria das máquinas e dos programas de Turing, destacando o conjunto das funções parcialmente Turing-computáveis. Definiremos os programas de Turing de modo ligeiramente diverso do habitual – normalmente, eles são definidos como conjuntos de quádruplas e nós os definiremos como conjuntos de quíntuplas. A partir desta maneira de defini-los, provaremos, de modo inédito, que toda função parcial recursiva é parcialmente Turing-computável. Em seguida, utilizando a aritmetização das máquinas de Turing, provaremos que a recíproca também é verdadeira. Estabelecida a equivalência entre os dois formalismos propostos para a noção intuitiva de função algorítmica, concluiremos o capítulo, expondo a famosa Tese de Church-Turing, segundo a qual todo procedimento computacional é realizável por uma máquina de Turing ou, em outras palavras, toda função algorítmica é Turing-computável.

Posto pela Tese de Church-Turing que o conjunto das funções parciais recursivas contém todas as funções algorítmicas, cabe-nos perguntar se este conjunto é propriamente algorítmico. E o que dizer de seus subconjuntos: são ou não algorítmicos? Investigaremos estas questões no terceiro capítulo. Nele, veremos, inicialmente, como construir uma lista efetiva dos programas de Turing e das funções parciais recursivas. Vamos utilizá-la na prova de dois resultados importantes da Teoria da Computabilidade: o Teorema s-m-n de Kleene e a indecidibilidade do Problema da Parada. Com base nestes resultados, demonstraremos o Teorema de Rice a partir do qual provaremos, por um lado, a decidibilidade do conjunto das funções parciais recursivas e, por outro, a indecidibilidade de qualquer um de seus subconjuntos próprios, que não seja vazio.

Para iniciarmos o estudo ao qual nos propomos, alguns esclarecimentos serão convenientes: (1) em concordância com a Teoria da Computabilidade, restringiremos o nosso estudo ao conjunto $\mathbb{N}$ de números naturais, tratando, deste modo, apenas de funções numéricas *n*-árias, ou seja, funções cujos argumentos são *n*-uplas ordenadas de $\mathbb{N}^n$ e os valores são elementos de $\mathbb{N}$; (2) classificaremos uma função numérica *n*-ária como *total* se ela estiver definida para todas as *n*-uplas de $\mathbb{N}^n$ e, como *parcial*, se ela estiver definida para todas, algumas ou nenhuma *n*-upla de $\mathbb{N}^n$; (3) diremos, em termos intuitivos, que uma função numérica *n*-ária total é algorítmica se, e somente se, for possível calcular o seu valor para cada *n*-upla de $\mathbb{N}^n$ através de um algoritmo, isto é, em um número finito de passos e de maneira inteiramente mecânica; diremos que uma função numérica *n*-ária parcial é algorítmica se, e somente se, existe um algoritmo para computá-la sempre que ela estiver definida para uma determinada *n*-upla de $\mathbb{N}^n$; (4) por fim, acompanhando os autores citados no início desta introdução, utilizaremos, nas páginas seguintes, o termo "função algorítmica" para nos referirmos somente às funções numéricas totais algorítmicas.

1 RECURSIVIDADE

A fim de alcançarmos o primeiro objetivo ao qual nos propomos, qual seja, estabelecer a igualdade entre o conjunto das funções parciais recursivas e o conjunto das funções parcialmente Turing-computáveis, precisamos, antes de mais nada, definir cada um desses conjuntos separadamente. Neste capítulo, apresentaremos o conjunto das funções parciais recursivas. Dentre os seus inúmeros subconjuntos, destacaremos o conjunto das funções recursivas e o conjunto das funções recursivas primitivas. Ambos, como veremos, contêm, exclusivamente, funções numéricas totais.

1.1 Funções parciais recursivas

Em geral, o conjunto das funções parciais recursivas é estabelecido por meio de uma definição indutiva. Para defini-lo, começaremos fixando os primeiros elementos deste conjunto conhecidos como funções iniciais e, em seguida, listaremos algumas regras, conhecidas como operações básicas, que nos permitem obter novas funções parciais recursivas a partir de outras previamente dadas. Na sequência, utilizaremos o símbolo x' para denotar o número que segue imediatamente *x* na ordem dos números naturais e a expressão '$f(x_1, \dots, x_n)\downarrow$' para indicar que a função f está definida para a *n*-upla $x_1, \dots, x_n$; para indicar o caso contrário, escreveremos '$f(x_1, \dots, x_n)\uparrow$'.

Definição 1.1 As *funções iniciais* são:

(1) Função sucessor: $S(x) = x'$, para qualquer x.

(2) Funções-constante: $C_k^n(x_1, \dots, x_n) = k$, para quaisquer $x_1, \dots, x_n$ e k.

(3) Funções-projeção: $U_i^n(x_1, \dots, x_n) = x_i$, $1 \leq i \leq n$, para qualquer $x_1, \dots, x_n$.

Definição 1.2 As *operações básicas* são:

(1) Composição (para $n, m \geq 1$)

Se f é uma função *m*-ária e $g_1, \dots, g_m$ são funções *n*-árias, então a função *n*-ária h é obtida, por *composição*, a partir de $f, g_1, \dots, g_m$ se, e somente se:

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

(2) Recursão primitiva (para $n \geq 0$)

Se g é uma função *n*-ária e f uma função *n*+2-ária, então a função *n*+1-ária h é obtida, por *recursão primitiva*, a partir de g e f se, e somente se:

$$h(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$$

$$h(x_1, \ldots, x_n, \mathrm{S}(y)) = f(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y))$$

Em particular, se $n = 0$, por recursão primitiva, é obtida a função unária h tal que:

$$h(0) = k$$
$$h(\mathrm{S}(y)) = f(y, h(y)),$$

onde k é um número natural qualquer.

(3) Minimização ilimitada (para $n \geq 1$)

Se g é uma função n+1-ária, então a função n-ária h é obtida, por *minimização ilimitada*, a partir de g se, e somente se:

$$h(x_1, \ldots, x_n) = \mu_y(g(x_1, \ldots, x_n, y) = 0) = \begin{cases} z_0\,, & \text{se } g(x_1, \ldots, x_n, z_0) = 0 \wedge \\ & \forall_{w<z_0}(g(x_1, \ldots, x_n, w) \downarrow \wedge \\ & g(x_1, \ldots, x_n, w) \neq 0), \\ & \text{para algum } z_0 \\ \uparrow\,, & \text{se não existe tal } z_0 \end{cases}$$

É oportuno esclarecer, em outras palavras, que a função $h(x_1, \ldots, x_n)$, obtida por minimização ilimitada, terá como valor o menor número natural z_0 tal que $g(x_1, \ldots, x_n, z_0) = 0$, sob a condição de não existir um $w < z_0$ para o qual $g(x_1, \ldots, x_n, w) \uparrow$. Porém, se tal z_0 não existir, $h(x_1, \ldots, x_n)$ permanecerá indefinida.

Apresentadas as funções iniciais e as operações básicas, podemos, agora, estabelecer, de modo apropriado, o conjunto das funções parciais recursivas.

Definição 1.3 Uma função numérica f é *parcial recursiva* se, e somente se, uma das seguintes condições é satisfeita:

(1) f é uma função inicial.

(2) f é obtida a partir de funções parciais recursivas pela aplicação de uma das operações básicas.

(3) Somente são funções parciais recursivas as funções numéricas determinadas de acordo com (1) ou (2).

Definição 1.4 Uma *sequência parcial recursiva* para uma função numérica f é uma sequência finita de funções $(\alpha_1, \ldots, \alpha_n)$ se, e somente se, $\alpha_n = f$ e, para cada $1 \leq i \leq n$, uma das seguintes condições é satisfeita:

(1) α_i é uma função inicial;

(2) α_i é obtida de funções anteriores da sequência por aplicação de uma das operações básicas.

Teorema 1.5 Uma função numérica f é parcial recursiva se, e somente se, existe uma sequência parcial recursiva para *f*.

Prova:

Imediata, pelas definições 1.3 e 1.4.

O conjunto das funções parciais recursivas, tal como foi definido, contém como seus elementos funções totais e funções estritamente parciais. Uma simples restrição sobre o campo de aplicação da operação de minimização ilimitada nos oferecerá um recorte deste conjunto. Sabendo disso, estabeleceremos, utilizando a noção de função regular, o conjunto das funções recursivas que, como veremos, contêm, exclusivamente, funções parciais recursivas totais.

Definição 1.6 Uma função numérica $g(x_1, \dots, x_n, y)$, $n \geq 1$, é *regular* se, e somente se, g é total e $\forall x_1 \dots \forall x_n \exists y(g(x_1, \dots, x_n, y) = 0)$.

Definição 1.7 Uma função numérica f é *recursiva* se, e somente se, uma das seguintes condições é satisfeita:
(1) *f* é uma função inicial.
(2) *f* é obtida a partir de funções recursivas por aplicação das operações de composição ou recursão primitiva ou minimização ilimitada, sendo esta última aplicada exclusivamente a funções regulares.
(3) Somente são funções recursivas as funções numéricas determinadas de acordo com (1) ou (2).

Definição 1.8 Uma *sequência recursiva* para uma função numérica *f* é uma sequência finita de funções $(\alpha_1, \dots, \alpha_n)$ se, e somente se, $\alpha_n = f$ e, para cada $1 \leq i \leq n$, uma das seguintes condições é satisfeita:
(1) α_i é uma função inicial;
(2) α_i é obtida de funções anteriores da sequência por aplicação das operações de composição ou recursão primitiva ou minimização ilimitada, sendo esta última aplicada exclusivamente a funções regulares.

Teorema 1.9 Uma função numérica *f* é recursiva se, e somente se, existe uma sequência recursiva para *f*.

Prova:

Imediata, considerando as definições 1.7 e 1.8.

Deve-se notar que o conjunto das funções recursivas é definido exatamente igual ao conjunto das funções parciais recursivas, salvaguardada a seguinte diferença: a operação de minimização ilimitada, na definição das funções recursivas, é aplicada somente às funções regulares, enquanto que na definição das funções parciais recursivas, tal operação pode ser aplicada indistintamente. Esta restrição faz com que o conjunto das funções recursivas contenha única e exclusivamente como seus elementos todas as funções parciais recursivas totais, pois, pela definição de função regular, sempre existirá um número natural z_0 tal que $\mu_y(g(x_1, \dots, x_n, y) = 0) = z_0$.

Sendo o conjunto das funções parciais recursivas uma versão formal oferecida para o conceito intuitivo de função parcial algorítmica e sabendo que o conjunto das funções recursivas contêm exclusivamente todas as funções parciais recursivas totais, o conjunto das funções recursivas constitui, portanto, a versão formal oferecida para o conceito intuitivo de função algorítmica.

Ainda sobre o conjunto das funções recursivas, destacamos, como uma de suas partes, o conjunto das funções recursivas primitivas que, sendo estabelecido somente a partir das funções iniciais e das operações de recursão primitiva e composição, contém, como apresentaremos mais adiante, diversas funções numéricas conhecidas como claramente algorítmicas.

Definição 1.10 Uma função numérica *f* é *recursiva primitiva* se, e somente se, uma das seguintes condições é satisfeita:
(1) *f* é uma função inicial.
(2) *f* é obtida a partir de funções recursivas por aplicação das operações de composição ou recursão primitiva.
(3) Somente são funções recursivas as funções numéricas determinadas de acordo com (1) ou (2).

Definição 1.11 Uma *sequência recursiva primitiva* para uma função numérica f é uma sequência finita de funções $(\alpha_1, \ldots, \alpha_n)$ se, e somente se, $\alpha_n = f$ e, para cada $1 \leq i \leq n$, uma das seguintes condições é satisfeita:
(1) α_i é uma função inicial;
(2) α_i é obtida de funções anteriores da sequência por aplicação das operações de composição ou recursão primitiva.

Teorema 1.12 Uma função numérica f é recursiva primitiva se, e somente se, existe uma sequência recursiva primitiva para f.

Prova:

Imediata, tendo em vista as definições 1.10 e 1.11.

Uma sequência recursiva primitiva para f, cujas funções estão acompanhadas por uma especificação de como foram obtidas, constitui uma derivação recursiva primitiva para f. Ora, dada uma sequência recursiva primitiva é sempre possível especificar cada uma de suas funções. Sendo assim, parafraseando o teorema anterior, uma função f é recursiva primitiva se, e somente se, existe uma derivação recursiva primitiva para ela. De modo análogo, falaremos também em derivações recursivas e derivações parciais recursivas.

Listaremos, a seguir, uma série de funções recursivas primitivas que serão essenciais para a obtenção de vários resultados apresentados mais adiante.

Teorema 1.13 As seguintes funções são recursivas primitivas:

(1) adição: $ad(x, y) = x + y$

$ad(x, 0) = x$
$ad(x, S(y)) = S(ad(x, y))$

(2) multiplicação: $ml(x, y) = x \cdot y$

$ml(x, 0) = 0$
$ml(x, S(y)) = ad(x, ml(x, y))$

(3) exponenciação: $ep(x, y) = x^y$

$ep(x, 0) = 1$
$ep(x, S(y)) = ml(x, ep(x, y))$

(4) fatorial: $ft(x) = x!$

$ft(0) = 1$

$ft(S(y)) = ml(S(y), ft(y))$

(5) predecessor: $pd(x) = \begin{cases} 0, & \text{se } x = 0 \\ x - 1, & \text{se } x \neq 0 \end{cases}$

$pd(0) = 0$

$pd(S(y)) = y$

(6) subtração própria: $sp(x, y) = \begin{cases} x - y, & \text{se } x \geq y \\ 0, & \text{se } x < y \end{cases}$

$sp(x, 0) = x$

$sp(x, S(y)) = pd(sp(x, y))$

(7) mínimo de um par ordenado: $mn(x, y) = \begin{cases} x, & \text{se } x \leq y \\ y, & \text{se } x > y \end{cases}$

$mn(x, y) = sp(x, sp(x, y))$

(8) máximo de um par ordenado: $mx(x, y) = \begin{cases} y, & \text{se } x \leq y \\ x, & \text{se } x > y \end{cases}$

$mx(x, y) = ad(y, sp(x, y))$

(9) mínimo de uma sequência finita: $min(x_1, \dots, x_n) = x_i$: x_i é o menor número da sequência

$min(x_1, \dots, x_n) = mn(\dots mn(mn(x_1, x_2), x_3), \dots, x_n)$

(10) máximo de uma sequência finita: $max(x_1, \dots, x_n) = x_i$: x_i é o maior número da sequência

$max(x_1, \dots, x_n) = mx(\dots mx(mx(x_1, x_2), x_3), \dots, x_n)$

(11) diferença absoluta: $db(x, y) = \begin{cases} x - y, & \text{se } x \geq y \\ y - x, & \text{se } x < y \end{cases}$

$db(x, y) = sp(mx(x, y), mn(x, y))$

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

$sg(0) = 0$

$sg(S(y)) = 1$

(13) contrassinal: $\overline{sg}(x) = \begin{cases} 0, & \text{se } x \neq 0 \\ 1, & \text{se } x = 0 \end{cases}$

$\overline{sg}(0) = 1$

$\overline{sg}(S(y)) = 0$

(14) resto da divisão de y por x: $rt(x, y) = \begin{cases} y - n \cdot x : n \leq \frac{y}{x} \text{ } \text{ } \sim\exists p(n < p < \frac{y}{x}), & \text{se } x \leq y \\ y, & \text{se x} > \text{y} \end{cases}$

$rt(x, 0) = 0$

$rt(x, S(y)) = ml(S(rt(x, y)), sg(db(x, S(rt(x, y)))))$

(15) quociente da divisão de y por x: $qt(x, y) = \begin{cases} 0, & \text{se y} = 0 \\ n: n \leq \frac{y}{x} \text{ } \text{ } \sim\exists p(n < p < \frac{y}{x}), & \text{se y} \neq 0 \end{cases}$

$qt(x, 0) = 0$

$qt(x, S(y)) = ad(qt(x, y), \overline{sg}(db(x,S(rt(x, y)))))$

Prova:

Considerando o exposto, percebe-se que a prova deste teorema deverá consistir na apresentação de pelo menos uma derivação recursiva primitiva para cada uma das funções acima listadas. Tais derivações devem ser construídas de tal modo que os valores da última função para um argumento qualquer seja igual aos valores da função que se deseja provar ser recursiva primitiva para o mesmo argumento. Como ilustração, vejamos, a seguir, duas derivações recursivas primitivas[1]:

(3) Exponenciação

1. $C_1^1(x) = 1$	Função inicial (FI)
2. $C_0^1(x) = 0$	FI
3. $U_1^3(x, y, z) = x$	FI
4. $U_3^3(x, y, z) = z$	FI
5. $g_2(x, y, z) = ad(U_1^3(x, y, z), U_3^3(x, y, z))$	Função recursiva primitiva (FRP)
6. $ml(x, 0) = C_0^1(x)$ $ml(x, S(y)) = g_2(x, y, ml(x, y))$	2, 5/Recursão primitiva (RP)
7. $g_4(x, y, z) = ml(U_1^3(x, y, z), U_3^3(x, y, z))$	6, 3, 4/Composição (C)
8. $g_5(x, 0) = C_1^1(x)$ $g_5(x, S(y)) = g_4(x, y, g_5(x, y))$	1, 7/RP

[1] Todas as derivações recursivas primitivas que constituem a prova deste teorema podem ser encontradas na obra de DIAS e WEBER. *Teoria da Recursão*, 1. ed. São Paulo: Editora UNESP, 2010. p. 31-35.

Ora, $g_5(x, y) = ep(x, y)$. Portanto, *ep* é recursiva primitiva.

(11) Diferença absoluta

1. $U_1^1(x) = x$	FI
2. $g_8(x, y, z) = pd(U_3^3(x, y, z))$	FRP
3. $sp(x, 0) = U_1^1(x)$ $sp(x, S(y)) = g_8(x, y, sp(x, y))$	1, 8/RP
4. $mn(x, y) = sp(U_1^2(x, y), sp(x, y))$	FRP
5. $mx(x, y) = ad(U_2^2(x, y), sp(x, y))$	FRP
6. $g_{10}(x, y) = sp(mx(x, y), mn(x, y))$	2, 4, 3/C

Ora, $g_{10}(x, y) = db(x, y)$. Portanto, *db* é recursiva primitiva.

Note-se que uma derivação recursiva primitiva para uma função *n*-ária *f* constitui, em termos intuitivos, um algoritmo que computa tal função para qualquer *n*-upla ordenada. Com efeito, para calcular o valor de *f* para $(m_1, \dots, m_n)$, basta instanciar as variáveis presentes na última linha da derivação, substituindo-as, uniformemente, por $m_1, \dots, m_n$ e, em seguida, calcular o valor das funções à direita da igualdade, que foram definidas em linhas anteriores, sempre partindo das funções mais internas para as mais externas.

Convenção notacional

- Com o objetivo de facilitar a leitura de algumas funções que serão apresentadas mais adiante, escreveremos, em alguns momentos, $x \dot{-} y$, $|x - y|$ e $x \circ y$ para expressar, respectivamente, $sp(x, y)$, $db(x, y)$ e $rt(x, y)$.

1.1.1 Somas e produtos limitados

Nesta subseção, apresentaremos duas operações funcionais - a soma e o produto limitados. Logo em seguida, provaremos que a aplicação de tais operações a funções recursivas primitivas conduz também a funções recursivas primitivas.

Definição 1.14 A *soma* e *o produto limitados* são definidos da seguinte maneira:

(1) Soma (para $n \geq 0$):

$$\sum_{y<z} f(x_1, \dots, x_n, y) = \begin{cases} 0, & \text{se } z = 0 \\ f(x_1, \dots, x_n, 0) + \dots + f(x_1, \dots, x_n, z-1), & \text{se } z > 0 \end{cases}$$

(2) Produto (para $n \geq 0$):

$$\prod_{y<z} f(x_1, \dots, x_n, y) = \begin{cases} 1, & \text{se } z = 0 \\ f(x_1, \dots, x_n, 0) \cdot \dots \cdot f(x_1, \dots, x_n, z-1), & \text{se } z > 0 \end{cases}$$

Teorema 1.15 Se $f(x_1, \dots, x_n, y)$ é recursiva primitiva, então as seguintes funções são recursivas primitivas:

(1) $\sum_{y<z} f(x_1, \dots, x_n, y)$

(2) $\prod_{y<z} f(x_1, \dots, x_n, y)$

Prova[2]*:*

Considere a seguinte função recursiva primitiva:

$$g_{31}(x_1, \dots, x_n, 0) = h_3(x_1, \dots, x_n)$$

$$g_{31}(x_1, \dots, x_n, \mathrm{S(z)}) = h_5(x_1, \dots, x_n, z, g_{31}(x_1, \dots, x_n, \mathrm{z})),$$

onde $h_3(x_1, \dots, x_n) = \mathrm{C}_0^1(\mathrm{U}_1^n(x_1, \dots, x_n))$ e $h_5(x_1, \dots, x_n, y, w) = ad(f(\mathrm{U}_1^{n+2}(x_1, \dots, x_n, y, w), \dots, \mathrm{U}_{n+1}^{n+2}(x_1, \dots, x_n, y, w)), \mathrm{U}_{n+2}^{n+2}(x_1, \dots, x_n, y, w))$.

Expresso de outra forma, $g_{31}(x_1, \dots, x_n, \mathrm{S}(z)) = h_5(x_1, \dots, x_n, z, h_5(x_1, \dots, x_n, z\text{-}1, \dots, h_5(x_1, \dots, x_n, z\text{-}z, g_{31}(x_1, \dots, x_n, z\text{-}z))))$. Ora, $g_{31}(x_1, \dots, x_n, z\text{-}z) = 0$ e, tal como foi definida, a função h_5 somará $g_{31}(x_1, \dots, x_n, z\text{-}z)$ a $f(x_1, \dots, x_n, z\text{-}z)$ e, assim, obteremos $g_{31}(x_1, \dots, x_n, 1)$ que, por sua vez, será somado a $f(x_1, \dots, x_n, 1)$, resultando em $g_{31}(x_1, \dots, x_n, 2)$. Este processo se repetirá até que h_5 some $g_{31}(x_1, \dots, x_n, z)$ a $f(x_1, \dots, x_n, z)$. O valor desta última soma será $g_{31}(x_1, \dots, x_n, \mathrm{S}(z))$. Esquematicamente, nós temos:

$g_{31}(x_1, \dots, x_n, z\text{-}z) = 0$

$g_{31}(x_1, \dots, x_n, z\text{-}z) + f(x_1, \dots, x_n, z\text{-}z) = g_{31}(x_1, \dots, x_n, 1)$

$g_{31}(x_1, \dots, x_n, 1) + f(x_1, \dots, x_n, 1) = g_{31}(x_1, \dots, x_n, 2)$

$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$

$g_{31}(x_1, \dots, x_n,, z) + f(x_1, \dots, x_n, z) = g_{31}(x_1, \dots, x_n, \mathrm{S}(z))$

Definida nestes termos, $g_{31}(x_1, \dots, x_n, z) = \sum_{y<z} f(x_1, \dots, x_n, y)$. Portanto, $\sum_{y<z} f(x_1, \dots, x_n, y)$ é recursiva primitiva.

1.1.2 Relações numéricas

Além das funções numéricas, podemos também definir relações numéricas como recursivas primitivas. Uma relação numérica *n*-ária ($n \geq 0$) é qualquer subconjunto de $\mathbb{N}^n$.

[2]Por questões práticas, provaremos apenas que $\sum_{y<z} f(x_1, \dots, x_n, y)$ é recursiva primitiva. Analogamente, prova-se o mesmo acerca da função $\prod_{y<z} f(x_1, \dots, x_n, y)$.

Sendo assim, $\{(x, y, z): x + y = z\}$ é uma relação ternária, enquanto que {(1, 3, 5, 7)} e $\{(x, y, w, z): x + y = w - z\}$ são relações quaternárias.

Definição 1.16 Seja R uma relação *n*-ária, a *função característica* de R é a seguinte:

$$\chi_R(x_1, \dots, x_n) = \begin{cases} 1, \text{se } (x_1, \dots, x_n) \in \text{R} \\ 0, \text{se } (x_1, \dots, x_n) \notin \text{R} \end{cases}$$

Em particular, se R é uma relação unária, R ⊆ ℕ e sua função característica é a seguinte:

$$\chi_R(x) = \begin{cases} 1, \text{se } x \in \text{R} \\ 0, \text{se } x \notin \text{R} \end{cases}$$

Definição 1.17 Uma relação *n*-ária R é recursiva primitiva se, e somente se, $\chi_R(x_1, \dots, x_n)$ é recursiva primitiva.

Teorema 1.18 As seguintes relações são recursivas primitivas: *x* é igual a *y*, *x* é menor que *y*, *x* divide *y*, *x* é metade de *y*, *x* é ímpar, *x* é primo.

Prova:

(1) *x* é igual a *y* se, e somente se, $|x - y| = 0$.

Então, $\chi_=(x, y) = \overline{sg}(|x - y|)$

(2) *x* é menor que *y* se, e somente se, $x \dot{-} y = 0$ e $|x - y| \neq 0$.

Então, $\chi_<(x, y) = \overline{sg}(x \dot{-} y) \cdot sg(|x - y|)$

(3) *x* divide *y* se, e somente se, $x \circ y = 0$.

Então, $\chi_|(x, y) = \overline{sg}(x \circ y)$

(4) *x* é metade de *y* se, e somente se, $x \circ y = 0$ e $|qt(x, y) - 2| = 0$.

Então, $\chi_{mt}(x, y) = \overline{sg}(x \circ y) \cdot \overline{sg}(|qt(x, y) - \text{C}_2^2(x, y)|)$

(5) *x* é ímpar se, e somente se, $2 \circ x \neq 0$.

Então, $\chi_{im}(x) = sg(\text{C}_2^1(x) \circ \text{U}_1^1(x))$

(6) *x* é primo se, e somente se, $|d(x) - 2| = 0$, sendo $d(x)$ o número de divisores de *x* determinado pela função parcial recursiva $\sum_{y<x+1} \overline{sg}(y \circ x)$.

Então, $\chi_{pr}(x) = \overline{sg}(|d(x) - \text{C}_2^1(x)|)$

Com o auxílio das operações de complemento, união e interseção, dos conectivos proposicionais, dos quantificadores limitados e da operação de minimização limitada, podemos obter novas relações recursivas primitivas a partir de outras já disponíveis. Por questões práticas, definiremos, em momento oportuno, apenas os quantificadores limitados e a operação de minimização limitada.

Teorema 1.19 Se $R(x_1, \dots, x_n)$ e $S(x_1, \dots, x_n)$ são relações recursivas primitivas, então as seguintes relações são recursivas primitivas:

(1) $\overline{R}(x_1, \dots, x_n)$

(2) $R(x_1, \dots, x_n) \cup S(x_1, \dots, x_n)$

(3) $R(x_1, \dots, x_n) \cap S(x_1, \dots, x_n)$

Prova:

Considerando a definição das operações de complemento, união e interseção, apresentamos, a seguir, as funções características de $\overline{R}$, $R \cup S$ e $R \cap S$.

(1)

$$\chi_{\overline{R}}(x_1, \dots, x_n) = \begin{cases} 1, \text{se } (x_1, \dots, x_n) \in \overline{R} \\ 0, \text{se } (x_1, \dots, x_n) \notin \overline{R} \end{cases}$$

$$= \begin{cases} 1, \text{se } (x_1, \dots, x_n) \notin R \\ 0, \text{se } (x_1, \dots, x_n) \in R \end{cases}$$

$$= \begin{cases} 1, \text{se } \chi_R(x_1, \dots, x_n) = 0 \\ 0, \text{se } \chi_R(x_1, \dots, x_n) = 1 \end{cases}$$

(2)

$$\chi_{R \cup S}(x_1, \dots, x_n) = \begin{cases} 1, \text{se } (x_1, \dots, x_n) \in R \cup S \\ 0, \text{se } (x_1, \dots, x_n) \notin R \cup S \end{cases}$$

$$= \begin{cases} 1, \text{se } (x_1, \dots, x_n) \in R \text{ ou se } (x_1, \dots, x_n) \in S \\ 0, \text{se } (x_1, \dots, x_n) \notin R \text{ e se } (x_1, \dots, x_n) \notin S \end{cases}$$

$$= \begin{cases} 1, \text{se } \chi_R(x_1, \dots, x_n) = 1 \text{ ou se } \chi_S(x_1, \dots, x_n) = 1 \\ 0, \text{se } \chi_R(x_1, \dots, x_n) = 0 \text{ e se } \chi_S(x_1, \dots, x_n) = 0 \end{cases}$$

(3)

$$\chi_{R \cap S}(x_1, \dots, x_n) = \begin{cases} 1, \text{se } (x_1, \dots, x_n) \in R \cap S \\ 0, \text{se } (x_1, \dots, x_n) \notin R \cap S \end{cases}$$

$$= \begin{cases} 1, \text{se } (x_1, \dots, x_n) \in R \text{ e se } (x_1, \dots, x_n) \in S \\ 0, \text{se } (x_1, \dots, x_n) \notin R \text{ ou se } (x_1, \dots, x_n) \notin S \end{cases}$$

$$= \begin{cases} 1, \text{se } \chi_R(x_1, \dots, x_n) = 1 \text{ e se } \chi_S(x_1, \dots, x_n) = 1 \\ 0, \text{se } \chi_R(x_1, \dots, x_n) = 0 \text{ ou se } \chi_S(x_1, \dots, x_n) = 0 \end{cases}$$

Em outros termos, as funções acima apresentadas podem ser definidas do seguinte modo:

$\chi_{\overline{R}}(x_1, \dots, x_n) = \overline{sg}(\chi_R(x_1, \dots, x_n))$

$\chi_{R \cup S}(x_1, \ldots, x_n) = sg(ad(\chi_R(x_1, \ldots, x_n), \chi_S(x_1, \ldots, x_n)))$

$\chi_{R \cap S}(x_1, \ldots, x_n) = ml(\chi_R(x_1, \ldots, x_n), \chi_S(x_1, \ldots, x_n))$

Portanto, conforme a definição 1.17, as relações *n*-árias, $\overline{R}$, $R \cup S$ e $R \cap S$ são recursivas primitivas.

Teorema 1.20 Se $R(x_1, \ldots, x_n)$ e $S(x_1, \ldots, x_n)$ são relações recursivas primitivas, então as seguintes relações são recursivas primitivas:

(1) $\sim R(x_1, \ldots, x_n)$
(2) $R(x_1, \ldots, x_n) \wedge S(x_1, \ldots, x_n)$
(3) $R(x_1, \ldots, x_n) \vee S(x_1, \ldots, x_n)$
(4) $R(x_1, \ldots, x_n) \rightarrow S(x_1, \ldots, x_n)$
(5) $R(x_1, \ldots, x_n) \leftrightarrow S(x_1, \ldots, x_n)$

Prova:

Imediata pela definição dos conectivos proposicionais.

Definição 1.21 Os *quantificadores limitados* são obtidos, por definição, a partir dos quantificadores existencial e universal, conforme vemos abaixo:

(1) Quantificador existencial (para $n \geq 0$):

$$\bigvee_{y<z} R(x_1, ..., x_n, y) \stackrel{\text{def}}{=} \exists_y (0 \leq y < z \wedge (x_1, ..., x_n, y) \in R)$$

(2) Quantificador universal (para $n \geq 0$):

$$\bigwedge_{y<z} R(x_1, ..., x_n, y) \stackrel{\text{def}}{=} \forall_y (0 \leq y < z \rightarrow (x_1, ..., x_n, y) \in R)$$

Teorema 1.22 Se $R(x_1, \ldots, x_n, y)$ é uma relação recursiva primitiva, então as relações abaixo são recursivas primitivas:

(1)$\bigvee_{y<z} R(x_1, ..., x_n, y)$

(2)$\bigwedge_{y<z} R(x_1, ..., x_n, y)$

Prova[3]*:*

Considere $\mathrm{M}(x_1, \ldots, x_n, z) = \bigvee_{y<z} R(x_1, ..., x_n, y)$.

[3] Exibiremos unicamente a prova de que $\bigvee_{y<z} R(x_1, ..., x_n, y)$ é recursiva primitiva. De modo semelhante, demonstra-se o mesmo acerca da relação $\bigwedge_{y<z} R(x_1, ..., x_n, y)$, cuja função caraterística é $\prod_{y<z} \chi_R(x_1, \ldots, x_n, y)$.

Segundo a definição 1.21, $\bigvee_{y<z}R(x_1, ..., x_n, y) = R(x_1, ..., x_n, 0) \vee ... \vee R(x_1, ..., x_n, z\text{-}1)$.
Consequentemente, $\mathrm{M}(x_1, \dots, x_n, z) = R(x_1, ..., x_n, 0) \vee ... \vee R(x_1, ..., x_n, z\text{-}1)$.
Neste caso,

$$\chi_M(x_1, \dots, x_n, z) = \begin{cases} 1, \text{se } (x_1, \dots, x_n, y) \in \mathrm{R}, \text{para algum y} < \mathrm{z} \\ 0, \text{se } (x_1, \dots, x_n, y) \notin \mathrm{R}, \text{para todo y} < \mathrm{z} \end{cases}$$
$$= \begin{cases} 1, \text{se } \chi_R(x_1, \dots, x_n, y) = 1, \text{para algum y} < \mathrm{z} \\ 0, \text{se } \chi_R(x_1, \dots, x_n, y) = 0, \text{para todo y} < \mathrm{z} \end{cases}$$

De outro modo, $\chi_M(x_1, \dots, x_n, z) = sg(\sum_{y<z} \chi_R(x_1, \dots, x_n, y))$. Portanto, tendo em vista a nossa consideração inicial, $\bigvee_{y<z}R(x_1, ..., x_n, y)$ é recursiva primitiva.

Definição 1.23 A operação de *minimização limitada* aplicada a uma relação *n*+1-ária *R* é estabelecida do seguinte modo:

$$\mu_{y<z}R(x_1, ..., x_n, y) = \begin{cases} \text{o menor } y < z : (x_1, \dots, x_n, y) \in \mathrm{R}, & \text{se } \bigvee_{y<z}R(x_1, \dots, x_n, y) \\ 0, & \text{se } \sim\bigvee_{y<z}R(x_1, \dots, x_n, y) \end{cases}$$

Intuitivamente, a minimização limitada é uma operação de pesquisa. Tendo em vista os limites impostos a sua aplicação, a pesquisa que ela faz, mais cedo ou mais tarde, chega ao fim, tendo sempre como valor o menor *y* tal que $(x_1, \dots, x_n, y) \in \mathrm{R}$, caso ele, de fato, exista, ou 0, caso contrário.

Teorema 1.24 Se $R(x_1, \dots, x_n, y)$ é uma relação recursiva primitiva, então a função $\mu_{y<z}R(x_1, ..., x_n, y)$ é recursiva primitiva.

Prova:

Se não existe um $y < z$ tal que $R(x_1, \dots, x_n, y)$, $sg(\sum_{y<z} \chi_R(x_1, \dots, x_n, y)) = 0$. Caso contrário, $sg(\sum_{y<z} \chi_R(x_1, \dots, x_n, y)) = 1$, e o menor $y < z$ será o valor determinado pela função $\sum_{y<z} \prod_{u<y+1} \overline{sg}(\chi_R(x_1, \dots, x_n, u))$ que a cada $\chi_R(x_1, \dots, x_n, u) = 0$ soma 1 até alcançar a primeira *n*+1-upla $(x_1, \dots, x_n, u)$ tal que $\chi_R(x_1, \dots, x_n, u) = 1$. Sendo assim, a função $\mu_{y<z}R(x_1, ..., x_n, y)$ é claramente recursiva primitiva, pois é obtida por composição, como vemos abaixo:

$$\mu_{y<z}R(x_1, ..., x_n, y) = ml(sg(\sum_{y<z} \chi_R(x_1, \dots, x_n, y)), \sum_{y<z} \prod_{u<y+1} \overline{sg}(\chi_R(x_1, \dots, x_n, u)))$$

Como ilustração, vamos calcular $\mu_{y<3} mt(1, y)$. Sabemos, de antemão, que um tal y existe. Portanto, até o momento, $\mu_{y<3} mt(1, y) = ml(1, \sum_{y<3} \prod_{u<y+1} \overline{sg}(\chi_{mt}(1,u)))$. Nosso próximo passo será identificá-lo calculando o somatório limitado do seguinte modo:

$$\sum_{y<3} \prod_{u<y+1} \overline{sg}(\chi_{mt}(1,u)) = \prod_{u<0+1} \overline{sg}(\chi_{mt}(1,u)) + \prod_{u<1+1} \overline{sg}(\chi_{mt}(1,u)) + \prod_{u<2+1} \overline{sg}(\chi_{mt}(1,u))$$

De outro modo:

$$\sum_{y<3} \prod_{u<y+1} \overline{sg}(\chi_{mt}(1,u)) = \overline{sg}(\chi_{mt}(1,0)) + \overline{sg}(\chi_{mt}(1,0)) \cdot \overline{sg}(\chi_{mt}(1,1)) + \overline{sg}(\chi_{mt}(1,0)) \cdot \overline{sg}(\chi_{mt}(1,1)) \cdot \overline{sg}(\chi_{mt}(1,2))$$

Portanto, como queríamos, $\mu_{y<3} mt(1, y) = ml(1, 2) = 2$. Com efeito, sabemos que 2 é o menor e único número do qual 1 é metade.

Exibiremos, a seguir, algumas funções recursivas primitivas determinadas a partir das operações de minimização, soma e produto limitados.

(1) Para $x \geq 0$, a função $p(x)$ determina o x–ésimo número primo em ordem crescente. Ela será definida levando-se em consideração o teorema de Euclides sobre a infinitude dos primos segundo o qual se p é primo, então existe um primo y tal que $p < y \leq p! + 1$. Isto posto, considerando $A = \{(x, y): p(x) < y \wedge Pr(y)\}$, definimos a função $p(x)$ como segue:

$$p(0) = 2$$
$$p(\mathrm{S}(x)) = \mu_{y<p(x)!+2} A(x, y)$$

(2) Seja $p_0^{a_0} \cdot p_1^{a_1} \cdot \ldots \cdot p_n^{a_n}$ a decomposição de x em fatores primos. A função binária $(x)_i$ determina o expoente do i-ésimo fator primo dessa decomposição, isto é, $(x)_i = a_i$. Por convenção, $(0)_i = 0$, para todo i. Note que na decomposição de x, o expoente de p_i é o (menor) número $y < x$ tal que p_i^y divide x e p_i^{y+1} não divide x. Sendo assim, considerando a relação $B = \{(x, i, y): p_i^y | x \wedge \sim(p_i^{y+1} | x)\}$, definimos a função $(x)_i$ da seguinte maneira:

$$(x)_i = \mu_{y<x} B(x, i, y)$$

(3) Para $x \neq 0$, a função *lh*(*x*) determina o número de expoentes diferentes de zero na decomposição de *x* em fatores primos ou, em outros termos, o número de primos distintos que dividem *x*. Por convenção, *lh*(0) = 0. Considerando o exposto, é fácil notar que o valor de *lh*(*x*) será obtido a partir da relação $C = \{(x, y): Pr(y) \wedge y | x \wedge x \neq 0\}$, como vemos abaixo:

$$lh(x) = \sum_{y<x+1} \chi_C(x, y)$$

(4) Seja $x = 2^{a_0} \cdot 3^{a_1} \cdot \ldots \cdot p_k^{a_k}$ e seja $y = 2^{b_0} \cdot 3^{b_1} \cdot \ldots \cdot p_m^{b_m}$, a função $x * y$ determina o valor da sequência obtida pela justaposição de x e y como vemos a seguir:

$$x * y = 2^{a_0} \cdot 3^{a_1} \cdot \ldots \cdot p_k^{a_k} \cdot p_{k+1}^{b_0} \cdot p_{k+2}^{b_1} \cdot \ldots \cdot p_{k+1+m}^{b_m}$$

Ou seja:

$$x * y = x \cdot \prod_{i<lh(y)} p_{lh(x)+i}^{(y)_i}$$

(5) Para $x \geq 0$, a função $q(x) = x^2$, estabelece o x–ésimo quadrado perfeito em ordem crescente ao passo que função $\lfloor \sqrt{x} \rfloor$ determina o maior número natural $y \leq \sqrt{x}$. Considerando $E = \{(x, y): q(y) > x\}$, definimos $[\sqrt{x}\,]$ da seguinte maneira:

$$\lfloor \sqrt{x} \rfloor = pd(\mu_{y<x+1} \mathrm{E}(x, y))$$

(6) A função $MDC(x_1, \ldots, x_n)$ determina o maior divisor comum de $x_1, \ldots, x_n$. Considerando $F = \{(x_1, \ldots, x_n, y): (y|x_1 \wedge \ldots \wedge y|x_n) \wedge \sim(y + 1|x_1)\}$, definimos $MDC(x_1, \ldots, x_n)$ abaixo:

$$MDC(x_1, \ldots, x_n) = \mu_{y<min(x_1,\ldots, x_n)+1} F(x_1, \ldots, x_n, y)$$

Concluindo esta subseção, queremos esclarecer que escolhemos destacar o conjunto das funções recursivas primitivas, pois acreditou-se, durante algum tempo, que ele seria a versão formal para o conjunto das funções algorítmicas, tendo em vista que, como falamos, anteriormente, muitas funções reconhecidamente algorítmicas foram provadas ser recursivas primitivas. No entanto, o matemático alemão Wilhelm Ackermann apresentou um contraexemplo. Ele construiu uma função algorítmica que não era recursiva primitiva. A partir daí, surgiu, então, a necessidade de se ampliar o conjunto das funções recursivas primitivas, a fim de se obter um conjunto mais abrangente de funções que constituísse uma versão formal para a noção intuitiva de função algorítmica. Como resultado desta ampliação, definiu-se o conjunto das funções recursivas. Para compreender também as funções

estritamente parciais algorítmicas, o conjunto das funções recursivas foi mais uma vez ampliado, o que resultou no conjunto das funções parciais recursivas.

1.1.3 Derivações parciais recursivas

Na subseção anterior, apresentamos resultados envolvendo unicamente funções recursivas primitivas que, como vimos, são alcançadas a partir das funções iniciais e das operações de recursão primitiva e composição. Falta-nos ainda apresentar, na prática, o uso da operação de minimização ilimitada, a partir da qual obtemos funções (parciais) recursivas. Com tal intuito, exibiremos o próximo teorema, cuja prova exigirá a construção de derivações (parciais) recursivas para cada uma das funções listadas.

Teorema 1.25 As seguintes funções são parciais recursivas:

(1) zero-zero: $zz(x) = \begin{cases} 0, & \text{se } x = 0 \\ \uparrow, & \text{caso contrário} \end{cases}$

$zz(x) = \mu_y(mx(x, y) = 0)$

(2) função vazia: $\emptyset(x_1, \dots, x_n) = \uparrow$, para qualquer $x_1, \dots, x_n$

$\emptyset(x_1, \dots, x_n) = \mu_y((\mathrm{S}(\mathrm{U}_1^{n+1}(x_1, \dots, x_n, y))) = 0)$

Prova:

(1) Zero-zero

1. $\mathrm{U}_2^2(x, y) = y$	FI
2. $g_0(x, y, z) = \mathrm{S}(\mathrm{U}_3^3(x, y, z))$	FRP
3. $ad(x, 0) = \mathrm{U}_1^1(x)$ $ad(x, \mathrm{S}(y)) = g_0(x, y, ad(x, y))$	FRP
4. $sp(x, 0) = \mathrm{U}_1^1(x)$ $sp(x, \mathrm{S}(y)) = g_8(x, y, sp(x, y))$	FRP
5. $mx(x, y) = ad(\mathrm{U}_2^2(x, y), sp(x, y))$	3, 1, 4/C
6. $g_{11}(x) = \mu_y(mx(x, y) = 0)$	5/Minimização ilimitada (MI)

Ora, $g_{11}(x) = zz(x)$. Portanto, *zz* é parcial recursiva.

(2) Função vazia

1.$\mathrm{S}(x) = x'$	FI
2.$\mathrm{U}_1^{n+1}(x_1, \dots, x_n, y) = x_1$	FI

3. $g_{12}(x_1, \dots, x_n, y)$ = S($U_1^{n+1}(x_1, \dots, x_n, y)$) 1,2/C
4. $g_{13}(x_1, \dots, x_n) = \mu_y(g_{12}(x_1, \dots, x_n, y) = 0)$ 3/MI

Ora, $g_{13}(x_1, \dots, x_n) = \emptyset(x_1, \dots, x_n)$. Portanto, $\emptyset$ é parcial recursiva.

Similarmente ao que vimos na subseção precedente, uma derivação (parcial) recursiva para uma função *f* é um algoritmo que computa tal função. Sabendo disso, vale ressaltar que embora as derivações recursivas primitivas e as derivações (parciais) recursivas sejam exemplos de algoritmos, apenas o conjunto das derivações recursivas primitivas e o conjunto das derivações parciais recursivas são decidíveis, pois dada uma sequência qualquer de funções, é possível reconhecer mecanicamente se cada função da sequência é uma função inicial ou se foi obtida de funções anteriores, a partir das operações de composição, recursão primitiva ou minimização ilimitada. No entanto, o mesmo não pode ser afirmado sobre o conjunto das derivações recursivas. Com efeito, a restrição imposta à aplicação da minimização ilimitada às funções regulares nos impede de identificarmos mecanicamente se uma dada sequência de funções é ou não uma derivação recursiva, já que para decidir se uma função $f(x_1, \dots, x_n, y)$ é regular é necessário calcular, para cada $(x_1, \dots, x_n)$, os valores $f(x_1, \dots, x_n, 0)$, $f(x_1, \dots, x_n, 1)$, $f(x_1, \dots, x_n, 2)$ etc., admitindo a real possibilidade deste cálculo nunca terminar, pois pode ser que para esta n-upla não haja um y tal que $f(x_1, \dots, x_n, y) = 0$. Portanto, em resumo, dada uma sequência qualquer de funções nunca saberemos, em geral, se ela é de fato uma derivação recursiva, por não sabermos, previamente, se uma de suas funções a qual foi aplicada a minimização ilimitada é regular.

Além das funções (parciais) recursivas, podemos também estabelecer relações (parciais) recursivas. Estas são definidas de modo análogo às relações recursivas primitivas. Por fim, posto que toda função recursiva primitiva é recursiva e toda função recursiva é parcial recursiva, convém explicitar, de antemão, que os teoremas vistos nas subseções 1.1.1 e 1.1.2 se aplicam, sem maiores dificuldades, às funções e às relações (parciais) recursivas.

2 TURING-COMPUTABILIDADE

Definido o conjunto das funções parciais recursivas, definiremos, neste capítulo, o conjunto das funções parcialmente Turing-computáveis, a fim de estabelecermos a igualdade entre eles, que é, como já dissemos, um dos nossos objetivos. Para isso, apresentaremos, previamente, as máquinas e os programas de Turing. Estes serão identificados como conjuntos de quíntuplas, o que nos possibilitará reconstruirmos, de modo inédito, a prova segundo a qual toda função parcial recursiva é parcialmente Turing-computável. A recíproca também será provada. Ao final, apresentaremos alguns argumentos a favor da Tese de Church-Turing que identifica as funções parciais algorítmicas às funções parcialmente Turing-computáveis.

2.1 Máquinas e programas de Turing

Com o objetivo de formular uma definição precisa de computabilidade efetiva, Turing definiu certos objetos teóricos que ficaram conhecidos como máquinas de Turing, a partir dos quais define-se de maneira exata o conjunto das funções parcialmente Turing-computáveis. Intuitivamente, entende-se por uma *máquina de Turing* M um mecanismo imaginário constituído por um dispositivo chamado *reading head* e por uma fita infinita à esquerda e à direita, secionada em quadrados, tal que:

(1) em cada quadrado está escrito apenas um símbolo da fita s_i do conjunto S = $\{s_0, s_1, s_2, ...\}$.

(2) o dispositivo sempre está em um estado interno q_i do conjunto Q = $\{q_0, q_1, q_2, ...\}$, sempre observa um quadrado da fita por vez e executa um dos seguintes movimentos: move-se para o quadrado imediatamente à direita daquele que está sendo observado (movimento que simbolizaremos por 'R', de *right*) ou move-se para o quadrado imediatamente à esquerda daquele que está sendo observado (movimento que simbolizaremos por 'L', de *left*).

(3) as ações do dispositivo são determinadas por um conjunto finito não-vazio $P \subset Q \times S \times S \times \{R, L\} \times Q$ que chamaremos de programa de Turing. Se $(q_i, s_k, s_t, x, q_l) \in P$, então o dispositivo, no estado interno q_i e observando o quadrado no qual está escrito s_k, substitui s_k por s_t, move-se para o quadrado imediatamente à direita, quando x = R (ou à esquerda, quando x = L) e assume o estado interno q_l.

Definição 2.1 A *linguagem* L de uma máquina de Turing é o conjunto de símbolos $S \cup Q \cup \{R, L\}$.

Definição 2.2 Uma *expressão* de L é uma sequência finita de símbolos de L.

Definição 2.3 Duas quíntuplas do conjunto $Q \times S \times S \times \{R, L\} \times Q$ são *inconsistentes* se, e somente se, são iguais quanto aos dois primeiros símbolos e diferentes quanto a, pelo menos, um dos símbolos restantes. De outro modo, são *consistentes*.

Definição 2.4 Um *programa de Turing* P é um conjunto finito não-vazio de quíntuplas consistentes de $Q \times S \times S \times \{R, L\} \times Q$.

A restrição presente na definição acima é chamada de "requisito de consistência". Ela evita comandos contraditórios, diante dos quais a máquina interromperia a sua computação. Observando tal requisito, as quíntuplas (q_0, s_1, s_1, L, q_1) e (q_0, s_1, s_1, L, q_2), por exemplo, não são admitidas em um programa de Turing.

Normalmente, a maioria dos manuais define um programa de Turing como um conjunto finito de quádruplas consistentes. Nós, no entanto, optamos por defini-lo como um conjunto finito de quíntuplas consistentes. Desta forma, obteremos, em geral, programas de menor cardinalidade, tendo em vista que, muitas vezes, o comando dado por duas quádruplas pode ser determinado por uma única quíntupla. As quádruplas (q_i, s_k, s_t, q_l) e (q_i, s_t, R, q_l), por exemplo, podem ser substituídas, sem maiores dificuldades, pela quíntupla (q_i, s_k, s_t, R, q_l). Este modo alternativo de definir um programa de Turing nos permitirá provarmos, mais adiante, o enunciado segundo o qual toda função parcial recursiva é parcialmente Turing-computável de modo ligeiramente diverso do habitual, pois conseguiremos prová-lo utilizando programas menores, oportunizando, consequentemente, computações com um menor número de passos.

Definição 2.5 Uma *configuração instantânea* de M é uma expressão de L do tipo aq_ib, tal que *a* e *b* são, respectivamente, uma sequência finita (possivelmente vazia) e uma sequência finita (não vazia), de símbolos da fita.

Sendo assim, as expressões $q_0s_1s_1$ e $s_0s_1q_5s_3$ constituem, por exemplo configurações instantâneas de M; no entanto, o mesmo não pode ser dito das expressões $Rs_1q_5s_3$ e $s_0s_1q_5$.

Observação

- Dada uma configuração instantânea c de M, assumimos os seguintes enunciados: (1) nos quadrados da fita não referidos em c está escrito s_0; (2) o símbolo da fita

observado pelo dispositivo é aquele que segue imediatamente o símbolo de estado interno.

Definição 2.6 Dados um programa de Turing P e as configurações instantâneas c e c' de M, c *acarreta* c' via P (em símbolos: $c \underset{P}{\rightarrow} c'$) se, e somente se, uma das seguintes condições é satisfeita, sendo t_1 e t_2 sequências finitas (possivelmente vazias) de símbolos da fita:

(1) $(q_i, s_k, s_t, R, q_l) \in P$, $c = t_1 q_i s_k s_p t_2$ e $c' = t_1 s_t q_l s_p t_2$

(2) $(q_i, s_k, s_t, R, q_l) \in P$, $c = t_1 q_i s_k$ e $c' = t_1 s_t q_l s_0$

(3) $(q_i, s_k, s_t, L, q_l) \in P$, $c = t_1 s_p q_i s_k t_2$ e $c' = t_1 q_l s_p s_t t_2$

(4) $(q_i, s_k, s_t, L, q_l) \in P$, $c = q_i s_k t_2$ e $c' = q_l s_0 s_t t_2$

Em outras palavras, a primeira condição é a seguinte: sendo (q_i, s_k, s_t, R, q_l) uma quíntupla de P e sendo $c = t_1 q_i s_k s_p t_2$, isto é, estando o dispositivo no estado interno q_i e observando o quadrado no qual está escrito s_k, ele substitui s_k por s_t, movimenta-se ao quadrado imediatamente à direita e assume o estado interno q_l, resultando em c'. As demais condições são análogas a esta que acabamos de explicitar.

Definição 2.7 Uma configuração instantânea c é *terminal* com respeito a um programa de Turing P (em símbolos: c_t^P) se, e somente se, $c = t_1 q_i s_k t_2$ e P não contém quíntuplas da forma (q_i, s_k, s_t, x, q_l).

Convenção notacional

- Assim como fizemos com as derivações parciais recursivas, utilizaremos os programas de Turing para realizarmos computações numéricas. Para tanto, são necessárias algumas convenções:
  (1) No lugar de s_0, escreveremos B para indicar um quadrado vazio durante a computação; no lugar de s_1, escreveremos | e, no lugar dos demais s_i (desde que haja a necessidade de usá-los), escreveremos alguns marcadores, que serão apresentados mais adiante.
  (2) Para representar um número natural x que ocorre no *input*, escreveremos | em x + 1 quadrados consecutivos. O *output* y será representado por y |'s escritos em quadrados não necessariamente consecutivos da configuração instantânea terminal.

Representaremos x + 1 |'s por $\overline{x}$ ou $|^{x+1}$. Em geral, para representar uma n-upla $(m_1, \ldots, m_n)$, escreveremos $\overline{m}_1$B...B$\overline{m}_n$.

- Eliminaremos os pares de parênteses e as vírgulas presentes nas quíntuplas. Escreveremos $q_i s_k s_t x q_l$, em vez de (q_i, s_k, s_t, x, q_l).

Definição 2.8 Uma *computação segundo um programa de Turing* P *com input* $(m_1, \ldots, m_n)$ é uma sequência finita de configurações instantâneas de M, $(c_1, c_2, ..., c_k)$, tal que a configuração instantânea inicial $c_1 = q_0\overline{m}_1$B...B$\overline{m}_n$, $c_k = c_t^P$ e $c_i \underset{P}{\rightarrow} c_{i+1}$, para cada $1 \leq i < k$.

Como ilustração, considere o programa de Turing P = $\{q_0|BRq_1, q_1||Lq_1\}$ e a seguinte sequência de configurações instantâneas que constitui uma computação segundo o programa P com o *input* (2,0):

c_1: $q_0|||B|$
c_2: $Bq_1||B|$
c_3: $q_1B||B|$

Definição 2.9 Dados um programa de Turing P e as configurações instantâneas c e c' de M, c' é *resultante* de c com respeito a P (em símbolos: c' = $Res_P(c)$) se e somente se existe uma computação $(c_1, c_2, ..., c_k)$ segundo P com *input* $(m_1, \ldots, m_n)$, tal que c_1 = c e c_k = c'.

Definição 2.10 Seja P um programa de Turing, a ele está associado, para cada $n \geq 1$, uma única função n-ária $\Psi_P^n(x_1, \ldots, x_n)$ tal que dado uma n-upla $(m_1, \ldots, m_n)$ duas situações alternativas podem acontecer:

(1) há uma computação $(c_1, c_2, ..., c_k)$ segundo P com *input* $(m_1, \ldots, m_n)$; neste caso, $\Psi_P^n(m_1, \ldots, m_n) = [c_k]$, onde $[c_k]$ é o número de |'s que ocorrem na descrição instantânea terminal c_k.

(2) não há uma computação $(c_1, c_2, ..., c_k)$ segundo P com *input* $(m_1, \ldots, m_n)$; neste caso, $\Psi_P^n(m_1, \ldots, m_n)$ e, consequentemente, $Res_P(q_0\overline{m}_1$B...B$\overline{m}_n)$ estão indefinidos.

Definição 2.11 Uma função numérica n-ária $h(x_1, \ldots, x_n)$ é *parcialmente Turing-computável* se, e somente se, existe um programa de Turing P tal que $h(x_1, \ldots, x_n) = \Psi_P^n(x_1, \ldots, x_n)$. Em particular, se $h(x_1, \ldots, x_n)$ é uma função total, dizemos que $h(x_1, \ldots, x_n)$ é *Turing-computável*.

Em outras palavras, uma função é (parcialmente) Turing-computável se e somente se existe um programa de Turing para computá-la.

2.2 Equivalência entre as funções parciais recursivas e as funções parcialmente Turing-computáveis

Reservaremos esta seção para reconstruirmos a prova segundo a qual o conjunto das funções parciais recursivas e o conjunto das funções parcialmente Turing-computáveis, embora sejam conceitualmente distintos, possuem as mesmas funções numéricas como seus elementos.

2.2.1 Toda função parcial recursiva é parcialmente Turing-computável

Obviamente, para provarmos que as funções parciais recursivas são parcialmente Turing-computáveis devemos provar os seguintes enunciados:

(1) as funções iniciais são parcialmente Turing-computáveis;

(2) o conjunto das funções parcialmente Turing-computáveis é fechado com respeito às operações de composição, recursão primitiva e minimização ilimitada (ou, em outras palavras, tais operações levam de funções parcialmente Turing-computáveis a funções parcialmente Turing-computáveis).

A seguir, expomos a prova do primeiro enunciado, apresentando os programas para computar as funções iniciais.

(1.1) Uma máquina de Turing computa a função sucessor S(x) de acordo com o programa $P_S = \{q_0||Rq_1\}$. Ou seja, $\Psi^1_{P_s}(x) = S(x)$. Sabendo disso, vejamos a computação de $\Psi^1_{P_s}(3)$:

$c_1: q_0||||$

$c_2: |q_1|||$

Portanto, $\Psi^1_{P_s}(3) = [\text{Res}_{P_S}(q_0||||)] = [|q_1|||] = 4$.

(1.2) Uma máquina de Turing computa as diversas funções-constante $C^n_k(x_1, \dots, x_n)$ de acordo com o programa $P_{C^n_k}$ abaixo:

$q_0|BRq_0$ apaga todos os |'s

q_0BBRq_1

$q_1|BRq_0$

$q_iB|Rq_{i+1}$ para cada i, tal que $0 < i \leq k$
escreve um | e vai para o quadrado imediatamente à direita

Assim sendo, $\Psi^n_{P_{C^n_k}}(x_1, \ldots, x_n) = C^n_k(x_1, \ldots, x_n)$. Isto posto, vejamos, como exemplo, a computação de $\Psi^2_{P_{C^2_3}}(2,0)$:

c_1: $q_0|||B|$

c_2: $Bq_0||B|$

c_3: $BBq_0|B|$

c_4: $BBBq_0B|$

c_5: $BBBBq_1|$

c_6: $BBBBBq_0B$

c_7: $BBBBBBq_1B$

c_8: $BBBBBB|q_2B$

c_9: $BBBBBB||q_3B$

c_{10}: $BBBBBB|||q_4B$

Portanto, $\Psi^2_{P_{C^2_3}}(2,0) = [Res_{P_{C^2_3}}(q_0|||B|)] = [BBBBBB|||q_4B] = 3$.

(1.3) Uma máquina de Turing computa as diversas funções-projeção $U^n_i(x_1, \ldots, x_n)$ de acordo com o programa $P_{U^n_i}$ abaixo:

$q_{2k}|BRq_{2k+1}$ para cada k, tal que $0 \leq k \leq n\text{-}1$ e $k \neq i\text{-}1$
$q_{2k+1}|BRq_{2k+1}$ apaga um bloco de |'s
$q_{2k+1}BBRq_{2k+2}$

$q_{2i-2}|BRq_{2i-1}$ apaga o primeiro | do i-ésimo bloco
$q_{2i-1}BBRq_{2i}$
$q_{2i-1}||Rq_{2i-1}$

Consequentemente, $\Psi^n_{P_{U^n_i}}(x_1, \ldots, x_n) = U^n_i(x_1, \ldots, x_n)$. Sabendo disso, vejamos a computação de $\Psi^3_{P_{U^3_1}}(1,1,2)$:

c_1: $q_0||B||B|||$

c_2: $Bq_1|B||B|||$

c_3: $B|q_1B||B|||$

c_4: $B|Bq_2||B|||$

c_5: $B|BBq_3|B|||$

c_6: $B|BBBq_3B|||$

c_7: $B|BBBBq_4|||$

c_8: $B|BBBBBq_5||$

c_9: $B|BBBBBBq_5|$

c_{10}: $B|BBBBBBBq_5B$

c_{11}: $B|BBBBBBBBq_6B$

Portanto, $\Psi^3_{P_{U_1^3}}(1,1,2) = [Res_{P_{U_1^3}}(q_0||B||B|||)]$ = $[B|BBBBBBBBq_6B]$ = 1.

Com os programas que acabamos de expor, fica provado que as funções iniciais são parcialmente Turing-computáveis. Nossa próxima atividade será provar, a partir de uma série de lemas apresentados mais adiante, que as operações básicas, quando aplicadas a funções parcialmente Turing-computáveis, geram novas funções parcialmente Turing-computáveis.

Convenção notacional

- Dado um programa de Turing P qualquer, $\theta(P)$ designará o maior número i tal que q_i é um estado interno de P; $P^{(k)}$, por sua vez, designará o programa de Turing obtido a partir da substituição de todas as ocorrências de q_i em P por q_{i+k}.

Definição 2.12 Um programa de Turing P é n-regular ($n \geq 1$) se, e somente se, as seguintes condições são satisfeitas:

(1) sempre que $Res_P(q_0\overline{m}_1B...B\overline{m}_n)$ estiver definido, $Res_P(q_0\overline{m}_1B...B\overline{m}_n) = q_{\theta(P)}\overline{r}_1B...B\overline{r}_s$, para convenientes $r_1, \ldots, r_s$, sendo $s \geq 1$;

(2) nenhuma quíntupla de P possui $q_{\theta(P)}|$ como os dois primeiros símbolos.

Dentre os vários lemas que provaremos a partir de agora, o primeiro deles permitirá a máquina realizar uma computação e, ao final, reescrever o *output* de tal modo que ele esteja pronto para iniciar uma nova computação.

Lema 2.13 Para cada programa de Turing P, existe um programa de Turing n-regular P' tal que $Res_{P'}(q_0\overline{m}_1B...B\overline{m}_n) = q_{\theta(P')}\overline{\Psi^n_P(m_1, \ldots, m_n)}$.

Prova:

O programa de Turing P', obtido a partir de P, será construído de tal modo que a computação (principal) determinada por $P^{(k)}$ se desenvolverá entre dois marcadores: $\lambda\ (= s_2)$ e $\rho\ (= s_3)$. Caso ela exija mais espaço, os marcadores serão afastados e, com isso, novos espaços serão disponibilizados. Concluída a computação, os |'s presentes na configuração instantânea terminal serão reunidos em um único bloco, os marcadores ρ e λ serão, respectivamente, apagados e um | será adicionado ao único bloco existente, no quadrado, até então, ocupado por λ. A esta altura, a máquina terá alcançado o estado interno $q_{\theta(P')}$, observará o | mais à esquerda do bloco, estando, deste modo, pronta para iniciar uma nova computação.

Seja P_1 o seguinte programa de Turing:

$q_0\|\|Lq_0$	imprime λ à esquerda
$q_0B\lambda Rq_1$	
$q_1\|\|Rq_1$	move-se à direita até encontrar um duplo vazio
q_1BBRq_2	
$q_2\|\|Rq_1$	
q_2BBLq_3	
$q_3B\rho Lq_4$	imprime ρ à direita; move-se à esquerda até encontrar λ; em seguida, move-se um quadrado à direita
$q_4\|\|Lq_4$	
q_4BBLq_4	
$q_4\lambda\lambda Rq_5$	

Então, $Res_{P_1}(q_0\overline{m}_1B...B\overline{m}_n) = \lambda q_5\overline{m}_1B...B\overline{m}_n\rho$. Impressos os marcadores nas extremidades da n-upla, a máquina encontra-se pronta para iniciar a computação principal.

Seja P_2 o programa de Turing que contém todas as quíntuplas de $P^{(5)}$ e, além disso, contém, para cada q_i de $P^{(5)}$, as quíntuplas apresentadas abaixo, nas quais $k = \theta(P^{(5)})$.

$q_i\lambda BLq_{2k+i}$	alcançando λ, apaga λ; move-se um quadrado à esquerda, no qual imprime λ; vai um quadrado à direita para dar continuidade à computação principal.
$q_{2k+i}B\lambda Rq_i$	
$q_i\rho BRq_{4k+i}$	alcançando ρ, apaga ρ; move-se um quadrado à direita, no qual imprime ρ; vai um quadrado à esquerda para dar continuidade à computação principal.
$q_{4k+i}B\rho Lq_i$	

Durante a computação de acordo com $P^{(5)}$, os marcadores podem ser alcançados. Se isto ocorrer, as quíntuplas acima apresentadas disponibilizarão novos espaços vazios à esquerda e à direita a fim de que a computação em desenvolvimento seja concluída, o que ocorrerá se, e somente se, o $Res_P(q_0\overline{m}_1B...B\overline{m}_n)$ estiver definido. Sendo este o caso, o $Res_{P_2}(\lambda q_5\overline{m}_1B...B\overline{m}_n\rho) = \lambda a q_{\theta(P^{(5)})} b\rho$, onde a e b são, respectivamente, uma sequência finita (possivelmente vazia) e uma sequência finita (não vazia), de símbolos da fita e $[\lambda a q_{\theta(P^{(5)})} b\rho] = [Res_P(q_0\overline{m}_1B...B\overline{m}_n)]$. Caso contrário, o $Res_{P_2}(\lambda q_5\overline{m}_1B...B\overline{m}_n\rho)$ também estará indefinido.

Com P_2, finalizamos a computação principal. O próximo passo na construção de P' será apresentar um programa que reúna, em um único bloco, à direita de λ, todos os |'s até então obtidos e, em seguida, substituía λ por |. Antes, porém, a fim de evitar quíntuplas inconsistentes, definiremos o programa P_3 do qual a única quíntupla utilizada ordenará que a máquina assuma um estado interno inédito no programa e mova-se um quadrado à esquerda.

Considerando u = 5k + 1, definimos P_3 como o programa de Turing que contém, para cada s_j de P_2, todas as quíntuplas da forma $q_{\theta(P^{(5)})}s_js_jLq_u$, exceto aquelas cujos dois primeiros símbolos também iniciam alguma quíntupla de P_2. Sendo assim,

$$Res_{P_3}(\lambda a q_{\theta(P^{(5)})} b\rho) = \begin{cases} q_u\lambda ab\rho, & \text{se a é uma sequência vazia} \\ \lambda q_u ab\rho, & \text{se a é uma sequência de comprimento 1} \\ \lambda s_1 \dots s_{n-1} q_u s_n b\rho, & \text{se a é uma sequência de comprimento } n > 1 \end{cases}$$

Seja P_4 o seguinte programa de Turing no qual s é qualquer símbolo da fita presente em P diferente de | e de B.

$q_u\|\|Lq_u$	move-se à esquerda até alcançar λ e vai um quadrado à direita
q_uBBLq_u	
q_ussLq_u	
$q_u\lambda\lambda Rq_{u+1}$	
$q_{u+1}sBRq_{u+1}$	alcançando s, apaga s e vai um quadrado à direita; alcançando |, apaga | e move-se um quadrado à esquerda; alcançando ρ, apaga ρ, preparando-se para finalizar a computação
$q_{u+1}BBRq_{u+1}$	
$q_{u+1}\|BLq_{u+2}$	
$q_{u+1}\rho BLq_{u+4}$	

$q_{u+2}BBLq_{u+2}$ $q_{u+2}\|\|Rq_{u+3}$ $q_{u+2}\lambda\lambda Rq_{u+3}$	move-se à esquerda até alcançar λ ou | e move-se um quadrado à direita
$q_{u+3}B\|Rq_{u+1}$	alcançando B, substitui B por | e move-se um quadrado à direita
$q_{u+4}BBLq_{u+4}$ $q_{u+4}\|\|Lq_{u+4}$ $q_{u+4}\lambda\|Rq_{u+5}$ $q_{u+5}\|\|Lq_{u+6}$	move-se à esquerda: alcançando λ, substitui λ por |, move-se um quadrado à direita e, em seguida, um quadrado à esquerda

De acordo com P_4, independentemente do símbolo da fita que está sendo observado, a máquina vai à esquerda até alcançar λ; quando isto acontece, ela procura | à direita; ao encontrá-lo, ela apaga-o e dirige-se novamente à esquerda para imprimi-lo à direita de λ; mais uma vez, a máquina vai à direita em busca de um outro |, quando o encontra, ela o apaga e dirige-se à esquerda para imprimi-lo à direita do primeiro |; este processo se repete até que todos os |'s presentes no $Res_{P_3}(\lambda a q_{\theta(P^{(5)})} b\rho)$ estejam reunidos em um único bloco imediatamente à direita de λ; quando a máquina encontra ρ, ela apaga-o, dirige-se à esquerda até alcançar λ, o substitui por |, vai para o quadrado da direita e retorna ao quadrado da esquerda, assumindo o estado interno q_{u+6}. Ao final de todas estas ações, a máquina encontra-se pronta para iniciar uma nova computação.

De posse dos quatro programas que acabamos de expor, considere, finalmente, $P' = P_1 \cup P_2 \cup P_3 \cup P_4$. Sendo assim, para qualquer que seja o programa de Turing P, podemos construir um programa P' tal que se houver, de acordo com P, uma computação para a n-upla $(m_1, \ldots, m_n)$, haverá, de acordo com P', uma computação para a mesma n-upla, sendo $Res_{P'}(q_0\overline{m}_1B...B\overline{m}_n) = q_{u+6}\overline{[Res_P(q_0\overline{m}_1B\ldots B\overline{m}_n)]} = q_{\theta(P')}\ \overline{\Psi_P^n(m_1, \ldots, m_n)}$. Portanto, P' é n-regular.

Lema 2.14 Para cada programa de Turing n-regular P e para cada $t > 0$, existe um programa de Turing (t+n)-regular $P^{\#}$ tal que:

(1) se $Res_P(q_0\overline{m}_1B...B\overline{m}_n) = q_{\theta(P)}\overline{r}_1B...B\overline{r}_s$, então $Res_{P^{\#}}(q_0\overline{k}_1B...B\overline{k}_tB\overline{m}_1B...B\overline{m}_n) = q_{\theta(P^{\#})}\overline{k}_1B...B\overline{k}_tB\overline{r}_1B...B\overline{r}_s$.

(2) se $Res_P(q_0\overline{m}_1B...B\overline{m}_n)$ estiver indefinido, então $Res_{P^\#}(q_0\overline{k}_1B...B\overline{k}_tB\overline{m}_1B...B\overline{m}_n)$ também estará indefinido.

Prova:

O programa $P^\#$, obtido a partir de P, oferecerá uma estratégia que, dado o *input* $\overline{k}_1B...B\overline{k}_tB\overline{m}_1B...B\overline{m}_n$, permite a máquina omitir a t-upla $\overline{k}_1B...B\overline{k}_t$, para desenvolver a computação principal (segundo $P^{(t+2)}$) somente sobre a n-upla $\overline{m}_1B...B\overline{m}_n$, reescrevendo ao término de tal computação a t-upla inicialmente omitida. A estratégia consiste em substituir todos os |'s presentes em $\overline{k}_1B...B\overline{k}_t$ por ε, exceto o mais à esquerda que será substituído por δ e, em seguida, imprimir ε no lugar do B que separa $\overline{k}_t$ e $\overline{m}_1$. De modo análogo ao lema 2.13, se houver a necessidade de mais espaços para que a computação principal seja desenvolvida, há quíntuplas em $P^\#$ que disponibilizam novos espaços afastando os blocos de ε's para a esquerda. Ao fim de tal computação, os marcadores δ e ε's são substituídos, fazendo com que $\overline{k}_1B...B\overline{k}_t$ reapareça tal como na configuração instantânea inicial.

Seja T_1 o seguinte programa de Turing no qual δ e ε são símbolos da fita não pertencentes ao programa P:

$q_0\|\delta Rq_1$	substitui o | mais à esquerda por δ
$q_i\|\varepsilon Rq_i$	para cada i, tal que $0 < i < t$
q_iBBRq_{i+1}	substitui os |'s presentes em $\overline{k}_1...\overline{k}_{t-1}$ por ε
$q_t\|\varepsilon Rq_t$	substitui os |'s presentes em $\overline{k}_t$ por ε; move-se um quadrado à direita; em
$q_tB\varepsilon Rq_{t+2}$	seguida, substitui o B (localizado entre $\overline{k}_t$ e $\overline{m}_1$) por ε e vai um quadrado à direita

Então, $Res_{T_1}(q_0\overline{k}_1B...B\overline{k}_tB\overline{m}_1B...B\overline{m}_n) = \delta\varepsilon^{k_1}B\ ...\ B\varepsilon^{k_t+1}\varepsilon q_{t+2}\overline{m}_1B...B\overline{m}_n$, tal que os índices sobrescritos determinam o número de ocorrências de ε. Omitida a t-upla $\overline{k}_1B...B\overline{k}_t$, a máquina está preparada para iniciar a computação principal.

Seja T_2 o programa de Turing que contém todas as quíntuplas de $P^{(t+2)}$ e, além disso, contém, para cada q_i de $P^{(t+2)}$, as quíntuplas apresentadas abaixo, nas quais $N = \theta(P^{(t+2)})$ e s é qualquer símbolo da fita presente em P, exceto | e B.

$q_i\varepsilon\|Lq_{N+i}$ $q_{N+i}\varepsilon\varepsilon Lq_{N+i}$ $q_{N+i}BBLq_{N+i}$ $q_{N+i}\delta BLq_{3N+i}$	interrompe a computação principal; substitui o ε mais à direita por |; move-se à esquerda até alcançar δ; apaga δ e dirige-se um quadrado à esquerda que estará vazio
$q_{3N+i}B\delta Rq_{4N+i}$	imprime δ no quadrado vazio e move-se um quadrado à direita
$q_{4N+i}\varepsilon\varepsilon Rq_{5N+i}$ $q_{4N+i}BBRq_{5N+i}$	move-se à direita para observar o símbolo escrito no quadrado seguinte
$q_{5N+i}\varepsilon\varepsilon Lq_{6N+i}$ $q_{5N+i}BBLq_{7N+i}$ $q_{5N+i}\|BRq_{8N+i}$	observando ε, volta um quadrado no qual copiará ε; observando B, volta um quadrado no qual copiará B; observando |, apaga | e move-se um quadrado à direita
$q_{6N+i}\varepsilon\varepsilon Rq_{4N+i}$ $q_{6N+i}B\varepsilon Rq_{4N+i}$	copia ε
$q_{7N+i}\varepsilon BRq_{4N+i}$ $q_{7N+i}BBRq_{4N+i}$	copia B
$q_{8N+i}\|\|Lq_i$ $q_{8N+i}BBLq_i$ $q_{8N+i}ssLq_i$	dirige-se um quadrado à esquerda (que estará vazio) para retomar a computação principal

Sob os comandos de T_2, a máquina de Turing realiza a computação principal, deslocando todos os ε's um quadrado à esquerda todas as vezes que um ε mais à direita é alcançado. Sendo assim, sempre que $Res_P(q_0\overline{m}_1B...B\overline{m}_n)$ estiver definido, $Res_{T_2}(\delta\varepsilon^{k_1}B \dots B\varepsilon^{k_t+1}\varepsilon q_{t+2}\overline{m}_1B...B\overline{m}_n) = \delta\varepsilon^{k_1}B \dots B\varepsilon^{k_t+1}\varepsilon q_N\overline{r}_1B...B\overline{r}_s$.

Definimos T_3 como o programa de Turing composto pelas seguintes quíntuplas, sendo $L = \theta(T_2)$.

$q_N \| \| L q_N$ $q_N \varepsilon B L q_{L+1}$	dirige-se um quadrado à esquerda e apaga o ε mais à direita
$q_{L+1} \varepsilon \| L q_{L+1}$ $q_{L+1} B B L q_{L+1}$ $q_{L+1} \delta \| R q_{L+2}$	desloca-se para a esquerda, substituindo ε por |; ao encontrar δ, imprime no seu lugar |
$q_{L+2} \| \| L q_{L+3}$ $q_{L+2} B B L q_{L+3}$	dirige-se para o quadrado imediatamente à esquerda

De acordo com T_3 , a t-upla $\overline{k}_1 B...B\overline{k}_t$ reaparece ao término da computação principal por meio da substituição de δ e de todos os ε's (com exceção do último) por |. No quadrado ocupado pelo último ε, primeiro símbolo a ser substituído sob os comandos de T_3 , é escrito B para separar $\overline{k}_t$ e $\overline{r}_1$

Por fim, seja $P^{\#} = T_1 \cup T_2 \cup T_3$. Então, para qualquer que seja o programa de Turing n-regular P e para qualquer $t > 0$, podemos construir um programa $P^{\#}$ tal que se houver, de acordo com P, uma computação para a n-upla $(m_1,...,m_n)$, haverá, de acordo com $P^{\#}$, uma computação para a (t+n)-upla $(\overline{k}_1,...,\overline{k}_t, \overline{m}_1,...,\overline{m}_n)$, sendo $Res_{P^{\#}}(q_0\overline{k}_1 B...B\overline{k}_t B\overline{m}_1 B...B\overline{m}_n) = q_{L+3}\overline{k}_1 B...B\overline{k}_t B\overline{r}_1 B...B\overline{r}_s = q_{\theta(P^{\#})}\overline{k}_1 B...B\overline{k}_t B\overline{r}_1 B...B\overline{r}_s$. Portanto, $P^{\#}$ é (t+n)-regular.

Lema 2.15 Para cada $n > 0$ e $t \geq 0$, existe um programa de Turing (t+n)-regular C_t tal que $Res_{C_t}(q_0\overline{k}_1 B...B\overline{k}_t B\overline{m}_1 B...B\overline{m}_n) = q_{\theta(C_t)}\overline{m}_1 B...B\overline{m}_n B\overline{k}_1 B...B\overline{k}_t B\overline{m}_1 B...B\overline{m}_n$.

Prova:

Podemos concluir a partir do enunciado acima que o programa C_t apenas copia a n-upla $\overline{m}_1 B...B\overline{m}_n$ à esquerda da t-upla $\overline{k}_1 B...B\overline{k}_t$. Em linhas gerais, a máquina comandada por C_t e alimentada pelo *input* $\overline{k}_1 B...B\overline{k}_t B\overline{m}_1 B...B\overline{m}_n$ escreve dois marcadores nas extremidades de $B\overline{k}_1 B...B\overline{k}_t$ obtendo como resultado de tal escrita a seguinte expressão: $\lambda B\overline{k}_1 B...B\overline{k}_t \delta\overline{m}_1 B...B\overline{m}_n$. Na sequência, ela dirige-se à direita, observa o último símbolo da n-upla $\overline{m}_1 B...B\overline{m}_n$ e retorna à esquerda para copiá-lo no quadrado onde está escrito λ. Novamente, ela dirige-se à direita, observa, desta vez, o penúltimo símbolo da n-upla $\overline{m}_1 B...B\overline{m}_n$ e retorna à esquerda para copiá-lo à esquerda do primeiro símbolo anteriormente

copiado; este processo se repete até que todos os símbolos de $\overline{m}_1$B...B$\overline{m}_n$ sejam copiados à esquerda de $\overline{k}_1$B...B$\overline{k}_t$. Neste cenário, C_t consiste das seguintes quíntuplas:

$q_0\|\|Lq_0$ q_0BBLq_1 $q_1B\lambda Rq_2$	escreve o marcador λ dois quadrados à esquerda e move-se um quadrado à direita
$q_i\|\|Rq_i$ q_iBBRq_{i+1} $q_{t+2}\|\|Rq_{t+2}$ $q_{t+2}B\delta Rq_{t+3}$	para cada i, tal que $2 \leq i \leq t+1$ move-se sobre t bloco(s) de |'s, escreve δ entre $\overline{k}_t$ e $\overline{m}_1$ e, em seguida, dirige-se um quadrado à direita
$q_{t+3}\|\|Rq_{t+3}$ $q_{t+3}BBRq_{t+4}$ $q_{t+4}\|\|Rq_{t+3}$ $q_{t+4}BBLq_{t+5}$ $q_{t+5}BBLq_{t+6}$	procura um duplo vazio à direita; ao encontrá-lo, move-se um quadrado à esquerda
$q_{t+6}\|\omega Lq_{t+7}$ $q_{t+6}B\alpha Lq_{t+10}$ $q_{t+6}\delta BLq_{t+14}$	observando |, substitui | por ω, preparando-se para copiar | à esquerda; observando B, substitui B por α, preparando-se para copiar B à esquerda; observando δ, substitui δ por B, preparando-se para terminar a computação
$q_{t+7}\|\|Lq_{t+7}$ $q_{t+7}BBLq_{t+7}$ $q_{t+7}\delta\delta Lq_{t+7}$ $q_{t+7}\lambda\omega Rq_{t+13}$ $q_{t+7}\omega\|Lq_{t+9}$ $q_{t+7}\alpha BLq_{t+9}$	dirige-se à esquerda: alcançando λ substitui λ por ω e vai um quadrado à direita; alcançando ω copia, em seu lugar, |; alcançando α copia, em seu lugar, B; nos dois últimos casos, vai um quadrado à esquerda
$q_{t+9}B\omega Rq_{t+13}$	imprime ω no quadrado mais à esquerda, marcando o lugar no qual será copiado |

$q_{t+10}\|\|Lq_{t+10}$ $q_{t+10}BBLq_{t+10}$ $q_{t+10}\delta\delta Lq_{t+10}$ $q_{t+10}\omega\|Lq_{t+12}$	move-se à esquerda até alcançar ω, copia, em seu lugar, | e vai um quadrado à esquerda
$q_{t+12}B\alpha Rq_{t+13}$	imprime α no quadrado mais à esquerda, marcando o lugar no qual será copiado B
$q_{t+13}\|\|Rq_{t+13}$ $q_{t+13}BBRq_{t+13}$ $q_{t+13}\delta\delta Rq_{t+13}$ $q_{t+13}\omega\|Lq_{t+6}$ $q_{t+13}\alpha BLq_{t+6}$	dirige-se à direita: alcançando ω substitui ω por |; alcançando α substitui α por B; em ambos os casos move-se um quadrado à esquerda
$q_{t+14}\|\|Lq_{t+14}$ $q_{t+14}BBLq_{t+14}$ $q_{t+14}\omega\|Rq_{t+15}$ $q_{t+15}\|\|Lq_{t+16}$	desloca-se para a esquerda, substitui ω por |; vai um quadrado à direita e, em seguida, retorna para o quadrado da esquerda, terminando, assim, a computação

Portanto, seguindo rigorosamente as instruções de C_t, obtemos para qualquer *input* $\overline{k}_1B...B\overline{k}_tB\overline{m}_1B...B\overline{m}_n$ (sendo $t \geq 0$ e $n > 0$) uma computação em cuja descrição instantânea terminal a n-upla $\overline{m}_1B...B\overline{m}_n$ ocorre também à esquerda de $\overline{k}_1B...B\overline{k}_t$, sendo imediatamente precedida por pelo símbolo de estado interno $q_{\theta(C_t)}$.

Lema 2.16 Para cada $n > 0$ e $t > 0$, existe um programa de Turing (t+n)-regular R_t tal que $Res_{R_t}(q_0\overline{k}_1B...B\overline{k}_tB\overline{m}_1B...B\overline{m}_n) = q_{\theta(R_t)}\overline{m}_1B...B\overline{m}_nB\overline{k}_1B...B\overline{k}_t$.

Prova:

O programa R_t, tal qual C_t, faz a máquina copiar a n-upla $\overline{m}_1B...B\overline{m}_n$ à esquerda de $\overline{k}_1B...B\overline{k}_t$, mas diferentemente de C_t, R_t não conserva do lado direito a n-upla original $\overline{m}_1B...B\overline{m}_n$. De fato, a cada símbolo de $\overline{m}_1B...B\overline{m}_n$ copiado à esquerda de $\overline{k}_1B...B\overline{k}_t$, a máquina comandada por R_t retorna à direita e, antes de observar qual será o próximo símbolo que copiará, apaga o símbolo da n-upla original anteriormente copiado.

Diante do exposto, definimos R_t como o programa de Turing que contém todas as quíntuplas de C_t, exceto a quíntupla $q_{t+13}\omega|Lq_{t+6}$ no lugar da qual assumiremos $q_{t+13}\omega BLq_{t+6}$. Esta substituição é fundamental. Com efeito, durante a cópia sob os comandos de C_t, cada | presente na n-upla original é substituído por ω que, por sua vez, será substituído por |. O programa R_t, ao contrário, determina a substituição de ω por B, impedindo, desta forma, que a n-upla original seja reescrita.

Lema 2.17 Para cada programa de Turing n-regular P, existe um programa de Turing n-regular $P^\star$ tal que:

(1) se $Res_P(q_0\overline{m}_1B...B\overline{m}_n) = q_{\theta(P)}\overline{r}_1B...B\overline{r}_s$, então $Res_{P^\star}(q_0\overline{m}_1B...B\overline{m}_n) = q_{\theta(P^\star)}\overline{r}_1B...B\overline{r}_sB\overline{m}_1B...B\overline{m}_n$.

(2) se $Res_P(q_0\overline{m}_1B...B\overline{m}_n)$ estiver indefinido, então $Res_{P^\star}(q_0\overline{m}_1B...B\overline{m}_n)$ também estará indefinido.

Prova:

Ora, como o programa de Turing P é n-regular, então, de acordo com o lema 2.14, há um programa de Turing (n+n)-regular $P^{\#}$, tal que $Res_{P^{\#}}(q_0\overline{m}_1B...B\overline{m}_nB\overline{m}_1B...B\overline{m}_n) = q_{\theta(P^{\#})}\overline{m}_1B...B\overline{m}_nB\overline{r}_1B...B\overline{r}_s$. Sabendo disso, considere $C_0 \cup P^{\#(16)} \cup R_n^{\theta(P^{\#(16)})} = P^\star$. Portanto, como queríamos provar, se $Res_P(q_0\overline{m}_1B...B\overline{m}_n)$ estiver definido, nós temos:

$Res_{C_0}(q_0\overline{m}_1B...B\overline{m}_n) = q_{16}\overline{m}_1B...B\overline{m}_nB\overline{m}_1B...B\overline{m}_n$

$Res_{P^{\#(16)}}(q_{16}\overline{m}_1B...B\overline{m}_nB\overline{m}_1B...B\overline{m}_n) = q_{\theta(P^{\#(16)})}\overline{m}_1B...B\overline{m}_nB\overline{r}_1B...B\overline{r}_s$.

$Res_{R_n^{\theta(P^{\#(16)})}}(q_{\theta(P^{\#(16)})}\overline{m}_1B...B\overline{m}_nB\overline{r}_1B...B\overline{r}_s) = q_{\theta(P^\star)}\overline{r}_1B...B\overline{r}_sB\overline{m}_1B...B\overline{m}_n$.

Lema 2.18 Sejam P_1, ..., P_k programas de Turing (para $k \geq 1$). Então, há um programa de Turing n-regular $P^\dagger$ tal que $Res_{P^\dagger}(q_0\overline{m}_1B...B\overline{m}_n) = q_{\theta(P^\dagger)}\overline{\Psi^n_{P_1}(m_1,\dots,m_n)}B...B\overline{\Psi^n_{P_k}(m_1,\dots,m_n)}$.

Prova (por indução em k):

Para k = 1, $P^\dagger$ é o programa P' obtido de acordo com lema 2.13.

Considerando Y_1 o programa $P^\dagger$ para um k qualquer, mostraremos, a seguir, que também existe um programa $P^\dagger$ para k + 1.

Sejam P_1, ..., P_{k+1} programas de Turing e seja $r_i = \Psi^n_{P_i}(m_1, \dots, m_n)$, para $1 \leq i \leq k + 1$. Por hipótese da indução, há um programa de Turing n-regular Y_1, tal que $Res_{Y_1}(q_0\overline{m}_1B...B\overline{m}_n) = q_{\theta(Y_1)}\overline{\Psi^n_{P_1}(m_1, \dots, m_n)}B...B\overline{\Psi^n_{P_k}(m_1, \dots, m_n)} = q_{\theta(Y_1)}\overline{r}_1B...B\overline{r}_k$. Sendo assim, pelo lema 2.17, há um programa de Turing n-regular Y_2 tal que $Res_{Y_2}(q_0\overline{m}_1B...B\overline{m}_n) = q_{\theta(Y_2)}\overline{r}_1B...B\overline{r}_kB\overline{m}_1B...B\overline{m}_n$. Além disso, dado o programa P_{k+1}, existe, de acordo com o lema 2.13, um programa de Turing n-regular Y_3, tal que $Res_{Y_3}(q_0\overline{m}_1B...B\overline{m}_n) = q_{\theta(Y_3)}\overline{r}_{k+1}$ e, uma vez, dispondo do programa Y_3, existe, de acordo com o lema 2.14, um programa de Turing (k+n)-regular Y_4, tal que $Res_{Y_4}(q_0\overline{r}_1B...B\overline{r}_kB\overline{m}_1B...B\overline{m}_n) = q_{\theta(Y_4)}\overline{r}_1B...B\overline{r}_kB\overline{r}_{k+1}$. Portanto, em linhas gerais, para k + 1, $P^\dagger = Y_2 \cup Y_4^{(\theta(Y_2))}$. Com estes resultados, o lema 2.18 está provado.

Lema 2.19 Se g_1, ..., g_m são funções n-árias parcialmente Turing-computáveis e f é uma função m-ária parcialmente Turing-computável, então a função $h(x_1, \dots, x_n) = f(g_1(x_1, \dots, x_n), ..., g_m(x_1, \dots, x_n))$ também é parcialmente Turing-computável.

Prova:

De acordo com o lema anterior, há um programa de Turing n-regular $P^\dagger$ tal que $Res_{P^\dagger}(q_0\overline{x}_1B...B\overline{x}_n) = q_{\theta(P^\dagger)}\overline{g_1(x_1, \dots, x_n)}B...B\overline{g_m(x_1, \dots, x_n)}$. Se $P_\wr$ é um programa de Turing para computar a função f, então $P = P^\dagger \cup P_\wr^{(\theta(P^\dagger))}$ será o programa de Turing que computará a função h obtida a partir de g e f por composição. Portanto, a função h também é parcialmente Turing-computável.

Neste cenário, se f e cada g_i (para $1 \leq i \leq m$) estiverem definidos, nós teremos:

$Res_{P^\dagger}(q_0\overline{x}_1B...B\overline{x}_n) = q_{\theta(P^\dagger)}\overline{g_1(x_1, \dots, x_n)}B...B\overline{g_m(x_1, \dots, x_n)}$

$Res_{P_\wr^{(\theta(P^\dagger))}}(q_{\theta(P^\dagger)}\overline{g_1(x_1, \dots, x_n)}B...B\overline{g_m(x_1, \dots, x_n)}) = c$, sendo $[c] = f(g_1(x_1, \dots, x_n), ..., g_m(x_1, \dots, x_n)) = h(x_1, \dots, x_n)$

Caso contrário, o $Res_P(q_0\overline{x}_1B...B\overline{x}_n)$ permanecerá indefinido.

Lema 2.20 Se g é uma função n-ária parcialmente Turing-computável, f é uma função n+2-ária parcialmente Turing-computável e h é obtida a partir de g e f por recursão primitiva, então h é parcialmente Turing-computável.

Prova:

Para computar a função $h(x_1, \dots, x_n, y)$, construiremos um programa de Turing Z tal que dado o argumento $\overline{x}_1 B \dots B \overline{x}_n B \overline{y}$, a máquina comandada por Z verifica, inicialmente, se y = 0 ou se y ≠ 0. Ocorrendo o primeiro caso, a máquina apaga o último |, vai ao | mais à esquerda, aplica o programa n-regular para g obtido pelo lema 2.13 (com os índices dos símbolos de estado interno devidamente aumentados) sobre $\overline{x}_1 B \dots B \overline{x}_n$ e finaliza a computação, apagando o primeiro | de $\overline{g(x_1, \dots, x_n)}$. Caso contrário, ou seja, sendo y = z + 1, a máquina escreverá π depois do último |, obtendo, deste modo, $\overline{x}_1 B \dots B \overline{x}_n B \overline{z+1} \pi$; em seguida, copiará esta n+1-upla à direita de π, sem imprimir o último | de $\overline{z+1}$ – $\overline{x}_1 B \dots B \overline{x}_n B \overline{z+1} \pi \overline{x}_1 B \dots B \overline{x}_n B \overline{z}$ – logo depois, a máquina escreverá η no quadrado imediatamente à direita de $\overline{z}$ e copiará $\overline{x}_1 B \dots B \overline{x}_n B \overline{z}$, não imprimindo o último | de $\overline{z}$ e escrevendo η ao final. Este procedimento se repete até que todos os traços de $\overline{z+1}$ sejam eliminados, fato que ocorre quando alcançamos a seguinte expressão:

$$\overline{x}_1 B \dots B \overline{x}_n B \overline{z+1} \pi \overline{x}_1 B \dots B \overline{x}_n B \overline{z} \eta \dots \eta \overline{x}_1 B \dots B \overline{x}_n B | \eta \overline{x}_1 B \dots B \overline{x}_n$$

Em seguida, a máquina aplicará o programa n-regular para g à $\overline{x}_1 B \dots B \overline{x}_n$, acarretando a expressão abaixo, na qual $r_1 = g(x_1, \dots, x_n)$:

$$\overline{x}_1 B \dots B \overline{x}_n B \overline{z+1} \pi \overline{x}_1 B \dots B \overline{x}_n B \overline{z} \eta \dots \eta \overline{x}_1 B \dots B \overline{x}_n B | \eta \overline{r}_1$$

Depois, a máquina apagará o último η e aplicará o programa n+2-regular para f (também obtido pelo lema 2.13) à $\overline{x}_1 B \dots B \overline{x}_n B | B \overline{r}_1$. Este procedimento se repete até π ser alcançado. Quando isso acontece, a máquina aplicará o programa n+2-regular para f à $\overline{x}_1 B \dots B \overline{x}_n B \overline{z}$ e eliminará um |. Na sequência, ela vai à até o primeiro |, apagará todos os |'s à esquerda de π, inclusive π e termina a computação.

Seja Z_1 o seguinte programa de Turing:

$q_0 \| \| R q_0$	move-se à direita até alcançar um duplo vazio
$q_0 B B R q_1$	
$q_1 \| \| R q_0$	
$q_1 B B L q_2$	

q_2BBLq_2 move-se dois quadrados à esquerda para verificar se y = 0 ou se y ≠ 0

$q_2||Lq_3$

Sendo assim, com respeito a Z_1, nós temos duas situações alternativas: (1) Se y = 0, então $Res_{Z_1}(q_0\overline{x}_1B...B\overline{x}_nB\overline{0}) = \overline{x}_1B...B\overline{x}_nq_3B|$; (2) Se y = $z+1$, então $Res_{Z_1}(q_0\overline{x}_1B...B\overline{x}_nB\overline{z+1}) = \overline{x}_1B...B\overline{x}_nB|^zq_3||$

Se (1) ocorre, considere K como um programa n-regular para computar g e Z_2 como o programa que consiste de todas as quíntuplas de $K^{(9)}$ e das quíntuplas abaixo, sendo U = $\theta(K^{(9)})$:

q_3BBRq_4 vai um quadrado à direita e apaga o último traço

$q_4|BLq_5$

q_5BBLq_5 move-se à esquerda até alcançar um duplo vazio e vai dois quadrados à direita para iniciar a computação de $\overline{x}_1B...B\overline{x}_n$ segundo $K^{(9)}$.

$q_5||Lq_6$

$q_6||Lq_6$

q_6BBLq_7

$q_7||Lq_6$

q_7BBRq_8

q_8BBRq_9

$q_U|BRq_{U+1}$ apaga o primeiro | de $\overline{g(x_1,\dots,x_n)}$ e termina

Então, $Res_{Z_2}(\overline{x}_1B...B\overline{x}_nq_3B|) = q_{\theta(Z_2)}|^{g(x_1,\dots,x_n)} = q_{\theta(Z_2)}|^{h(x_1,\dots,x_n,0)}$ e, assim, finaliza-se a computação.

Se (2) ocorre, considere o programa de Turing Z_3 abaixo, sendo F = $\theta(Z_2)$:

$q_3||Rq_{F+1}$ move-se à direita até alcançar um duplo vazio

$q_{F+1}||Rq_{F+1}$

$q_{F+1}BBRq_{F+2}$

$q_{F+2}B\lambda Lq_{F+3}$ imprime λ no segundo quadrado vazio; em seguida, imprime π no primeiro quadrado vazio

$q_{F+3}B\pi Lq_{F+5}$

$q_{F+5}\|\|Lq_{F+5}$ $q_{F+5}BBLq_{F+6}$ $q_{F+5}\eta\eta Rq_{F+8}$ $q_{F+5}\pi\pi Rq_{F+8}$ $q_{F+6}\|\|Lq_{F+5}$ $q_{F+6}BBRq_{F+7}$ $q_{F+7}BBRq_{F+8}$ $q_{F+7}\|\|Rq_{F+8}$	dirige-se à esquerda até alcançar um η, π ou um duplo vazio para iniciar o procedimento de cópia da última n+1-upla à direita
$q_{F+8}\|\omega Rq_{F+9}$ $q_{F+8}B\alpha Rq_{F+12}$ $q_{F+8}\pi\pi Rq_{F+16}$ $q_{F+8}\eta\eta Rq_{F+16}$	observando |: substitui | por ω, preparando-se para copiar | à direita; observando B: substitui B por α, preparando-se para copiar B à direita; observando π ou η: prepara-se para terminar a cópia
$q_{F+9}\|\|Rq_{F+9}$ $q_{F+9}BBRq_{F+9}$ $q_{F+9}\pi\pi Rq_{F+9}$ $q_{F+9}\eta\eta Rq_{F+9}$ $q_{F+9}\lambda\omega Lq_{F+15}$ $q_{F+9}\omega\|Rq_{F+11}$ $q_{F+9}\alpha BRq_{F+11}$	move-se à direita; observando λ: substitui λ por ω e vai um quadrado à esquerda; observando ω: copia, em seu lugar, |; observando α: copia, em seu lugar, B; nos dois últimos casos, vai um quadrado à direita
$q_{F+11}B\omega Lq_{F+15}$	imprime ω no quadrado mais à direita, marcando o lugar no qual será copiado |
$q_{F+12}\|\|Rq_{F+12}$ $q_{F+12}BBRq_{F+12}$ $q_{F+12}\pi\pi Rq_{F+12}$ $q_{F+12}\eta\eta Rq_{F+12}$ $q_{P+12}\omega\|Rq_{F+14}$	move-se à direita; observando ω: copia, em seu lugar, |
$q_{F+14}B\alpha Lq_{F+15}$	imprime α no quadrado mais à direita, marcando o lugar no qual será copiado B

$q_{F+15}\|\|Lq_{F+15}$ $q_{F+15}BBLq_{F+15}$ $q_{P+15}\pi\pi Lq_{F+15}$ $q_{F+15}\eta\eta Lq_{F+15}$ $q_{P+15}\omega\|Rq_{F+8}$ $q_{F+15}\alpha BRq_{F+8}$	desloca-se à esquerda; observando ω: substitui ω por |; observando α: substitui α por B
$q_{F+16}\|\|Rq_{F+16}$ $q_{F+16}BBRq_{F+16}$ $q_{P+16}\omega BLq_{F+18}$	vai à direita; observando ω: substitui ω por B, finalizando, assim, a cópia da n+1-upla anterior sem imprimir o seu último |
$q_{F+18}BBLq_{F+23}$ $q_{F+18}\|\|Rq_{F+19}$ $q_{F+19}B\eta Rq_{F+20}$ $q_{F+20}B\lambda Lq_{F+22}$ $q_{F+22}\eta\eta Lq_{F+5}$	dirige-se um quadrado à esquerda; observando B, finaliza-se o procedimento total de cópia, tendo sido eliminados todos os |'s de $\overline{z+1}$; observando |, vai um quadrado à direita e prepara-se para copiar a última n+1-upla
$q_{F+23}\|\|Lq_{F+23}$ $q_{F+23}BBLq_{F+23}$ $q_{F+23}\eta\eta Rq_{F+24}$	vai à esquerda; alcançando η, vai um quadrado à direita para iniciar a computação da n-upla $\overline{x}_1B...B\overline{x}_n$ segundo o programa n-regular para g

Então, com respeito ao programa Z_3, nós temos que $Res_{Z_3}(\overline{x}_1B...B\overline{x}_nB|^zq_3||) = \overline{x}_1B...B\overline{x}_nB\overline{z+1}\pi\overline{x}_1B...B\overline{x}_nB\overline{z}\eta...\eta\overline{x}_1B...B\overline{x}_nB|\eta q_{F+24}\overline{x}_1B...B\overline{x}_n$.

Agora, sendo $N = \theta(K^{(F+24)})$, assuma Z_4 como o programa de Turing que contém todas as quíntuplas de $K^{(F+24)}$ e, além disso, contém, para cada q_i de $K^{(F+24)}$, as quíntuplas apresentadas abaixo:

$q_i\eta\rho Lq_{N+i}$ $q_i\pi\delta Lq_{N+i}$	interrompe a computação; alcançando η: substitui η por ρ; alcançando π: substitui π por δ.

$q_{N+i}\|\|Lq_{N+i}$	vai à esquerda até alcançar um duplo vazio
$q_{N+i}BBLq_{2N+i}$	
$q_{N+i}\eta\eta Lq_{N+i}$	
$q_{N+i}\pi\pi Lq_{N+i}$	
$q_{2N+i}\|\|Lq_{N+i}$	
$q_{2N+i}BBRq_{3N+i}$	
$q_{3N+i}\|\|Rq_{4N+i}$	move-se uma quadrado à direita: alcançando ρ ou δ, apaga-os.
$q_{3N+i}BBRq_{4N+i}$	
$q_{3N+i}\pi\pi Rq_{4N+i}$	
$q_{3N+i}\eta\eta Rq_{4N+i}$	
$q_{3N+i}\rho BLq_{9N+i}$	
$q_{3N+i}\delta BLq_{9N+i}$	
$q_{4N+i}\|\|Lq_{5N+i}$	observando |, B, π ou η, prepara para copiar no quadrado anterior |, B, π ou η, respectivamente; observando ρ ou δ prepara para copiar no quadrado anterior η ou π, respectivamente
$q_{4N+i}BBLq_{6N+i}$	
$q_{4N+i}\pi\pi Lq_{7N+i}$	
$q_{4N+i}\eta\eta Lq_{8N+i}$	
$q_{4N+i}\rho\rho Lq_{8N+i}$	
$q_{4N+i}\delta\delta Lq_{7N+i}$	
$q_{5N+i}B\|Rq_{3N+i}$	copia |
$q_{5N+i}\|\|Rq_{3N+i}$	
$q_{5N+i}\pi\|Rq_{3N+i}$	
$q_{5N+i}\eta\|Rq_{3N+i}$	
$q_{6N+i}\|BRq_{3N+i}$	copia B
$q_{7N+i}B\pi Rq_{3N+i}$	copia π
$q_{7N+i}\|\pi Rq_{3N+i}$	
$q_{8N+i}\|\eta Rq_{3N+i}$	copia η

$q_{9N+i}\eta\eta Rq_i$ vai um quadrado à direita e retoma a computação

$q_{9N+i}\pi\pi Rq_i$

Sob os comandos de Z_4, a máquina de Turing aplica o programa n-regular $K^{(F+24)}$ para g a $\overline{x}_1B...B\overline{x}_n$. Se durante esta computação, η é alcançado, ele e todos os símbolos da fita que o antecedem são movidos um quadrado à esquerda, criando assim um novo espaço para que a computação segundo $K^{(F+24)}$ seja concluída. Sendo assim, temos que $Res_{Z_4}(\overline{x}_1B...B\overline{x}_nB\overline{z+1}\pi\overline{x}_1B...B\overline{x}_nB\overline{z}\eta...\eta\overline{x}_1B...B\overline{x}_nB|\eta q_{F+24}\overline{x}_1B...B\overline{x}_n) =$ $\overline{x}_1B...B\overline{x}_nB\overline{z+1}\pi$ $\overline{x}_1B...B\overline{x}_nB\overline{z}\eta...\eta\overline{x}_1B...B\overline{x}_nB|\eta q_N\overline{r}_1$.

Considerando L = $\theta(Z_4)$, assuma Z_5 como o seguinte programa de Turing:

$q_N||Lq_N$ vai um quadrado à esquerda e apaga η

$q_N\eta BLq_{L+1}$

$q_{L+1}||Lq_{L+1}$ dirige-se à esquerda; alcançando η ou π, move-se um quadrado à direita

$q_{L+1}BBLq_{L+1}$

$q_{L+1}\eta\eta Rq_{L+2}$

$q_{L+1}\pi\pi Rq_{L+2}$

A máquina de Turing, seguindo as ordens do programa Z_5, apaga o η mais à direita e procura o próximo η (ou π) à esquerda. Ao alcançá-lo, vai um quadrado à direita para iniciar a computação segundo o programa n+2-regular para a função f. Neste contexto, temos que $Res_{Z_5}(\overline{x}_1B...B\overline{x}_nB\overline{z+1}\pi\overline{x}_1B...B\overline{x}_nB\overline{z}\eta...\eta\overline{x}_1B...B\overline{x}_nB|\eta q_N\overline{r}_1) =$ $\overline{x}_1B...B\overline{x}_nB\overline{z+1}\pi\overline{x}_1B...B\overline{x}_nB$ $\overline{z}\eta...\eta q_{L+2}\overline{x}_1B...B\overline{x}_nB|B\overline{r}_1$.

Agora, sendo R um programa n+2-regular para f e S = $\theta(R^{(L+2)})$, considere Z_6 como o programa de Turing que contém $q_S||Lq_S$, $q_S\eta BLq_{L+1}$, todas as quíntuplas de $R^{(L+2)}$, e, além disso, contém, para cada q_i de $R^{(L+2)}$ as quíntuplas adicionais de Z_4 sendo as ocorrências de N substituídas por S.

De acordo com Z_6, a máquina de Turing aplica o programa $R^{(L+2)}$ sobre a n+2-upla anteriormente obtida. Se durante esta computação, π ou η são alcançados, eles e todos os símbolos da fita que os antecedem são movidos um quadrado à esquerda, criando assim um novo espaço para que a computação segundo $R^{(L+2)}$ seja concluída. Quando isso acontece, a máquina vai um quadrado à esquerda; encontrando η, elimina-o e procura o próximo η (ou π) à esquerda. Ao alcançá-lo, a máquina vai para o quadrado imediatamente à direita e aplica o programa $R^{(L+2)}$ à n+2-upla que acabou de ser obtida com a última eliminação de η; encontrando π, prepara-se para concluir a computação. Sendo assim, temos que:

$Res_{Z_6}(\overline{x}_1 B...B \overline{x}_n B \overline{z+1} \pi \overline{x}_1 B...B \overline{x}_n B \overline{z} \eta...\eta q_{L+2} \overline{x}_1 B...B \overline{x}_n B|B\overline{r}_1) = \overline{x}_1 B...B\overline{x}_n B \overline{z+1} q_S \pi \overline{h(x_1, \dots, x_n, z+1)}$.

Seja $M = \theta(Z_6)$. O programa Z_7 é composto pelas seguintes quíntuplas:

$q_S \pi \pi R q_{M+1}$	observando π: vai um quadrado à direita, preparando-se para terminar
$q_{M+1}|BLq_{M+2}$	apaga um | do bloco $\overline{h(x_1, \dots, x_n, z+1)}$
$q_{M+2}\pi\pi Lq_{M+2}$	move-se à esquerda até alcançar um duplo vazio
$q_{M+2}\|\|Lq_{M+2}$	
$q_{M+2}BBLq_{M+3}$	
$q_{M+3}\|\|Lq_{M+2}$	
$q_{M+3}BBRq_{M+4}$	
$q_{M+4}BBRq_{M+4}$	apaga a n+1-upla $\overline{x}_1 B...B\overline{x}_n B\overline{z+1}$; apaga π e termina
$q_{M+4}\|BRq_{M+4}$	
$q_{M+4}\pi BRq_{M+5}$	

Então, $Res_{Z_7}(\overline{x}_1 B...B\overline{x}_n B\overline{z+1} q_S \pi \overline{h(x_1, \dots, x_n, z+1)}) = q_{\theta(Z_7)}B|^{h(x_1,\dots,x_n,z+1)}$.

Finalmente, seja $Z = Z_1 \cup Z_2 \cup Z_3 \cup Z_4 \cup Z_5 \cup Z_6 \cup Z_7$. Então, se houver uma computação para a n+1-upla $(x_1, \dots, x_n, y)$ de acordo com Z, teremos:

(1) Sendo y = 0, $Res_Z(q_0\overline{x}_1 B...B\overline{x}_n B\overline{y}) = q_{\theta(Z_2)}|^{h(x_1,\dots,x_n,y)}$

(2) Sendo $y \neq 0$, $Res_Z(q_0\overline{x}_1 B...B\overline{x}_n B\overline{y}) = q_{\theta(Z_7)}B|^{h(x_1,\dots,x_n,y)}$

Evidentemente, se $h(x_1, \dots, x_n, y)$ estiver indefinida para y, estará indefinida também para todo $w \geq y$. Neste termos, Z constitui um programa de Turing para computar $h(x_1, \dots, x_n, y)$. Portanto, como existe um tal programa, a função h obtida por recursão primitiva a partir da funções parcialmente Turing-computáveis g e f é também parcialmente Turing-computável.

Lema 2.21 Se g é uma função n+1-ária parcialmente Turing-computável, então a função $h(x_1, \dots, x_n) = \mu_y(g(x_1, \dots, x_n, y) = 0)$ é parcialmente Turing-computável.

Prova:

Construiremos para a função h um programa de Turing de acordo com o qual a máquina computará para cada $y \geq 0$ a função $g(x_1, \dots, x_n, y)$ até alcançar o primeiro y tal que $g(x_1, \dots, x_n, y) = 0$. Se um tal y não existir, a máquina entrará em *loop*.

Seja H_1 o seguinte programa de Turing:

$q_0||Rq_0$ imprime um | à direita do último B que encerra a n-upla

q_0BBRq_1

$q_1||Rq_0$

$q_1B|Lq_2$

$q_2||Lq_2$ move-se à esquerda até que um duplo vazio seja alcançado; em seguida, move-se dois quadrados à direita

q_2BBLq_3

$q_3||Lq_2$

q_3BBRq_4

q_4BBRq_5

Então, $Res_{H_1}(q_0\overline{x}_1B...B\overline{x}_n) = q_5\overline{x}_1B...B\overline{x}_nB\overline{0}$.

Seja Y o programa para computar a função $g(x_1, \dots, x_n, y)$ e, Y' o programa n+1-regular obtido a partir de Y pelo lema 2.13. Então, $Res_{Y'}(q_0\overline{x}_1B...B\overline{x}_nB\overline{y}) = q_{\theta(Y')}\overline{g(x_1, \dots, x_n, y)}$. Considere, agora, Y' como sendo o programa de Turing P do lema 2.17. Então, por este mesmo lema, há um programa n+1-regular $Y^\star$ tal que $Res_{Y^\star}(q_0\overline{x}_1B...B\overline{x}_nB\overline{y}) = q_{\theta(Y^\star)}\overline{g(x_1, \dots, x_n, y)}B\overline{x}_1B...B\overline{x}_nB\overline{y}$. Neste contexto, sendo $N = \theta(Y^{\star(5)})$, temos que $Res_{Y^{\star(5)}}(q_5\overline{x}_1B...B\overline{x}_nB\overline{y}) = q_N\overline{g(x_1, \dots, x_n, y)}B\overline{x}_1B...B\overline{x}_nB\overline{y}$.

Seja H_2 o programa de Turing contendo as seguintes quíntuplas:

$q_N\|Bq_{N+1}$ $q_{N+1}\|\|Lq_{N+2}$ $q_{N+2}BBRq_{N+3}$ $q_{N+1}BBRq_{N+9}$	apaga o primeiro | de $\overline{g(x_1, \dots, x_n, y)}$ e vai um quadrado à direita: (1) observando | (caso em que $g(x_1, \dots, x_n, y) > 0$), move-se um quadrado à esquerda e retorna um quadrado à direita; (2) observando B (caso em que $g(x_1, \dots, x_n, y) = 0$), substitui B pelo próprio B, vai à direita e prepara para terminar

Então, com respeito a H_2, obtemos duas configurações instantâneas terminais alternativas: (1) se $g(x_1, \dots, x_n, y) = k > 0$, $Res_{H_2}(q_N\overline{g(x_1, \dots, x_n, y)}B\overline{x}_1B...B\overline{x}_nB\overline{y}) = q_{N+3}|^kB\overline{x}_1B...B\overline{x}_nB\overline{y}$; (2) se $g(x_1, \dots, x_n, y) = 0$, $Res_{H_2}(q_N\overline{g(x_1, \dots, x_n, y)}B\overline{x}_1B...B\overline{x}_nB\overline{y}) = q_{N+9}\overline{x}_1B...B\overline{x}_nB\overline{y}$.

Assuma H_3 como o seguinte programa:

$q_{N+3}\|Bq_{N+3}$ $q_{N+3}BBRq_{N+4}$	apaga $g(x_1, \dots, x_n, y)$ |'s e move-se um quadrado à direita
$q_{N+4}\|\|Rq_{N+4}$ $q_{N+4}BBRq_{N+5}$ $q_{N+5}\|\|Rq_{N+4}$ $q_{N+5}BBLq_{N+6}$ $q_{N+6}B\|Lq_{N+7}$	desloca-se à direita até alcançar um duplo vazio, acrescenta um | a $\overline{y}$ e vai um quadrado à esquerda
$q_{N+7}\|\|Lq_{N+7}$ $q_{N+7}BBLq_{N+8}$ $q_{N+8}\|\|Lq_{N+7}$ $q_{N+8}BBRq_4$	move-se à esquerda até alcançar um duplo vazio e, em seguida, vai um quadrado à direita, preparando–se, deste modo, para iniciar a computação de $\overline{x}_1B...B\overline{x}_n\overline{y+1}$ segundo $Y^{\star(5)}$

Então, nestes termos, $Res_{H_3}(q_{N+3}|^kB\overline{x}_1B...B\overline{x}_nB\overline{y}) = q_4B\overline{x}_1B...B\overline{x}_nB\overline{y+1}$

Seja W o programa de Turing que computa a função $U_{n+1}^{n+1}(x_1, \dots, x_n, y)$. Neste caso, como sabemos, $Res_W(q_0\overline{x}_1B...B\overline{x}_nB\overline{y}) = c$, tal que $[c] = y$. Considere, agora, o programa $W^{(N+9)}$. De modo análogo, $Res_{W^{(N+9)}}(q_{N+9}\overline{x}_1B...B\overline{x}_nB\overline{y}) = c$, tal que $[c] = y$. Sendo assim, sob os comandos de $W^{(N+9)}$, alcançamos, finalmente, o resultado que procurávamos, a saber, $\mu_y(g(x_1, \dots, x_n, y) = 0)$, ou seja, o valor da função $h(x_1, \dots, x_n)$.

Portanto, em síntese, $H = H_1 \cup Y^{\star(5)} \cup H_2 \cup H_3 \cup W^{(N+9)}$ constitui o programa de Turing para computar $h(x_1, \ldots, x_n)$. Como existe um tal programa, a função $h(x_1, \ldots, x_n)$ obtida por minimização ilimitada a partir da função parcialmente Turing-computável $g(x_1, \ldots, x_n, y)$ é também parcialmente Turing-computável.

Teorema 2.22 Toda função parcial recursiva é parcialmente Turing-computável.

Prova:

Como vimos, as funções iniciais são parcialmente Turing-computáveis e, de acordo com os lemas 2.19, 2.20 e 2.21, o conjunto das funções parcialmente Turing-computáveis é fechado em relação às operações básicas. Logo, o conjunto das funções parciais recursivas está contido no conjunto das funções parcialmente Turing-computáveis.

Corolário 2.23 Toda função recursiva é Turing-computável.

Prova:

Obtida por particularização do teorema anterior.

2.2.2 Toda função parcialmente Turing-computável é parcial recursiva

Conforme dissemos anteriormente, em outras palavras, o nosso principal objetivo, neste capítulo, é provar o seguinte teorema: uma função numérica *f* é parcial recursiva se, e somente se, *f* é parcialmente Turing-computável. Parte deste objetivo acabou de ser alcançada com a prova do teorema 2.22. Falta-nos ainda provar que se *f* é parcialmente Turing-computável, *f* é parcial recursiva. Para isso, aritmetizaremos, *a la* Gödel, a teoria das máquinas e programas de Turing.

A aritmetização godeliana foi, inicialmente, utilizada para traduzir os enunciados metamatemáticos da Aritmética Elementar de Primeira Ordem de Peano para a linguagem da própria aritmética composta por números naturais. De modo semelhante, podemos aritmetizar qualquer linguagem formal, atribuindo números aos seus componentes básicos de modo que:

(1) Objetos distintos tenham números distintos.
(2) Dado um objeto qualquer, possamos efetivamente encontrar o seu número.
(3) Dado um número qualquer, possamos efetivamente decidir se ele está atribuído a algum objeto e, se estiver, a qual objeto.

Para aritmetizar a teoria das máquinas e programas de Turing, começaremos codificando os símbolos, expressões e sequências finitas de expressões de L.

Definição 2.24 Seja A = {k : k é um símbolo de L}.O *código de k* é o número natural g(k) determinado pela função injetiva g: A → ℕ tal que:

(1) se k = R, $g(k) = 3$; se k = L, $g(k) = 5$;
(2) se $k = s_i$, $g(k) = 7 + 4i$, para $i \geq 0$;
(3) se $k = q_i$, $g(k) = 9 + 4i$, para $i \geq 0$.

Os códigos obtidos de acordo com a definição acima são facilmente gerados obedecendo a sequência estabelecida na tabela abaixo.

R	L	s_0	q_0	s_1	q_1	s_2	q_2	s_3	q_3	⋯
3	5	7	9	11	13	15	17	19	21	⋯

Definição 2.25 Seja B = {r : r é uma expressão de L}. O *código de r*, para $r = k_1, \ldots, k_n$, é o número natural g'(r) determinado pela função injetiva g': B → ℕ tal que $g'(r) = \prod_{i<n} p_i^{g(k_{i+1})}$. Por convenção, se r é uma expressão vazia, g'(r) = 1.

Nestes termos, se $r = q_0 R s_2 q_1 L$, $g'(r) = 2^{g(q_0)} \cdot 3^{g(R)} \cdot 5^{g(s_2)} \cdot 7^{g(q_1)} \cdot 11^{g(L)} = 2^9 \cdot 3^3 \cdot 5^{15} \cdot 7^{13} \cdot 11^5$.

Definição 2.26 Seja C = {t : t é uma sequência finita de expressões de L}. O *código de t*, para $t = r_1, \ldots, r_n$, é o número natural g''(t) determinado pela função injetiva g'': C → ℕ tal que $g''(t) = \prod_{i<n} p_i^{g'(r_{i+1})}$.

Dada a seguinte sequência de expressões:

$$r_1 = q_2 R s_2$$
$$r_2 = s_{17} q_5 s_1 R$$
$$r_3 = RL$$

Se $t = r_1, r_2, r_3$, então $g''(t) = 2^{g'(r_1)} \cdot 3^{g'(r_2)} \cdot 5^{g'(r_3)} = 2^{2^{17} \cdot 3^3 \cdot 5^{15}} \cdot 3^{2^{75} \cdot 3^{29} \cdot 5^{11} \cdot 7^3} \cdot 5^{2^3 \cdot 3^5}$.

Considerando as codificações acima sugeridas, algumas observações serão convenientes:

- Como g, g’ e g’’ são funções injetivas totais, além de cada símbolo, cada expressão e cada sequência de expressões de L ter um único código, símbolos diferentes, expressões diferentes e sequências diferentes de expressões de L terão códigos diferentes.

- Dado um número natural $n > 0$, podemos, sem grandes dificuldades, identificar se n é código de um símbolo, expressão ou de uma sequência de expressões de L da seguinte maneira:

  (1) se n é ímpar e maior que 1, então n é código de um único símbolo de L;
  (2) se n = 1 ou se n é par e a sua decomposição consiste no segmento inicial do conjunto dos números primos com expoentes ímpares maiores que 1, então n é código de uma única expressão de L;
  (3) se n é par e a sua decomposição consiste no segmento inicial do conjunto dos números primos com expoentes que satisfazem (2), então n é o código de uma única sequência de expressões de L.

- Se n satisfaz alguma condição acima exposta, podemos ainda identificar exatamente o símbolo, a expressão ou a sequência de expressões que n codifica; para isso, basta recuperar, considerando a definição 2.24, os símbolos ou a expressão vazia codificados pelos expoentes ímpares.

Definição 2.27 Seja P um programa de Turing, o número natural n é um *código de P* se, e somente se, n é o código de uma sequência das quíntuplas de P.

Como consequência da definição anterior, se P contém x quíntuplas, P possuirá x! códigos, correspondentes às x! permutações de suas quíntuplas.

Para concluirmos a aritmetização a qual nos propomos, apresentaremos uma lista de relações que transpõem para a linguagem aritmética as definições referentes à Teoria das máquinas e dos programas de Turing apresentadas na primeira seção deste capítulo. Estas relações são as seguintes:

(1) $G_n = \{x : \sim \bigvee_{y<lh(x)-1}[(x)_y = 0 \wedge (x)_{y+1} \neq 0]\}$

$G_n(x)$ se, e somente se, a decomposição de x ocorre no segmento inicial dos números primos.

(2) $\text{Term} = \{(x, z) : G_n(z) \wedge \bigvee_{i<lh(z)}[x = (z)_i]\}$

Term(x, z) se, e somente se, x é um dos expoentes da decomposição de z.

(3) SI = $\{x : \bigvee_{y<x}[x = 4y + 9]\}$

SI(x) se, e somente se, x é código de um símbolo de estado interno.

(4) AI $= \{x: \bigvee_{y<x}[x = 4y + 7]\}$

AI(x) se, e somente se, x é código de um símbolo da fita.

(5) Quint = $\{x : G_n(x) \wedge lh(x) = 5 \wedge SI((x)_0) \wedge AI((x)_1) \wedge AI((x)_2) \wedge ((x)_3 = 3 \vee (x)_3 = 5) \wedge SI((x)_4)\}$

Quint(x) se, e somente se, x é código de uma quíntupla.

(6) Inc = $\{(x, y) : Quint(x) \wedge Quint(y) \wedge (x)_0 = (y)_0 \wedge (x)_1 = (y)_1 \wedge x \neq y\}$

Inc(x, y) se, e somente se, x e y são códigos de quíntuplas inconsistentes.

(7) PT = $\{z : G_n(z) \wedge \bigwedge_{i<lh(z)} [Quint((z)_i) \wedge \sim\bigvee_{j<lh(z)} [Inc((z)_j, (z)_i)]]\}$

PT(z) se, e somente se, z é código de um programa de Turing.

(8) A função NR(x) determina o código da expressão $\overline{x}$ da seguinte maneira:

NR(0) = 2^{11}
NR(x + 1) = $2^{11} * NR(x)$

(9) A função $Init_n(x_1, ..., x_n)$ determina o código de uma configuração instantânea inicial $q_0\overline{x}_1B...B\overline{x}_n$ da seguinte maneira:

$Init_n(x_1, ..., x_n) = 2^9 * NR(x_1) * 2^7 * ... * 2^7 * NR(x_n)$

(10) CI = $\{x : G_n(x) \wedge \bigvee_{i<lh(x)-1}[SI((x)_i) \wedge \bigwedge_{j<lh(x)}[j \neq i \rightarrow AI((x)_j)]]\}$

CI(x) se, e somente se, x é código de uma configuração instantânea.

(11) $Acarr_1$ = $\{(x, y, z) : CI(x) \wedge CI(y) \wedge PT(z) \wedge \bigvee_{b_1<x} \bigvee_{b_2<x} \bigvee_{i<x} \bigvee_{k<x} \bigvee_{p<x} \bigvee_{t<y} \bigvee_{l<y} [x = b_1 * 2^i * 2^k * 2^p * b_2 \wedge y = b_1 * 2^t * 2^l * 2^p * b_2 \wedge SI(i) \wedge SI(l) \wedge AI(k) \wedge AI(t) \wedge AI(p) \wedge Term(2^i . 3^k . 5^t . 7^3 . 11^l, z)]\}$

$Acarr_1$ (x, y, z) se, e somente se, sendo t_1 e t_2 expressões possivelmente vazias, x é o código de $t_1q_is_ks_pt_2$, y é o código de $t_1s_tq_ls_pt_2$, e o programa de Turing, que z tem como um de seus códigos, contém a quíntupla (q_i, s_k, s_t, R, q_l)

(12) $\text{Acarr}_2 = \{(x, y, z) : \text{CI}(x) \wedge \text{CI}(y) \wedge \text{PT}(z) \wedge \bigvee_{b_1<x} \bigvee_{b_2<x} \bigvee_{i<x} \bigvee_{k<x} \bigvee_{t<y} \bigvee_{l<y} [x = b_1 * 2^i * 2^k \wedge y = b_1 * 2^t * 2^l * 2^7 \wedge \text{SI}(i) \wedge \text{SI}(l) \wedge \text{AI}(k) \wedge \text{AI}(t) \wedge \text{Term}(2^i.3^k.5^t.7^3.11^l, z)]\}$

Acarr_2 (x, y, z) se, e somente se, sendo t_1 uma expressão possivelmente vazia, x é o código de $t_1 q_i s_k$, y é o código de $t_1 s_t q_l s_0$ e o programa de Turing, que z tem como um de seus códigos, contém a quíntupla (q_i, s_k, s_t, R, q_l)

(13) $\text{Acarr}_3 = \{(x, y, z) : \text{CI}(x) \wedge \text{CI}(y) \wedge \text{PT}(z) \wedge \bigvee_{b_1<x} \bigvee_{b_2<x} \bigvee_{i<x} \bigvee_{k<x} \bigvee_{p<x} \bigvee_{t<y} \bigvee_{l<y} [x = b_1 * 2^p * 2^i * 2^k * b_2 \wedge y = b_1 * 2^l * 2^p * 2^t * b_2 \wedge \text{SI}(i) \wedge \text{SI}(l) \wedge \text{AI}(k) \wedge \text{AI}(t) \wedge \text{AI}(p) \wedge \text{Term}(2^i.3^k.5^t.7^5.11^l, z)]\}$

(14) $\text{Acarr}_4 = \{(x, y, z) : \text{CI}(x) \wedge \text{CI}(y) \wedge \text{PT}(z) \wedge \bigvee_{b_1<x} \bigvee_{b_2<x} \bigvee_{i<x} \bigvee_{k<x} \bigvee_{t<y} \bigvee_{l<y} [x = 2^i * 2^k * b_2 \wedge y = 2^l * 2^7 * 2^t * b_2 \wedge \text{SI}(i) \wedge \text{SI}(l) \wedge \text{AI}(k) \wedge \text{AI}(t) \wedge \text{Term}(2^i.3^k.5^t.7^5.11^l, z)]\}$

As relações Acarr_i (x, y, z), $1 \leq i \leq 4$, correspondem à contraparte aritmética da definição 2.6.

(15) $\text{Acarr}_5 = \{(x, y, z) : \text{Acarr}_1 (x, y, z) \vee \text{Acarr}_2 (x, y, z) \vee \text{Acarr}_3 (x, y, z) \vee \text{Acarr}_4 (x, y, z)\}$

Acarr_5 (x, y, z) se, e somente se, x acarreta y via z.

(16) $\text{Fin} = \{(x, z) : \text{CI}(x) \wedge \text{PT}(z) \wedge \bigvee_{t_1<x} \bigvee_{t_2<x} \bigvee_{i<x} \bigvee_{k<x} [x = t_1 * 2^i * 2^k * t_2 \wedge \text{SI}(i) \wedge \text{AI}(k) \wedge \bigwedge_{n<\text{lh}(z)} [((z)_n)_0 \neq i \vee ((z)_n)_1) \neq k]]\}$

Fin(x, z) se, e somente se, x é o código de uma configuração instantânea terminal com respeito a um programa de Turing que tem z como um de seus códigos.

(17) $\text{Comp} = \{(y, z) : \text{PT}(z) \wedge \bigvee_{t<y} [G_n(t) \wedge \bigwedge_{n<\text{lh}(t)-1} [\text{Acarr}_5((t)_n, (t)_{n+1}, z) \wedge \text{Fin}((t)_{\text{lh}(t)-1}, z)] \wedge y = 2^z . \prod_{i<\text{lh}(t)} p_{i+1}^{(t)_i}]\}$

Comp(y, z) se, e somente se, y é o código de uma computação com respeito ao programa de Turing de código z. Deste modo, y não é simplesmente o código de uma mera computação, mas de uma computação associado a um dos códigos do programa que a permite. Observe que como um programa de Turing com n quíntuplas possui n! códigos, então uma computação de acordo com este programa terá n! códigos, o que nos autorizará mencionarmos, mais adiante, o menor código de uma computação com respeito a um programa de Turing.

(18) $T_n = \{(z, x_1, \ldots, x_n, y) : \text{Comp}(y, z) \wedge (y)_1 = \text{Init}_n(x_1, ..., x_n)\}$

$T_n(z, x_1, \ldots, x_n, y)$ se, e somente se, y é o código de uma computação com *input* $(x_1, \ldots, x_n)$ realizada de acordo com um programa de Turing de código z.

Todas as relações acima definidas são claramente recursivas primitivas. Abaixo, apresentaremos três funções também recursivas primitivas, dentre as quais a última delas determina o valor de uma computação de acordo com um programa de Turing.

(a) A função C(n, x) estabelece se $(x)_n$ é ou não o código de |.

$$C(n, x) = \begin{cases} 1, & \text{se } (x)_n = 11 \\ 0, & \text{se } (x)_n \neq 11 \end{cases}$$

(b) Se x é código de uma expressão, então a função Out(x) determina o números de |'s de x.

$$\text{Out}(x) = \sum_{n<\text{lh}(x)} C(n, x)$$

(c) Se x é o código de uma computação de acordo com z, isto é, $x = 2^z \cdot 3^{m_1} \cdot \ldots \cdot p_n^{m_n}$ sendo m_1, ..., m_n códigos das configurações instantâneas c_1, ..., c_n, respectivamente, então U(x) estabelece o número de |'s presentes na configuração instantânea terminal.

$$U(x) = \text{Out}((x)_{\text{lh}(x)-1})$$

Aritmetizada a teoria das máquinas e programas de Turing, dispomos agora dos instrumentos necessários para obtermos como corolário do Teorema da Forma Normal de Kleene, que será provado mais adiante, o resultado que buscávamos: toda função parcialmente Turing-computável é parcial recursiva. A prova deste teorema requisitará o lema seguinte.

Lema 2.28 Se P é um programa de Turing e z, um de seus possíveis códigos, temos que:

(1) $\text{dom}(\Psi_P^n(x_1, ..., x_n)) = \text{dom}(\mu_y(T_n(z, x_1, ..., x_n, y)))$

(2) $\Psi_P^n(x_1, ..., x_n) = U(\mu_y(T_n(z, x_1, ..., x_n, y)))$

Prova:

(1) Pela definição 2.10, $\Psi_P^n(x_1, ..., x_n)\downarrow$ se e somente se existe uma computação de acordo com P para a n-upla $(x_1, ..., x_n)$; em termos aritméticos, isso ocorre se, e somente se, existe

um número y, tal que $T_n(z, x_1, ..., x_n, y)$. Sendo assim, neste caso, $\mu_y(T_n(z, x_1, ..., x_n, y))$ também está definido.

(2) Como vimos, ainda pela definição 2.10, se $\Psi_P^n(x_1, ..., x_n) \downarrow$, então $\Psi_P^n(x_1, ..., x_n) = [Res_P(q_0\overline{x}_1B...B\overline{x}_n)]$. Por outro lado, se existe um y tal que $T_n(z, x_1, ..., x_n, y)$ e $w = \mu_y(T_n(z, x_1, ..., x_n, y))$, então $(w)_{lh(w)-1}$ é o código da configuração instantânea terminal c_n de y e $U(w) = [c_n] = [Res_P(q_0\overline{x}_1B...B\overline{x}_n)]$. Portanto, existindo uma computação de acordo com P para a n-upla $(x_1, ..., x_n)$, $\Psi_P^n(x_1, ..., x_n) = U(\mu_y(T_n(z, x_1, ..., x_n, y)))$.

Teorema 2.29 (*Teorema da Forma Normal de Kleene*) Uma função numérica $h(x_1, ..., x_n)$ é parcialmente Turing-computável se, e somente se, existe um z tal que $h(x_1, ..., x_n) = U(\mu_y(T_n(z, x_1, ..., x_n, y)))$.

Prova:

1º lado: A função $h(x_1, \ldots, x_n)$ é parcialmente Turing-computável. Então, segundo a definição 2.10, existe um programa de Turing P tal que $h(x_1, \ldots, x_n) = \Psi_P^n(x_1, \ldots, x_n)$. Pelo lema 2.28, $\Psi_P^n(x_1, ..., x_n) = U(\mu_y(T_n(z, x_1, ..., x_n, y)))$. Portanto, $h(x_1, \ldots, x_n) = U(\mu_y(T_n(z, x_1, ..., x_n, y)))$, para algum código z de P.

2º lado: Existe um z tal que $h(x_1, ..., x_n) = U(\mu_y(T_n(z, x_1, ..., x_n, y)))$. Pelo lema 2.28, $\Psi_P^n(x_1, ..., x_n) = U(\mu_y(T_n(z, x_1, ..., x_n, y)))$. Portanto, $h(x_1, \ldots, x_n) = \Psi_P^n(x_1, \ldots, x_n)$, para algum programa de Turing P. Em outras palavras, segundo a definição 2.10, $h(x_1, ..., x_n)$ é parcialmente Turing-computável.

Corolário 2.30 Se $h(x_1, ..., x_n)$ é parcialmente Turing-computável, então $h(x_1, ..., x_n)$ é parcial recursiva.

Prova:

De acordo com o teorema anterior, se $h(x_1, ..., x_n)$ é parcialmente Turing-computável, então $h(x_1, \ldots, x_n) = U(\mu_y(T_n(z, x_1, ..., x_n, y)))$, para algum z. Como $U(\mu_y(T_n(z, x_1, ..., x_n, y)))$ é, claramente, parcial recursiva, então, por consequência, $h(x_1, ..., x_n)$ também o é.

Corolário 2.31 Se $h(x_1, ..., x_n)$ é Turing-computável, então $h(x_1, ..., x_n)$ é recursiva.

Prova:

Obtida por particularização do corolário anterior.

2.3 Tese de Church-Turing

No ano de 1936, Alan Turing publica o artigo *On the Computable numbers with an application to the Entscheidunsproblem*, no qual apresenta as suas máquinas abstratas como

representação formal para a noção intuitiva de procedimento efetivo, idealizadas segundo ele, de acordo com o ato de computar do "computador humano". Para Turing, os inúmeros procedimentos computacionais realizados pelo homem, orientados por instruções previamente estabelecidas, podiam também ser executados por suas máquinas. Esta constatação o levou a enunciar a sua famosa tese - a Tese de Turing - apresentada, normalmente, sob duas versões.

Tese de Turing (*versão estrita*)

Toda função algorítmica é Turing-computável.

Tese de Turing (*versão estendida*)

Toda função parcial algorítmica é parcialmente Turing-computável.

Identificação equivalente a esta foi obtida, de maneira independente, por Alonzo Church, utilizando as funções lambda-definíveis, versão formal que ele propôs para o conceito de função algorítmica. Na literatura especializada, usa-se comumente o termo "Tese de Church-Turing", fazendo jus a ambos os autores que a enunciaram.

Sobre esta tese, Kalmár afirma:

> [Ela] não é um teorema matemático que possa ser provado ou refutado num sentido matemático exato, pois estabelece a identidade de duas noções, em que somente uma é definida matematicamente, enquanto a outra é usada pelos matemáticos sem uma definição exata. (KALMÁR, 1957, p. 72-73)

Embora não haja, para esta tese, uma demonstração, ela é largamente admitida, pois existem fortes argumentos "empíricos" que lhe dão plausibilidade, dentre os quais destacamos:

(1) Apesar das inúmeras tentativas, nunca se conseguiu apresentar uma função algorítmica ou parcial algorítmica que não fosse, respectivamente, Turing-computável ou parcialmente Turing-computável.

(2) Até hoje, todas as tentativas de caracterizar formalmente as noções vagas e imprecisas de função algorítmica e função parcial algorítmica forneceram exatamente as mesmas classes de funções, a saber, a classe das funções Turing-computáveis e a classe das funções parcialmente Turing-computáveis.

Os argumentos contrários à Tese de Church-Turing, por sua vez, nunca foram fortes o suficiente para refutá-la. Ela é aceita pela maioria dos matemáticos e cientistas da computação e os poucos que demonstram algum grau de descrença quanto ao seu enunciado são tachados

como excêntricos. No capítulo seguinte, vamos utilizá-la como método informal de prova para a obtenção de alguns resultados. Em Teoria da Computabilidade, esta prática é recorrente, como pode ser atestado a seguir nas palavras do matemático Cohen (apud CARNIELLI, EPSTEIN, 2009, p.300):

> Ou seja, desde que tenhamos dado um argumento intuitivo de que a função é [parcialmente] computável [...] afirmamos então que a Tese de Church nos diz que a função é recursiva parcial. Isto simplifica cálculos tediosos; os leitores devem se convencer, no entanto, que todas as vezes que a Tese de Church é usada, uma prova formal pode ser elaborada por alguém que seja suficientemente industrioso.

Por fim, atentos ao enunciado da Tese de Church-Turing, podemos, em última análise, considerá-la como uma tentativa de se determinar precisamente o alcance e os limites da computação teórica. Com ela, acredita-se que se tenha definitivamente, captado, de forma precisa, as noções intuitivas de algoritmo e de função algorítmica.

3 PROBLEMA DA DECISÃO PARA OS SUBCONJUNTOS DAS FUNÇÕES PARCIAIS RECURSIVAS

Nos capítulos anteriores, vimos que as funções (parciais) recursivas ou (parcialmente) Turing-computáveis são (parciais) algorítmicas. Mas será que os conjuntos dessas funções também são algorítmicos? E os seus subconjuntos próprios: o que dizer sobre eles? Para responder estas perguntas, mostraremos, neste capítulo, como enumerar efetivamente os programas de Turing e as funções parciais recursivas. Com o auxílio desta enumeração, vamos provar o Teorema s-m-n de Kleene, utilizado na demonstração de diversos teoremas da Teoria da Computabilidade. Contextualizaremos o problema da decisão para uma relação numérica qualquer, introduzindo, deste modo, conceitos importantes como solubilidade e insolubilidade recursiva a fim de apresentarmos, em termos formais, alguns problemas insolúveis, dentre eles, o famoso Problema da Parada. Por fim, apresentaremos o Teorema de Rice, com o qual alcançaremos o nosso último objetivo, a saber: provar, por um lado, a decidibilidade do conjunto das funções parciais recursivas e, por outro, a indecidibilidade dos conjuntos das funções recursivas e das funções recursivas primitivas.

3.1 Lista efetiva de programas e funções

Considerando a codificação dos programas de Turing apresentada no capítulo anterior, sabemos que cada programa possui ao menos um código e cada um deles, por sua vez, codifica um único programa. Isto posto, estabeleceremos, a seguir, um modo, dentre muitos, de se construir uma lista $P_0,\ P_1,\ P_2,$ ... de todos os programas de Turing na qual P_x denotará o *x*-ésimo programa listado.

Definição 3.1 Seja Z_x o programa de Turing que tem *x* como um de seus códigos. O *x*-ésimo programa da lista $P_0,\ P_1,\ P_2,$ ... é determinado de acordo com a seguinte função:

$$P_x = \begin{cases} Z_x, & \text{se } \mathrm{PT}(x) \\ \{\mathrm{q}_0||\mathrm{Lq}_0, \mathrm{q}_0\mathrm{BBLq}_0\}, & \text{se } \sim\mathrm{PT}(x) \end{cases}$$

Em outras palavras, a construção da lista obedece ao seguinte raciocínio: dado um número natural *x* qualquer, se *x* é código de um programa de Turing, então P_x, o *x*-ésimo programa da lista, será Z_x; se *x* não é código de um programa de Turing, P_x será o programa $\{\mathrm{q}_0||\mathrm{Lq}_0, \mathrm{q}_0\mathrm{BBLq}_0\}$ que computa a função vazia, indefinida para quaisquer argumentos.

A respeito desta lista, convém explicitar alguns aspectos a ela associados:

(1) sua construção é efetiva: dado um número natural *x*, podemos determinar mecanicamente se *x* é ou não código de um programa de Turing; em caso afirmativo, podemos, via decomposição de *x*, recuperar precisamente o programa que *x* codifica e, em ambos os casos, somos capazes de identificar, também mecanicamente, o programa P_x.

(2) ela comporta a possibilidade de repetições de programas: de acordo com a codificação apresentada, um programa com n quíntuplas, possui n! códigos; sendo assim, ele ocorrerá n! vezes na lista; além disso, o programa $\{q_0||Lq_0, q_0BBLq_0\}$ ocorrerá na lista todas as vezes em que *x* não for código de programa.

(3) ela torna possível a elaboração de uma listagem efetiva de todas as funções parcialmente Turing-computáveis: segundo a definição 2.10, um programa computa uma única função n-ária para cada n ≥ 1, então a partir da lista de programas, podemos elaborar, também para cada n ≥ 1, uma lista de funções n-árias parcialmente Turing-computáveis (como vemos abaixo) na qual φ_e^k denota a função k-ária (k ≥ 1) computada segundo o programa P_e.

$$1.\ \varphi_0^1, \varphi_1^1, \varphi_2^1, \varphi_3^1, \ldots$$

$$2.\ \varphi_0^2, \varphi_1^2, \varphi_2^2, \varphi_3^2, \ldots$$

$$3.\ \varphi_0^3, \varphi_1^3, \varphi_2^3, \varphi_3^3, \ldots$$

$$\vdots$$

$$n.\ \varphi_0^n, \varphi_1^n, \varphi_2^n, \varphi_3^n, \ldots$$

$$\vdots$$

Atentos às convenções estabelecidas, cada função φ_e^k acima listada ou é a função vazia (quando o índice subscrito não é código de um programa de Turing) ou é uma função parcialmente Turing-computável diferente da função vazia (quando ocorre o caso contrário).

Os índices de uma função serão os índices do programa que a computa. No entanto, seria um engano pensar que a função φ_e^n computada pelo programa P_e, que possui um número finito de índices, possuiria unicamente os mesmos e a mesma quantidade de índices de P_e. Com efeito, basta acrescentar ao conjunto P_e quíntuplas inutilizáveis nas computações realizadas de acordo com tal conjunto para obtermos novos programas diferentes de P_e que computarão, da mesma forma, a função φ_e^n. Deste modo, uma mesma função pode ser computada por mais de um programa e, portanto, os seus índices não se limitarão somente aos

índices de um único programa que a computa. Dentro desta perspectiva, apresentamos o teorema seguinte.

Lema 3.2 (*Padding Lemma*) Cada função parcial recursiva φ_x possui $\aleph_0$ índices distintos e, além disso, para cada *x*, é possível determinar efetivamente um conjunto infinito de índices para φ_x.

Prova:

Sendo P_x um programa qualquer, considere:

$$\pi(x, 0) = P_x$$

$$\pi(x, y+1) = \pi(x, y) \cup \{q_{\theta(\pi(x,y))+1} || Lq_{\theta(\pi(x,y))+1}\}$$

Cada programa $\pi(x, y)$, para $y \geq 1$, é obtido pelo acréscimo de uma quíntupla inutilizável nas computações de acordo com P_x ao programa anterior $\pi(x, y\text{-}1)$. Deste modo, obtemos um conjunto infinito enumerável de programas - $\pi(x, 0)$, $\pi(x, 1)$, $\pi(x, 2)$, ... - que embora distintos computam a função φ_x. Portanto, φ_x possui $\aleph_0$ índices distintos, cada um dos quais pode ser efetivamente determinado pela seguinte função:

$$f(x, 0) = \begin{cases} U_1^1(x), & \text{se } PT(x) \\ 2^{2^9 \cdot 3^{11} \cdot 5^{11} \cdot 7^5 \cdot 11^9} \cdot 3^{2^9 \cdot 3^7 \cdot 5^7 \cdot 7^5 \cdot 11^9}, & \text{se } \sim PT(x) \end{cases}$$

$$f(x, y+1) = f(x, y) * 2^{2^{\alpha(f(x,y))+4} \cdot 3^{11} \cdot 5^{11} \cdot 7^5 \cdot 11^{\alpha(f(x,y))+4}},$$

sendo $\alpha(f(x, y)) = \mu_{z<f(x,y)}(\bigvee_{i<lh(f(x,y))} (((f(x,y))_i)_0 = z \vee ((f(x,y))_i)_4 = z \wedge \bigwedge_{j<lh(f(x,y))}(((f(x,y))_j)_0 \leq z \wedge ((f(x,y))_j)_4 \leq z))$. Em outras palavras, $\alpha(f(x, y))$ é o código do maior estado interno de $f(x, y)$.

Tal como foi definida, nota-se claramente que a função *f* é recursiva, ou seja, *f* está definida para todos os argumentos e cada um dos seus valores é obtido algoritmicamente. Sendo assim, para cada *x*, podemos achar um conjunto infinito enumerável de índices para φ_x.

3.2 Teorema *s-m-n* de Kleene

Nesta seção, teremos como principal propósito demonstrar um importante teorema da Teoria da Computabilidade, o chamado Teorema s-m-n de Kleene. Nossa demonstração será construída mediante dois lemas, que provaremos a seguir.

Lema 3.3 (a) Para cada número natural *n*, existe um programa de Turing P_n com estados internos $q_0, q_1, ..., q_{n+2}$, tal que $Res_{P_n}[q_0| t] = q_{n+2}, \overline{n}B|t$, sendo t qualquer expressão da fita; (b) A função *a*(*n*), que determina um dos códigos de P_n, é recursiva primitiva.

Prova:

(a) Devemos observar que a máquina comandada pelo programa P_n, achando-se no estado interno q_0 e observando o traço mais à esquerda do *input*, vai para o quadrado imediatamente anterior (que permanecerá vazio), à esquerda dele imprime n+1 traços e assume o estado interno q_{n+2}.

Neste cenário, o programa P_n é estabelecido recursivamente de acordo com as seguintes cláusulas:

$$P_0 = \{q_0||Lq_0, q_0BBLq_1, q_1B|Lq_2, q_2BBRq_2\}$$

$$P_{n+1} = P_n \cup \{q_{n+2}||Lq_{n+2}, q_{n+2}B|Lq_{n+3}, q_{n+3}BBRq_{n+3}\}$$

(b) Uma sequência recursiva primitiva para a função *a*(*n*) pode ser obtida a partir das seguintes cláusulas:

$$a(0) = 2^{2^9 \cdot 3^{11} \cdot 5^{11} \cdot 7^5 \cdot 11^9} \cdot 3^{2^9 \cdot 3^7 \cdot 5^7 \cdot 7^5 \cdot 11^{13}} \cdot 5^{2^{13} \cdot 3^7 \cdot 5^{11} \cdot 7^5 \cdot 11^{17}} \cdot 7^{2^{17} \cdot 3^7 \cdot 5^7 \cdot 7^3 \cdot 11^{17}}$$

$$a(n+1) = a(0) * \prod_{i \leq n} [p_{2i}^{2^{13+4(i+1)} \cdot 3^{11} \cdot 5^{11} \cdot 7^5 \cdot 11^{13+4(i+1)}} \cdot p_{2i+1}^{2^{13+4(i+1)} \cdot 3^7 \cdot 5^{11} \cdot 7^5 \cdot 11^{17+4(i+1)}} \cdot p_{2i+2}^{2^{17+4(i+1)} \cdot 3^7 \cdot 5^7 \cdot 7^3 \cdot 11^{17+4(i+1)}}]$$

Lema 3.4: A função S(*e*, *n*), que determina um dos códigos de $P_e^{(n)}$, é recursiva primitiva.

Prova:

A título de recordação, $P_e^{(n)}$ é o programa de Turing obtido a partir da substituição de todas as ocorrências de q_i em P_e por q_{i+n}. Um de seus códigos é determinado pela função S(*e*, *n*) que, como podemos ver abaixo, é claramente recursiva primitiva.

$$S(e, n) = (\chi_{PT}(e) \cdot \prod_{i \leq lh(e)} p_{2i}^{2^{((e)_i)_0+4n} \cdot 3^{((e)_i)_1} \cdot 5^{((e)_i)_2} \cdot 7^{((e)_i)_3} \cdot 11^{((e)_i)_4+4n}}) + (\overline{sg}(\chi_{PT}(e)) \cdot 2^{2^{9+4n} \cdot 3^{11} \cdot 5^{11} \cdot 7^5 \cdot 11^{9+4n}} \cdot 3^{2^{9+4n} \cdot 3^7 \cdot 5^7 \cdot 7^5 \cdot 11^{9+4n}})$$

Provados os lemas anteriores, podemos, agora, demonstrar o Teorema s-m-n de Kleene.

Teorema 3.5 (*Teorema s-m-n de Kleene*) Para cada *m*, $n \geq 1$, existe uma função recursiva primitiva s_n^m de *m*+1 variáveis, tal que, para todo *e*, $x_1, \ldots, x_m$,

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

sendo $x_1, \ldots, x_m$ parâmetros.

Prova:

Dito de outro modo, o teorema s-m-n de Kleene garante que, sendo P_e um programa de Turing qualquer e $x_1, \ldots, x_m$ parâmetros, existe um programa de Turing Z cujo índice é $s_n^m(e,\ x_1, \ldots,\ x_m)$ tal que:

$$Res_{P_e}[\mathrm{q}_0\overline{x}_1\mathrm{B}...\mathrm{B}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n] = Res_Z[\mathrm{q}_0\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n]$$

O primeiro passo da prova consiste em elaborar um programa Z com os seguintes comandos: observando o *input* $\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n$, escreva o argumento $\overline{x}_1\mathrm{B}...\mathrm{B}\overline{x}_m$ à esquerda de $\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n$, obtendo como resultado a expressão $\overline{x}_1\mathrm{B}...\mathrm{B}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n$; por fim, aplique a esta expressão o programa P_e. O passo seguinte consiste em mostrar que a função $s_n^m(e,\ x_1, \ldots,\ x_m)$, pela qual se obtém um dos índices do programa Z, é recursiva primitiva.

Sejam Z_{x_m}, ..., Z_{x_1} programas de Turing obtidos de acordo com o lema 3.3. Z_{x_m}instrui a máquina a escrever $\overline{x}_m$ à esquerda de $\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n$, de modo que:

$$Res_{Z_{x_m}}[\mathrm{q}_0\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n] = \mathrm{q}_{x_m+2}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n$$

A partir de $Z_{x_{m-1}}$, determine $Z_{x_{m-1}}^{(x_{m+2})}$. O símbolo de estado interno de $\mathrm{c_t}^{Z_{x_{m-1}}^{(x_{m+2})}} = \mathrm{q}_{(x_{m-1}+2)+x_m+2} = \mathrm{q}_{x_m+x_{m-1}+4}$. A máquina regida pelo programa $Z_{x_{m-1}}^{(x_{m+2})}$ escreve $\overline{x}_{m-1}$ à esquerda de $\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n$, de tal maneira que:

$$Res_{Z_{x_{m-1}}^{(x_{m+2})}}[\mathrm{q}_{x_m+2}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n] = \mathrm{q}_{x_m+x_{m-1}+4}\overline{x}_{m-1}\mathrm{B}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n$$

De modo análogo, a partir de $Z_{x_{m-2}}$, determine $Z_{x_{m-2}}^{(x_m+x_{m-1}+4)}$. O símbolo de estado interno de $\mathrm{c_t}^{Z_{x_{m-2}}^{(x_m+x_{m-1}+4)}} = \mathrm{q}_{(x_{m-2}+2)+x_m+x_{m-1}+4} = \mathrm{q}_{x_m+x_{m-1}+x_{m-2}+6}$. Sob os comandos de $Z_{x_{m-2}}^{(x_m+x_{m-1}+4)}$, a máquina imprime $\overline{x}_{m-2}$ à esquerda de $\overline{x}_{m-1}\mathrm{B}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n$, de modo que:

$$Res_{Z_{x_{m-2}}^{(x_m+x_{m-1}+4)}}[\mathrm{q}_{x_m+x_{m-1}+4}\overline{x}_{m-1}\mathrm{B}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n] =$$

$$\mathrm{q}_{x_m+x_{m-1}+x_{m-2}+6}\overline{x}_{m-2}\mathrm{B}\overline{x}_{m-1}\mathrm{B}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}...\mathrm{B}\overline{y}_n$$

Para cada Z_{x_i}, sendo $m \geq i \geq 1$, executamos o procedimento descrito acima. O programa que escreverá, finalmente, $\overline{x}_1$ na extremidade esquerda será $Z_{x_1}^{(x_m+\ \ldots+x_2+2m-2)}$. Sendo assim, teremos que:

$$Res_{Z_{x_1}^{(x_m+\ \ldots+x_2+2m-2)}}[\mathrm{q}_{x_m+\ \ldots+x_2+2m-2}\overline{x}_2\mathrm{B}\ \ldots\ \mathrm{B}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}\ldots\mathrm{B}\overline{y}_n] =$$

$$\mathrm{q}_{x_m+\ldots+x_1+2m}\overline{x}_1\mathrm{B}\ \ldots\ \mathrm{B}\overline{x}_m\mathrm{B}\overline{y}_1\mathrm{B}\ldots\mathrm{B}\overline{y}_n$$

Considere $Z = Z_{x_m} \cup Z_{x_{m-1}}^{(x_{m+2})} \cup Z_{x_{m-2}}^{(x_m+x_{m-1}+4)} \cup \ \ldots\ \cup Z_{x_1}^{(x_m+\ \ldots+x_2+2m-2)} \cup P_e^{(x_m+\ \ldots+x_1+2m)}$. Definido desta forma, Z é o programa de Turing que instrui a máquina a realizar a computação desejada.

De acordo com os lemas 3.3 e 3.4, os programas constituintes de Z, na ordem em que foram apresentados possuem, respectivamente, $a(x_m)$, S($a(x_{m-1})$, x_{m+2}), S($a(x_{m-2})$, $x_m + x_{m-1} + 4$), ..., S($a(x_1)$, $x_m +\ \ldots + x_2 + 2m - 2$) e S($e$, $x_m +\ \ldots + x_1 + 2m$) como um de seus códigos. Sabendo disso, um dos códigos do programa Z é o valor da função $s_n^m(e,\ x_1, \ldots,\ x_m)$, definida como segue:

$s_n^m(e,\ x_1, \ldots,\ x_m) = a(x_m) * \mathrm{S}(a(x_{m-1}), x_{m+2}) * \mathrm{S}(a(x_{m-2}), x_m + x_{m-1} + 4) * \ldots * \mathrm{S}(a(x_1),$ $x_m +\ \ldots + x_2 + 2m - 2) * \mathrm{S}(e, x_m +\ \ldots + x_1 + 2m)$

O caráter recursivo primitivo da operação de concatenação e das funções obtidas pelos lemas 3.3 e 3.4 garante, evidentemente, que a função $s_n^m(e,\ x_1, \ldots,\ x_m)$ é recursiva primitiva.

O Teorema s-m-n de Kleene é um dos mais utilizados em Teoria da Computabilidade. Em geral, suas aplicações, como veremos mais adiante, seguem, normalmente, o mesmo modelo: apresentamos, em primeiro lugar, uma função parcial algorítmica Ψ (m+n)-ária e, concluímos, via Tese de Church, que Ψ é parcial recursiva, isto é, Ψ = φ_e^{m+n}; por fim, utilizando o Teorema s-m-n, concluímos que existe uma função recursiva primitiva s_n^m tal que $\varphi_e^{m+n}(x_1, \ldots, x_m, y_1, \ldots, y_n) = \varphi_{s_n^m(e,\ x_1, \ldots, x_m)}^n(y_1, \ldots, y_n)$. Embora a função s_n^m possua *m*+1 variáveis, este número pode ser reduzido a *m* variáveis: basta definirmos, por composição, uma função *m*-ária f tal que $f(x_1, \ldots, x_m) = s_n^m(C_e^1(U_i^m(x_1, \ldots,\ x_m)), U_1^m(x_1, \ldots,$ $x_m), \ldots, U_m^m(x_1, \ldots,\ x_m))$. Procedendo desta maneira, suprimimos o índice *e*, deduzindo, analogamente, pelo Teorema s-m-n, que existe uma função recursiva primitiva f tal que $\varphi_e^{m+n}(x_1, \ldots, x_m, y_1, \ldots, y_n) = \varphi_{f\ (x_1, \ldots, x_m)}^n(y_1, \ldots, y_n)$.

3.3 Problema da Parada

Considerando a noção intuitiva de algoritmo, introduzimos o problema da decisão para

uma relação numérica n-ária R: Dada uma *n*-upla ordenada $(x_1, \dots, x_n)$ qualquer, há um algoritmo que nos permita sempre decidir se $(x_1, \dots, x_n) \in$ R ou $(x_1, \dots, x_n) \notin$ R[4]? Ou simplesmente: há um algoritmo para computar a função característica de R? Se houver, afirmamos que o problema da decisão para R é algoritmicamente solúvel e que, portanto, R é decidível. Caso contrário, dizemos que o problema da decisão é algoritmicamente insolúvel ou que R é indecidível. De modo menos intuitivo, podemos estabelecer, via Tese de Church, a seguinte definição:

Definição 3.6 O problema da decisão para uma relação *n*-ária R é *recursivamente solúvel* se, e somente se, R é recursiva. Do contrário, o problema é *recursivamente insolúvel*.

Entre os diversos casos existentes de insolubilidade recursiva, um dos mais famosos é o Problema da Parada. Antes de apresentá-lo, vamos relembrar as duas situações alternativas que podem ocorrer quando alimentamos uma máquina de Turing com um determinado *input*: (1) ou a máquina, de acordo com um programa previamente dado, executa a computação do *input* e para após algum intervalo de tempo, fornecendo um *output* ou (2) ela entra em *loop*, isto é, continuará operando "eternamente" sem nunca atingir um termo. Neste caso, um *output* jamais será alcançado. Portanto, das duas, uma: ou a máquina para ou não para.

Neste cenário, surge o Problema da parada, expresso, informalmente, pela seguinte pergunta: existe um algoritmo que nos permita sempre decidir, dada uma máquina de Turing alimentada com um *input*, sobre o qual ela opera, se ela para ou não? Em uma versão menos informal: existe um algoritmo para decidir se, dados x e *y*, o programa P_y aplicado ao *input* x gera um *output*? Finalmente, em termos estritamente formais: o problema da decisão para o conjunto K = $\{(x, y) : \varphi_x(y) \downarrow\}$ é recursivamente solúvel? Na literatura especializada, o mesmo problema é, normalmente, apresentado em sua versão autorreferente: existe um algoritmo para decidir se, dado x, o programa P_x aplicado ao *input* x gera um *output*? O problema da decisão para o conjunto K = $\{x : \varphi_x(x) \downarrow\}$ é recursivamente solúvel?

O próximo teorema, como veremos, estabelecerá a insolubilidade recursiva do Problema da parada. Este é um dos resultados negativos mais importantes da Teoria da Computabilidade, a partir do qual podemos, facilmente, estabelecer a insolubilidade recursiva de inúmeros problemas de decisão.

Teorema 3.7 O Problema da parada é recursivamente insolúvel.

[4] Um algoritmo com esta característica é também conhecido como procedimento de decisão.

Prova (por redução ao absurdo):

Considere a seguinte lista de todas as funções parciais recursivas unárias e de seus possíveis valores:

$$\begin{array}{cccc} \varphi_0(0) & \varphi_0(1) & \varphi_0(2) & \ldots \\ \varphi_1(0) & \varphi_1(1) & \varphi_1(2) & \ldots \\ \varphi_2(0) & \varphi_2(1) & \varphi_2(2) & \ldots \\ \vdots & \vdots & \vdots & \end{array}$$

Suponhamos, por absurdo, que K é recursivo ou, equivalentemente, que a função característica de K é recursiva:

$$\chi_K(x,y) = \begin{cases} 1, & \text{se } \varphi_x(y) \downarrow \\ 0, & \text{se } \varphi_x(y) \uparrow \end{cases}$$

Agora, a partir de $\chi_K(x,y)$, definimos a seguinte função:

$$\Psi(x) = \begin{cases} 1, & \text{se } \chi_K(x,x) = 0 \\ \uparrow, & \text{se } \chi_K(x,x) = 1 \end{cases}$$

Evidentemente, Ψ é parcial algorítmica. Então, pela Tese de Church, Ψ é parcial recursiva e, portanto, $\Psi(x)$ é uma das funções da lista acima, isto é, $\Psi(x) = \varphi_i(x)$, para algum $i \geq 0$. Sendo assim, $\Psi(i) = \varphi_i(i)$. Neste caso, derivamos as seguintes consequências:

Por um lado,

$$\varphi_i(i) \uparrow \Rightarrow \chi_K(i,i) = 0 \Rightarrow \Psi(i) = 1 \Rightarrow \varphi_i(i) = 1 \Rightarrow \varphi_i(i) \downarrow$$

Por outro lado,

$$\varphi_i(i) \downarrow \Rightarrow \chi_K(i,i) = 1 \Rightarrow \Psi(i) \uparrow \Rightarrow \varphi_i(i) \uparrow$$

Logo, alcançamos o seguinte absurdo:

$$\varphi_i(i) \downarrow \Leftrightarrow \varphi_i(i) \uparrow$$

Portanto, a nossa suposição inicial de que K é recursivo é falsa. Consequentemente, o Problema da parada é recursivamente insolúvel.

Corolário 3.8 O Problema da parada, em sua versão autorreferente, é recursivamente insolúvel.

Prova:

Análoga a prova do teorema anterior.

3.4 Teorema de Rice

O Teorema de Rice tem uma grande força dedutiva. A partir dele, como veremos, podemos inferir que qualquer conjunto de índices de um subconjunto próprio das funções parciais recursivas que não seja vazio é indecidível. O motivo pelo qual o Teorema de Rice trata diretamente dos índices de funções em vez das funções propriamente ditas ficará claro mais adiante.

Definição 3.9 Seja C um conjunto qualquer de funções parciais recursivas. O conjunto $\mathrm{I_C} = \{x: \varphi_x \in \mathrm{C}\}$ denomina-se o *conjunto dos índices de C*.

Como consequência imediata da definição acima, note-se que se $x \in \mathrm{I_C}$ e $\varphi_x = \varphi_y$, então $y \in \mathrm{I_C}$. Portanto, se $\varphi \in \mathrm{C}$, então $\mathrm{I_C}$ contém todos os índices de φ.

Teorema 3.10 (*Teorema de Rice*) Seja FpR o conjunto de todas as funções parciais recursivas unárias e seja C ⊆ FpR. O problema da decisão para $\mathrm{I_C}$ é recursivamente solúvel se, e somente se, C = ∅ ou C = FpR.

Antes de provarmos o teorema acima, convém destacar que ele, em seu enunciado, explicita apenas as funções parciais recursivas unárias. Isto, no entanto, não limita o seu alcance, pois podemos reduzir todas as funções parciais recursivas n-árias, com $n \geq 2$, à funções parciais recursivas de uma variável, através da função J de Cantor[5] que nos possibilita codificar n-uplas ordenadas. Deste modo, os resultados do Teorema de Rice se estendem, implicitamente, a funções parciais recursivas de qualquer aridade.

Prova:

Inicialmente, provaremos a segunda direção da bicondicional. Suponhamos que C = ∅ ou C = FpR. Por um lado, se C = ∅, então $\mathrm{I_C} = \emptyset$. Como sabemos, a função característica do conjunto vazio, $C_0^1(x)$, é recursiva. Por outro lado, se C = FpR, então $\mathrm{I_C} = \mathbb{N}$. Sabemos também que a função característica do conjunto dos números naturais, $C_1^1(x)$, é recursiva. Portanto, em ambos os casos, o problema da decisão para $\mathrm{I_C}$ é recursivamente solúvel.

[5] Uma exposição detalhada dessa função pode ser encontrada em DIAS, M. F.; LIMA, L. W. C. Teoria da recursão. São Paulo: Editora UNESP, 2010, p.134.

Agora, por redução ao absurdo, provaremos a primeira direção da bicondicional. Suponhamos que o problema da decisão para I_C é recursivamente solúvel, ou seja, suponhamos que existe uma função recursiva χ tal que:

$$\chi(x) = \begin{cases} 1, & \text{se } \varphi_x \in C \\ 0, & \text{se } \varphi_x \notin C \end{cases}$$

Sendo Ψ' a função vazia, suponhamos, por absurdo, que $C \neq \emptyset$ e $C \neq FpR$. Então, ou $\Psi' \in C$ ou $\Psi' \notin C$. Analisemos, separadamente, ambos os casos, a fim de derivarmos em cada um deles uma contradição.

1º caso: Admitamos que $\Psi' \in C$. Como $C \neq FpR$, existe $\varphi \in FpR - C$, tal que $\Psi' \neq \varphi$. Sabendo disso, considere a seguinte função:

$$\Psi_1(x,y) = \begin{cases} \varphi(y), & \text{se } \varphi_x(x) \downarrow \\ \uparrow, & \text{se } \varphi_x(x) \uparrow \end{cases}$$

Tal como foi definida, Ψ_1 é, claramente, parcial algorítmica. Então, dada a Tese de Church, Ψ_1 é parcial recursiva. Logo, existe um índice *e*, tal que $\Psi_1 = \varphi_e$. Pelo Teorema s-m-n de Kleene, $\varphi_e(x,y) = \varphi_{h(x)}(y)$, sendo *h* uma função recursiva primitiva. Mas se este é o caso, as seguintes consequências são válidas:

Por um lado,

$$\varphi_x(x) \downarrow \Rightarrow$$
$$\text{para todo } y, \Psi_1(x,y) = \varphi(y) \Rightarrow \text{para todo } y,\ \varphi_e(x,y) = \varphi(y) \Rightarrow \text{para todo } y, \varphi_{h(x)}(y) =$$
$$\varphi(y) \Rightarrow \varphi_{h(x)} = \varphi \Rightarrow \varphi_{h(x)} \in FpR - C \Rightarrow \varphi_{h(x)} \notin C$$

Por outro lado,

$$\varphi_x(x) \uparrow \Rightarrow \text{para todo } y, \Psi_1(x,y) \uparrow \Rightarrow \text{para todo } y,\ \varphi_e(x,y) \uparrow \Rightarrow \text{para todo } y, \varphi_{h(x)}(y) \uparrow \Rightarrow$$
$$\varphi_{h(x)} = \Psi' \Rightarrow \varphi_{h(x)} \in C$$

Ora, em resumo:

$$\varphi_x(x) \downarrow \Rightarrow \varphi_{h(x)} \notin C \wedge \varphi_x(x) \uparrow \Rightarrow \varphi_{h(x)} \in C$$

A partir dessa conjunção, podemos inferir, pela definição dos conectivos proposicionais, as equivalências abaixo:

$$\varphi_x(x) \downarrow \Leftrightarrow \varphi_{h(x)} \notin C$$

$$\varphi_x(x) \uparrow \Leftrightarrow \varphi_{h(x)} \in \mathrm{C}$$

Agora, considere a seguinte função recursiva obtida por composição:

$$\chi(h(x)) = \begin{cases} 1, & \text{se } \varphi_{h(x)} \in \mathrm{C} \\ 0, & \text{se } \varphi_{h(x)} \notin \mathrm{C} \end{cases}$$

Ou, de acordo com as equivalências anteriores:

$$\chi(h(x)) = \begin{cases} 1, & \text{se } \varphi_x(x) \uparrow \\ 0, & \text{se } \varphi_x(x) \downarrow \end{cases}$$

Sendo assim, note-se que $g(x) = \overline{sg}(\chi(h(x))$ seria uma função característica recursiva para K. Isto, porém, é um absurdo, pois, como já vimos, o corolário 3.8 garante que o conjunto K não é recursivo.

2º caso: Suponhamos que $\Psi' \notin \mathrm{C}$. Como $\mathrm{C} \neq \emptyset$, existe $\varphi \in \mathrm{C}$, tal que $\Psi' \neq \varphi$. Sabendo disso, considere a seguinte função:

$$\Psi_2(x, y) = \begin{cases} \varphi(y), & \text{se } \varphi_x(x) \downarrow \\ \uparrow, & \text{se } \varphi_x(x) \uparrow \end{cases}$$

Tal como foi definida, Ψ_2 é, nitidamente, parcial algorítmica. Então, considerando a Tese de Church, Ψ_2 é parcial recursiva. Logo, há um índice e', tal que $\Psi_2 = \varphi_{e'}$. Dado o Teorema s-m-n de Kleene, $\varphi_{e'}(x, y) = \varphi_{h'(x)}(y)$, para uma função recursiva primitiva h'. Mas se este é o caso, as seguintes consequências são válidas:

Por um lado,

$$\varphi_x(x) \downarrow \Rightarrow$$
$$\text{para todo } y, \Psi_2(x, y) = \varphi(y) \Rightarrow \text{para todo } y,$$
$$\varphi_{e'}(x, y) = \varphi(y) \Rightarrow \text{para todo } y, \varphi_{h'(x)}(y) = \varphi(y) \Rightarrow \varphi_{h'(x)} = \varphi \Rightarrow \varphi_{h'(x)} \in \mathrm{C}$$

Por outro lado,

$$\varphi_x(x) \uparrow \Rightarrow \text{para todo } y, \Psi_2(x, y) \uparrow \Rightarrow \text{para todo } y,\ \varphi_{e'}(x, y) \uparrow \Rightarrow \text{para todo } y, \varphi_{h'(x)}(y) \uparrow$$
$$\Rightarrow \varphi_{h'(x)} = \Psi' \Rightarrow \varphi_{h'(x)} \notin \mathrm{C}$$

Ora, em resumo:

$$\varphi_x(x) \downarrow \Rightarrow \varphi_{h'(x)} \in \mathrm{C} \wedge \varphi_x(x) \uparrow \Rightarrow \varphi_{h'(x)} \notin \mathrm{C}$$

A partir desta conjunção, podemos inferir, pela definição dos conectivos proposicionais, as equivalências abaixo:

$$\varphi_x(x)\downarrow \Leftrightarrow \varphi_{h'(x)} \in \mathrm{C}$$

$$\varphi_x(x)\uparrow \Leftrightarrow \varphi_{h'(x)} \notin \mathrm{C}$$

Agora, considere a seguinte função recursiva obtida por composição:

$$\chi(h'(x)) = \begin{cases} 1, & \text{se } \varphi_{h'(x)} \in \mathrm{C} \\ 0, & \text{se } \varphi_{h'(x)} \notin \mathrm{C} \end{cases}$$

Ou, de acordo com as equivalências anteriores:

$$\chi(h'(x)) = \begin{cases} 1, & \text{se } \varphi_x(x)\downarrow \\ 0, & \text{se } \varphi_x(x)\uparrow \end{cases}$$

Sendo assim, note-se que $\chi(h'(x))$ seria uma função característica recursiva para K. Isto, porém, é um absurdo, pois, como já vimos, o corolário 3.8 assegura que o conjunto K não é recursivo.

Por fim, em ambos os casos, obtivemos uma contradição, o que garante que a nossa hipótese segundo a qual C ≠ ∅ e C ≠ FpR, admitindo C como um conjunto recursivo, é falsa. Portanto, como queríamos provar, o problema da decisão para I_C é recursivamente solúvel se, e somente se, C = ∅ ou C = FpR.

Como consequência imediata do Teorema de Rice, podemos obter inúmeros resultados de insolubilidade recursiva, tendo em vista que todo conjunto de índices de qualquer subconjunto próprio de FpR que não seja vazio não é recursivo.

Corolário 3.11 Sejam FRP e FR, respectivamente, o conjunto das funções recursivas primitivas e o conjunto das funções recursivas. O problema da decisão para os conjuntos $I_{FRP} = \{x: \varphi_x \in \mathrm{FRP}\}$ e $I_{FR} = \{x: \varphi_x \in \mathrm{FR}\}$ é recursivamente insolúvel.

Prova:

Os conjuntos FRP e FR são subconjuntos próprios de FpR diferentes do ∅. Então, o problema da decisão para I_{FRP} e I_{FR} é recursivamente insolúvel.

Apesar das consequências do Teorema de Rice, poderíamos deparar-nos com a argumentação apresentada nos parágrafos seguintes, segundo a qual o conjunto dos índices das funções recursivas primitivas é recursivo.

Assim como codificamos os programas de Turing, podemos também determinar uma codificação com características semelhantes para as derivações parciais recursivas, de modo que cada uma delas tenha um código. Neste contexto, dado um número qualquer, decidimos mecanicamente se ele é ou não código de uma derivação parcial recursiva. Em caso afirmativo, podemos decodificá-lo e recuperar a derivação que ele codifica. Por sua vez, dada uma derivação parcial recursiva, determinamos mecanicamente se ela é ou não uma derivação recursiva primitiva. Sendo assim, podemos definir tanto o conjunto de todos os números que são códigos de derivações recursivas primitivas quanto o conjunto de todos os números que são códigos de derivações parciais recursivas que não são recursivas primitivas.

De modo análogo ao que foi apresentado no início deste capítulo, poderíamos, por exemplo, indexar cada função parcial recursiva com o código de sua respectiva derivação. Neste caso, o código de uma derivação recursiva primitiva seria o índice da função recursiva primitiva que ela deriva. Sabendo disso e considerando que conjunto dos códigos das derivações recursivas primitivas é recursivo, alguém poderia, erroneamente, alegar que o conjunto dos índices das funções recursivas primitivas também é recursivo. Assim, dado um número qualquer, sendo ele código de uma derivação parcial recursiva, poderíamos decidir mecanicamente se ele seria ou não índice de uma função recursiva primitiva. E, portanto, contrariando o Teorema de Rice, o problema da decisão para o conjunto dos índices das funções recursivas primitivas seria recursivamente solúvel.

Esse resultado, à primeira vista, poderia ser bastante convincente se não fosse um pequeno detalhe: o conjunto dos códigos de todas as derivações recursivas primitivas não nos oferece todos os índices possíveis de uma função recursiva primitiva, pois podemos ter uma derivação parcial recursiva não recursiva primitiva cuja última função seja recursiva primitiva e, portanto, essa função terá como um de seus índices um número que não é código de uma derivação recursiva primitiva. Sendo assim, o conjunto dos índices das funções recursivas primitivas é mais abrangente que o conjunto dos códigos das derivações recursivas primitivas. A rigor, o conjunto que nos oferece todos os índices de funções recursivas primitivas é aquele que contém os códigos de todas as derivações parciais recursivas que derivam uma função recursiva primitiva, sejam elas recursivas primitivas ou não. E esse conjunto, como o Teorema de Rice demonstra, é claramente indecidível. Tal resultado é, intuitivamente, explícito: com efeito, dada uma derivação parcial recursiva (ou um programa de Turing) não é possível decidir, em geral, se a função que ela deriva (ou que ele computa) é recursiva primitiva.

Uma síntese dessas considerações é apresentada no esquema abaixo:

C_{DpR} = {i : i é código de uma derivação parcial recursiva}

C_{DRP} = {i : i é código de uma derivação recursiva primitiva}

I_{FpR}= {x: φ_x é parcial recursiva}

I_{FRP} = {x: φ_x é recursiva primitiva}

$C_{DpR} = I_{FpR} = \mathbb{N}$

$C_{DRP} \subset I_{FRP}$

C_{DpR} é recursivo e I_{FpR} é recursivo.

C_{DRP} é recursivo, mas I_{FRP} não é recursivo (corolário 3.11).

A falsa ideia de que o problema da decisão para o conjunto dos índices das funções recursivas primitivas é recursivamente solúvel provém do fato de, considerando a indexação apresentada, admitir-se equivocadamente como iguais dois conjuntos que, na verdade, são diversos, a saber: o conjunto dos códigos das derivações recursivas primitivas e o conjunto dos índices das funções recursivas primitivas. O primeiro, como vimos, é decidível, ao passo que o segundo, por sua vez, é indecidível e mais abrangente que o primeiro.

Estabelecemos a aritmetização da teoria das máquinas e dos programas de Turing e obtivemos, como consequência imediata do Teorema de Rice, a indecidibilidade dos conjuntos de índices das funções recursivas e das funções recursivas primitivas. Com efeito, o Teorema de Rice trata diretamente dos índices de funções em vez das funções por eles indexadas. Não é estranha esta abordagem. De fato, parece haver uma certa impropriedade em se investigar se um conjunto de funções propriamente dito é ou não decidível, pois devemos lembrar que tal investigação será protagonizada por uma máquina, cujas computações devem partir de *inputs* finitos. Uma função, ao contrário, pode ser um conjunto infinito de *n*-uplas, possibilidade que inviabilizaria a investigação. Neste contexto, para que a máquina seja minimamente capaz de nos responder se uma função pertence ou não a um determinado conjunto devemos então substituir o *input*: no lugar da função apresentamos, sob um determinado formalismo, uma de suas descrições. Deste modo, a máquina investigará, a rigor, se a descrição dada pertence ao conjunto das descrições que especificam uma função do conjunto em questão. Posto que os índices, tais como definimos, constituem um exemplo particular dessas descrições, podemos estabelecer as seguintes equivalências: dada uma função numérica f qualquer, $f \in \{f: f$ é parcial recursiva$\}$ se, e somente se, $\exists x(f = \varphi_x \wedge$

$x \in \mathrm{I}_{\mathrm{FpR}}\}$ ou $f \in \{f: f$ é recursiva primitiva$\}$ se, e somente se, $\exists x(f = \varphi_x \wedge x \in \mathrm{I}_{\mathrm{FRP}}\}$ou, ainda, $f \in \{f: f$ é recursiva$\}$ se, e somente se, $\exists x(f = \varphi_x \wedge x \in \mathrm{I}_{\mathrm{FR}})$. Ora, como pelo Teorema de Rice, $\mathrm{I}_{\mathrm{FpR}}$ é decidível e pelo corolário 3.11, $\mathrm{I}_{\mathrm{FRP}}$ e I_{FR} são indecidíveis, concluímos, indiretamente, que o conjunto das funções parciais recursivas é decidível, enquanto que o conjunto das funções recursivas e o conjunto das funções recursivas primitivas são indecidíveis, alcançando, com este resultado, o último objetivo ao qual nos propomos. Portanto, embora haja um algoritmo para computar as funções recursivas (primitivas), não há um algoritmo para decidir se uma função é ou não recursiva (primitiva).

CONCLUSÃO

O *Entscheindungsproblem* trouxe à tona a necessidade de se precisar as noções intuitivas de algoritmo e função algorítmica, abrindo espaço para a instauração da Teoria da Computabilidade. Acredita-se que estas noções tenham sido, rigorosamente, caracterizadas sob diversos formalismos, dentre os quais apresentamos as funções parciais recursivas e as funções parcialmente Turing-computáveis, acompanhadas de seus respectivos algoritmos: as derivações parciais recursivas e as máquinas de Turing. Lógicos e matemáticos demonstraram que todos os formalismos oferecidos para as referidas noções intuitivas eram equivalentes. Essa equivalência ficou conhecida como *Resultado fundamental* da Teoria da Computabilidade e constitui, até hoje, um dos argumentos fortes a favor da Tese de Church-Turing que, apesar de não ser matematicamente provada, é largamente aceita.

A partir da definição de função recursiva, noção fundamental da Tese de Church-Turing, inferimos, por um lado, de forma imediata, a existência de funções numéricas algorítmicas. Por outro lado, a partir de uma simples comparação entre a cardinalidade do conjunto das funções recursivas ($\aleph_0$) e a cardinalidade do conjunto das funções numéricas totais ($2^{\aleph_0}$), é possível provar que existem funções numéricas para as quais não há um procedimento mecânico que determine o seu respectivo valor a partir de seus argumentos. Ou seja, não existem algoritmos para computá-las. Por isso, elas são chamadas funções não-algorítmicas. Em outras palavras, podemos dizer que para tais funções não há capacidade computacional suficiente para solucioná-las. Sendo assim, descobrir os limites entre funções algorítmicas e não-algorítmicas é equivalente a descobrir o alcance e os limites do computador em geral. Nesse contexto, a Tese de Church-Turing representa um enorme ganho computacional, pois, de antemão, saberemos identificar as atividades que um computador poderá ou não desenvolver. Esta tese, no entanto, pode vir a ser, algum dia, refutada, possivelmente a partir de um contraexemplo. Mesmo que isto venha a ocorrer, a teoria das funções parciais recursivas e a teoria das funções parcialmente Turing-computáveis não perderão a sua importância, pois elas trazem consigo motivações suficientemente fortes para se consolidarem (o que, de fato, já aconteceu) como um campo vasto de estudo.

Como parte do *Resultado Fundamental*, demonstramos a equivalência entre as funções parciais recursivas e as funções parcialmente Turing-computáveis. Ao desenvolvermos a primeira parte desta demonstração, o que fizemos, nas entrelinhas, foi, na verdade, apresentar programas de Turing para computar as funções parciais recursivas. Estes programas, como

definimos, são constituídos por quíntuplas em vez de quádruplas. Do ponto de vista computacional, as computações realizadas de acordo com eles são, em geral, mais eficientes quando comparadas com aquelas realizadas de acordo com programas cujos elementos são quádruplas, pois alcança-se o *output* com um menor número de passos e, evidentemente, em um intervalo menor de tempo. Esse resultado nos introduz em um tópico muito discutido, atualmente, em Ciência da Computação, a análise da eficiência de algoritmos, que não só considera o tempo de execução de um algoritmo como também a sua capacidade de armazenar, efetuar e recuperar os passos de uma computação.

Vimos que a Teoria da Computabilidade busca, entre outras coisas, oferecer uma resposta matematicamente precisa para o problema da decisão de inúmeros conjuntos, classificando-os como decidíveis ou indecidíveis. De modo particular, concluímos, utilizando o Teorema de Rice, que embora o conjunto de índices das funções parciais recursivas seja decidível, os conjuntos de índices das funções recursivas e das funções recursivas primitivas não o são. Estabelecidas algumas equivalências, estendemos estes resultados sobre os conjuntos de índices para os conjuntos de funções por eles indexadas e concluímos que a recursividade de I_{FpR} garante a decidibilidade de FpR, enquanto que a não-recursividade de I_{FRP} e I_{FR} garante, respectivamente, a indecidibilidade de FRP e FR.

Diante deste resultado, surge-nos uma outra questão ainda mais geral: dado um conjunto de funções parciais recursivas C e o seu respectivo conjunto de índices I_C que propriedades de I_C “espelham” propriedades de C? Ou ainda: todas as propriedades que (não) predicam-se de I_C também (não) predicam-se de C? Uma resposta para estas questões seria imediata se houvesse entre os conjuntos I_C e C uma função biunívoca. Mas, como vimos, não estamos diante de uma codificação ortodoxa, porque cada elemento de C possui infinitos índices a ele associados. Isto posto, fica claro, de antemão, que, pelo menos, quanto à cardinalidade, ambos os conjuntos não coincidem. Mas, o que dizer quanto a outras propriedades? Poderíamos aprofundar essa questão a partir do estudo das hierarquias (aritmética e analítica) de conjuntos de funções numéricas, apresentadas em capítulos mais avançados da Teoria da Computabilidade, investigando quais as propriedades de I_C na hierarquia aritmética dos conjuntos numéricos "espelham" propriedades de C na hierarquia aritmética dos conjuntos de funções numéricas. Esta discussão, porém, demandaria o desenvolvimento de um outro trabalho a ser analisado por nós em uma próxima oportunidade.

REFERÊNCIAS

CHURCH, A. An Unsolvable Problem of Elementary Number Theory. *The American Journal of Mathematics*, vol. 58, 1936, p. 345-363.

_____. A Note on the Entscheidungsproblem. *The Journal of Symbolic Logic*, v. 1, n. 1. p. 40-41, mar., 1936. Disponível em: <https://www.fdi.ucm.es/profesor/fraguas/CC/church-A%20Note%20on%20the%20Entscheidungsproblem.pdf>. Acesso em: 10 de janeiro de 2016.

BOOLOS, G. S.; BURGESS, J. P.; JEFFREY, R. C. *Computabilidade e lógica***.** Trad. Cezar A. Mortari. São Paulo: Editora UNESP, 2012

COOPER, S. B. *Computability Theory*. New York: Chapman and Hall/CRC, 2004.

DAVIS, M. *Computability and Unsolvability*. Dover: New York, 1982.

_____. *The Undecidable:* Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions. Dover: New York, 1993.

DIAS, M. F.; FILHO, R. N. A. P. Teoria de la recursión y lógica. *Revista de Filosofia*. Buenos Aires: Asociación de Estudios Filosóficos, 1987.

DIAS, M. F.; WEBER, L. *Teoria da recursão*. São Paulo: Ed. UNESP, 2010.

EPSTEIN, R. C.; CARNIELLI, W. A. *Computabilidade, funções computáveis, lógica e os fundamentos da Matemática.* 2. ed. São Paulo: Editora UNESP, 2009

FONSECA FILHO, C. *História da Computação*: O caminho do pensamento e da tecnologia. Porto Alegre: EDIPUCRS, 2007.

KALMÁR, L. An argument against the plausibility of Church's Thesis. *Constructivity in mathematics*. Heyting: Amsterdam, North HOLLAND, 1959.

KLEENE, S. C. *Introduction to Metamathematics*. Ishi Press: New York and Tokyo, 2009.

MENDELSON, E. *Introduction to Mathematical Logic*, 5. Ed. New York: Chapman and Hall/CRC, 2009.

MONK, J. D. *Mathematical Logic*. New York, Heidelberg, Berlin: Springer-Verlag, 1976.

ODIFREDDI, P. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*, I. North-Holland: Amsterdam, New York, 1989.

ROGERS, H. *Theory of Recursive Functions and Effective Computability*. Cambridge, Massachusetts, London: MIT Press, 1987.

SOARE, R. *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets*. Berlin, Heidelberg, New York: SpringerVerlag, 1987.

SOBRINHO, J.Z. Aspectos da Tese de Church-Turing. *Matemática Universitária*, nº 6, p. 1-23, dez., 1987. Disponível em: http://matematicauniversitaria.ime.usp.br/Conteudo/n06/n06_Artigo01.pdf. Acesso em: 12 de dezembro de 2015.

TURING, A. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society,* v. 42, p. 230-265, 1936. Disponível em: http://plms.oxfordjournals.org/content/s2-42/1/230.full.pdf+html?ijkey=bvNIrAXLJ7n4ODP&keytype=ref. Acesso em: 10 de janeiro de 2016.