UNIVERSIDADE FEDERAL DA PARAÍBA

PROGRAMA DE PÓS-GRADUAÇÃO EM FILOSOFIA

VICTOR PEREIRA GOMES

FUNÇÕES RECURSIVAS PRIMITIVAS: caracterização e alguns resultados para esta classe de funções

João Pessoa, PB

2016

VICTOR PEREIRA GOMES

FUNÇÕES RECURSIVAS PRIMITIVAS: caracterização e alguns resultados para esta classe de funções

Dissertação apresentada ao Programa de Pós Graduação em Filosofia da Universidade Federal da Paraíba, para obtenção do título de Mestre em Filosofia.

Orientador: Prof. Dr. Matias Francisco Dias

João Pessoa, PB

2016

G633f Gomes, Victor Pereira.
Funções recursivas primitivas: caracterização e alguns resultados para esta classe de funções / Victor Pereira Gomes.- João Pessoa, 2016.
55f.
Orientador: Matias Francisco Dias
Dissertação (Mestrado) - UFPB/CCHL
1. Filosofia. 2. Funções algorítmicas. 3. Funções recursivas primitivas. 4. Algoritmo reconhecedor. 5. Função universal.

UFPB/BC CDU: 1(043)

VICTOR PEREIRA GOMES

FUNÇÕES RECURSIVAS PRIMITIVAS: caracterização e alguns resultados para esta classe de funções

Dissertação apresentada ao Programa de Pós Graduação em Filosofia da Universidade Federal da Paraíba, para obtenção do título de Mestre em Filosofia.

Aprovada em 21 / 06 / 16

BANCA EXAMINADORA

Matias Francisco Dias

Prof. Dr. Matias Francisco Dias – Orientador
Universidade Federal da Paraíba

Ana Leda de Araújo

Profª. Drª. Ana Leda de Araújo
Universidade Federal da Paraíba

Garibaldi

Prof. Dr. Garibaldi Monteiro Sarmento
Universidade Federal da Paraíba

Prof. Dr. Américo Augusto Nogueira Vieira
Universidade Federal do Paraná

Dedico este trabalho àqueles que de forma direta ou indireta contribuíram para a realização do mesmo.

Existem muitas hipóteses em ciência que estão erradas. Isso é perfeitamente aceitável, elas são a abertura para achar as que estão certas.

Carl Sagan

AGRADECIMENTOS

Agradeço à minha família, pelo incentivo, paciência e carinho.

Ao meu orientador e amigo Prof. Dr. Matias Francisco Dias, pelos ensinamentos proporcionados ao longo desta caminhada, por acreditar em mim e no futuro deste projeto, e também por ser exemplo de profissional e pessoa, o qual fará sempre parte da minha vida.

À Profª. Drª. Ana Leda de Araújo, pela participação fundamental na realização desta dissertação, por sua contribuição para o meu crescimento profissional e por ser também um exemplo a ser seguido.

Ao coordenador do programa, Prof. Dr. Antônio Rufino Vieira, pela paciência e pelo suporte dado a mim e aos demais alunos do programa.

Ao colega de mestrado e amigo Gustavo Cavalcanti de Melo, pelos dias de estudo, pelo ombro amigo nos momentos de desespero e por sua contribuição direta nesta dissertação.

À amiga Paula Fernanda Patrício de Amorim, pela paciência, incentivo e também pela sua contribuição nesta dissertação.

Ao meu primo Leandro Ribeiro Gomes Andrade, pelo apoio, incentivo e suporte em João Pessoa.

A todos aqueles que me incentivaram e contribuíram de forma direta ou indireta para a realização desta dissertação.

RESUMO

A classe das funções recursivas primitivas não constitui uma versão formal para a classe das funções algorítmicas, estudamos esta classe especial de funções numéricas devido ao fato de que muitas das funções conhecidas como algorítmicas são recursivas primitivas. A abordagem acerca da classe das funções recursivas primitivas tem como objetivo explorar esta classe especial de funções e, a partir disto, apresentar soluções para os seguintes problemas: (1) dada a classe das derivações recursivas primitivas, há um algoritmo, ou seja, um procedimento mecânico, para reconhecer derivações recursivas primitivas? (2) Existe uma função universal para a classe das funções recursivas primitivas? Se sim, essa função é recursiva primitiva? (3) Toda função algorítmica é recursiva primitiva? Para apresentar soluções para estas questões, nos pautamos no método hipotético-dedutivo e argumentamos com base nos manuais de Davis (1982), Mendelson (2009), Dias e Weber (2010), Rogers (1987), Soare (1987), Cooper (2004), entre outros. Apresentamos a teoria das máquinas de Turing, que constitui uma versão formal para a noção intuitiva de algoritmo, e, em seguida, a famosa tese de Church-Turing, a qual identifica a classe das funções algorítmicas com a classe das funções Turing-computáveis. Exibimos a classe das funções recursivas primitivas, e mostramos que a mesma constitui uma subclasse das funções Turing-computáveis. Tendo explorado a classe das funções recursivas primitivas, como resultados, provamos que existe um algoritmo reconhecedor para a classe das derivações recursivas primitivas; que existe uma função universal para a classe das funções recursivas primitivas a qual não pertence a esta classe; e que nem toda função algorítmica é recursiva primitiva.

Palavras-chave: funções algorítmicas, funções recursivas primitivas, algoritmo reconhecedor, função universal.

ABSTRACT

The class of primitive recursive functions is not a formal version to the class of algorithmic functions, we study this special class of numerical functions due to the fact of that many of the functions known as algorithmic are primitive recursive. The approach on the class of primitive recursive functions aims to explore this special class of functions and from that, present solutions for the following problems: (1) given the class of primitive recursive derivations, is there an algorithm, that is, a mechanical procedure for recognizing primitive recursive derivations? (2) Is there a universal function for the class of primitive recursive functions? If so, is this function primitive recursive? (3) Are all the algorithmic functions primitive recursive? To provide solutions to these issues, we base on the hypothetical-deductive method and argue based on the works of Davis (1982), Mendelson (2009), Dias e Weber (2010), Rogers (1987), Soare (1987), Cooper (2004), among others. We present the theory of Turing machines which is a formal version to the intuitive notion of algorithm, and after that the famous Church-Turing tesis which identifies the class of algorithmic functions with the class of Turing-computable functions. We display the class of primitive recursive functions and show that it is a subclass of Turing-computable functions. Having explored the class of primitive recursive functions we proved as results that there is a recognizer algorithm to the class of primitive recursive derivations; that there is a universal function to the class of primitive recursive functions which does not belong to this class; and that not every algorithmic function is primitive recursive.

Keywords: algorithmic functions, primitive recursive functions, recognizer algorithm, universal function.

SUMÁRIO

INTRODUÇÃO

A Teoria da Computabilidade, Teoria da Recursão, ou Teoria das Funções Recursivas (como também é conhecida), constitui uma subárea da lógica matemática e também da teoria da computação. Um dos objetivos dessa teoria é classificar certos problemas como *solúveis* ou *insolúveis*. Entretanto, para que fosse possível classificar esses problemas nessas duas categorias, lógicos e matemáticos como Turing, Post, Church, Gödel e Kleene, entre outros, tiveram num primeiro momento, a tarefa de formalizar a noção intuitiva de algoritmo. Nesse sentido, a Teoria da Computabilidade é importante para a epistemologia e, portanto, para a filosofia, uma vez que em epistemologia, tal como em outras ciências, geralmente parte-se de um conceito intuitivo para precisá-lo posteriormente. Dessa forma, acredita-se que a noção intuitiva de *algoritmo* foi precisada de uma maneira formal através dos estudos da Teoria da Computabilidade. Os argumentos a favor da tese de Church-Turing, que é a tese segundo a qual toda função algorítmica é Turing-computável, e vice-versa, reforçam a ideia de que a Teoria da Computabilidade precisou definitivamente a noção intuitiva de algoritmo.

A palavra algoritmo teve sua origem devido ao trabalho do matemático e astrônomo persa do século IX, Mohammed Ibn Musa al'Khwarizmi, cujo livro sobre o uso dos numerais hindu-arábicos descreve procedimentos para operações aritméticas usando esses numerais. Autores europeus usaram uma adaptação latina de seu nome, até finalmente chegarem na palavra algoritmo para descrever a área da aritmética com numerais hindu-arábicos. (FONSECA FILHO, 2007).

Independentemente de sua etimologia, a noção intuitiva de algoritmo, a saber, um conjunto finito (não vazio) de instruções precisas para computar uma função numérica (função cujo domínio e codomínio são conjuntos de números naturais), possui algumas características que passamos a expor a seguir:

1. Existe um agente computacional A que aplica um procedimento P (um conjunto de instruções) a um *input* simbólico I (numa notação padrão), e, de acordo com P, desenvolve uma computação C (uma sequência finita de passos $p_1,...,p_n$), obtendo ao final um *output* simbólico O (numa notação padrão).
2. Chamamos o procedimento P de algoritmo (um número finito de instruções precisas, não ambíguas, escrito em linguagem simbólica ou natural).

3. O agente computacional não precisa ser necessariamente humano, pois devido à precisão de P, nenhuma criatividade é exigida de sua parte para realizar uma computação, que é completamente mecânica, podendo ser realizada também por uma máquina, contanto que ela apresente algumas capacidades mínimas como efetuar, armazenar e recuperar os passos $p_1,...,p_n$ de uma computação C.
4. A computação C desenvolvida a partir da aplicação das instruções de P a um dado *input* I deve ser executada da seguinte forma: (1) de maneira discreta, ou seja, passo a passo, e (2) de maneira determinística, ou seja, não pode ser regida por recursos aleatórios externos ao algoritmo.
5. Levando em consideração o que é computável em *princípio* e não o que é computável na *prática*, podemos afirmar que não existe um limite finito para: (1) o tamanho dos *inputs* simbólicos, (2) o tamanho do conjunto P de instruções, (3) a capacidade de efetuar, armazenar e recuperar os passos $p_1,...,p_n$ de uma computação C, e (4) o tamanho de uma computação C. Entretanto, a única coisa que se exige é que tenham um limite. Por exemplo: um algoritmo pode ter 1 trilhão de instruções, um *input* pode ter comprimento 1 milhão, mas embora sejam compostos de números demasiado grandes, esses números são finitos.

A noção intuitiva de algoritmo teve várias tentativas de formalização nos anos 30, século XX, como as noções de funções recursivas, funções Turing-computáveis, funções λ-definíveis e outras mais. Um fato importante a ser observado é que essas várias noções produzem uma mesma classe [1] de funções, a saber, funções algorítmicas, funções cujos valores podem ser calculados mecanicamente num número finito de passos, o que é em parte argumento para a tese de Church-Turing. Ressalta-se que os trabalhos de Turing, Post, Church, Gödel e Kleene, entre outros, foram decisivos para que se tivesse formalizado – assim se acredita – de uma vez por todas a noção intuitiva de algoritmo. A partir desses fatos desenvolveu-se intensamente o que nós conhecemos hoje como Teoria da Computabilidade.

[1] Ao longo desta dissertação, sempre que o termo classe for mencionado, estaremos nos referindo a conjunto.

Uma parte dessa teoria procura responder a problemas do seguinte tipo: dado um conjunto A e um elemento x quaisquer, há um algoritmo que nos permita sempre decidir se $x \in$ A ou $x \notin$ A? Esse problema é chamado de *problema de decisão* para A. Se a resposta for positiva, dizemos que A é *decidível*; caso contrário, dizemos que A é *indecidível*.

Considerando a classe das funções recursivas primitivas (que será exibida), busca-se, em nossa dissertação ponderar, avançar ou mesmo resolver as seguintes questões:

1. Dada a classe das derivações recursivas primitivas, isto é, a classe de todos os conjuntos de instruções para computar funções recursivas primitivas, há um algoritmo reconhecedor para ela? Note-se que: por um reconhecedor de uma classe de algoritmos, entendemos um algoritmo que decide se um dado conjunto de instruções pertence ou não à classe. Nesse caso, a existência de um reconhecedor de uma classe de algoritmos seria a solução para o problema de decisão dessa classe.
2. É possível demonstrar a existência, ou a não existência, de uma função universal para a classe das funções recursivas primitivas? Em caso afirmativo, seria essa função recursiva primitiva?
3. Toda função algorítmica é passível de ser classificada como recursiva primitiva?

Para que possamos apresentar respostas (ou pelo menos avanços) para estas questões, precisamos traçar alguns objetivos específicos (ou objetivos intermediários). Dessa forma, para lograrmos êxito em nossa empreitada, nossa abordagem será realizada em três capítulos.

No primeiro capítulo apresentaremos uma versão formal para as noções intuitivas de algoritmo e função algorítmica, a saber, a teoria das máquinas de Turing. Escolhemos a versão de Turing por se tratar da versão mais adotada nos manuais (tomada aqui como versão *standard*) e também por ser a versão que mais se assemelha ao modelo de computador digital que conhecemos.

No segundo capítulo exibiremos a classe das funções recursivas primitivas, e mostraremos que ela constitui uma subclasse das funções Turing-computáveis.

Feito isto, no terceiro e último capítulo apresentaremos os resultados obtidos para as questões anteriormente mencionadas como o problema de dissertação. Embora essas questões já tenham sido por demais trabalhadas, a nossa abordagem será constituída de uma maneira diferente da encontrada nos manuais pesquisados na literatura.

Nós nos pautaremos no Método Hipotético-Dedutivo, e argumentaremos baseados em ampla base bibliográfica, principalmente com base em Martin Davis, Elliott Mendelson, Matias Francisco Dias, Hartley Rogers, Robert Soare e Barry Cooper.

1. MÁQUINAS E PROGRAMAS DE TURING: UMA TENTATIVA DE FORMALIZAR A NOÇÃO INTUITIVA DE ALGORITMO

Neste capítulo, temos como propósito apresentar uma versão formal para a noção intuitiva de algoritmo. Exibiremos aqui a teoria das *máquinas e programas de Turing*, a sua aritmetização, e também a tese de Church-Turing, que reforça a ideia de que a noção intuitiva de algoritmo foi (de uma vez por todas) formalizada.

1.1. CARACTERIZANDO MÁQUINAS E PROGRAMAS DE TURING

Podemos definir uma máquina de Turing como um mecanismo teórico constituído de um cabeçote de leitura/escrita e de uma fita infinita em ambas as direções (tanto à esquerda quanto à direita), dividida em quadrados de tamanhos iguais (em sentido horizontal), que são observados individualmente pelo cabeçote. Em cada quadrado, somente um dos elementos do conjunto alfabeto A = {s_0, s_1, s_2, s_3, ...} chamados de *símbolos da fita* pode estar escrito. Quando o cabeçote de leitura/escrita examina um quadrado, a máquina só pode estar em um dos elementos do conjunto de estados internos Q = {q_0, q_1, q_2, q_3, ...} chamados de *estados internos*. Contudo, como a máquina examina apenas um quadrado por vez, ela só pode executar um dos seguintes procedimentos:

1. Apagar o símbolo da fita que está sendo examinado, e escrever outro símbolo da fita em seu lugar;
2. Mover-se para o quadrado adjacente à direita;
3. Mover-se para o quadrado adjacente à esquerda.

Utilizaremos o símbolo R (de *right*) para indicar que a máquina passa a observar o quadrado imediatamente à direita do qual estava sendo observado anteriormente, e L (de *left*) para indicar que a máquina passa a observar o quadrado imediatamente à esquerda do qual estava sendo observado anteriormente. Esses elementos do conjunto de movimento M = {R, L}, são chamados de *símbolos de movimento*. Com base nessa descrição, para que uma máquina de Turing possa executar tais procedimentos, ela deve ser guiada por um conjunto finito (não vazio) de

instruções, chamado de *programa de Turing*. No que se segue, caracterizaremos o que é um programa de Turing.

Definição 1: A linguagem utilizada para criar programas para uma máquina de Turing é composta pelos elementos dos conjuntos A, Q e M, que serão chamados de *símbolos de programas de Turing*.

Definição 2: Uma *expressão* da linguagem de programas é uma sequência finita de *símbolos de programas de Turing*.

Definição 3: Uma *quádrupla* é uma expressão de um dos seguintes tipos:

1. $q_i s_j s_k q_l$
2. $q_i s_j R q_l$
3. $q_i s_j L q_l$

Observação: De acordo com a primeira quádrupla, quando a máquina está no estado interno q_i examinando o quadrado com o símbolo s_j, ela apaga s_j, escreve s_k e assume o estado interno q_l. No que concerne à segunda quádrupla, quando a máquina está no estado interno q_i examinando o quadrado com o símbolo s_j, ela move-se para o quadrado imediatamente à direita e assume o estado interno q_l. Por fim, de acordo com a terceira quádrupla, quando a máquina está no estado inicial q_i examinando o quadrado com o símbolo s_j, ela move-se para o quadrado imediatamente à esquerda e assume o estado interno q_l.

Definição 4: Dizemos que duas quádruplas são *inconsistentes*, se ambas têm os dois primeiros símbolos iguais, e pelo menos um dos dois símbolos restantes diferentes; caso contrário dizemos que são *consistentes*.

Definição 5: Um *programa de Turing* é um conjunto finito (não vazio) de quádruplas consistentes. Essa restrição é adotada para evitar comandos contraditórios, e é chamada de *restrição de consistência*.

Observação: Se fosse o caso de termos num programa de Turing as quádruplas $q_0S_1Lq_1$ e $q_0S_1S_0q_2$, ambas seriam inconsistentes e portanto, a máquina não saberia como agir diante de tais comandos. Contudo, se levarmos em consideração máquinas de Turing não-determinísticas – as quais não entraremos em detalhes aqui –, a máquina executaria as duas possibilidades simultaneamente.

Definição 6: Uma *descrição instantânea* é uma expressão da linguagem de programas de Turing que contém um único q_i, não contém nenhum dos símbolos de movimento, e a única restrição é que q_i não seja o último símbolo à direita.

Exemplo: A expressão $q_0S_0S_1$ é uma descrição instantânea, ao passo que a expressão $S_0S_1Rq_2$ não é uma descrição instantânea.

Definição 7: Sendo t_1 e t_2 sequências finitas de símbolos da fita, dado um programa de Turing W e dadas descrições instantâneas α e β, escrevemos α → β (W) ["α acarreta β via W"] se uma das seguintes condições vale:

1. $q_iS_jS_kq_l \in W$, $\alpha = t_1q_iS_jt_2$, $\beta = t_1q_lS_kt_2$;
2. $q_iS_jRq_l \in W$, $\alpha = t_1q_iS_jS_kt_2$, $\beta = t_1S_jq_lS_kt_2$, ou $\alpha = t_1q_iS_j$, $\beta = t_1S_jq_lS_0$;
3. $q_iS_jLq_l \in W$, $\alpha = t_1S_kq_iS_jt_2$, $\beta = t_1q_lS_kS_jt_2$, ou $\alpha = q_iS_jt_2$, $\beta = q_lS_0S_jt_2$.

Observação: Perceba que na primeira condição, dada uma quádrupla do programa W e uma descrição instantânea α, quando a máquina encontra-se no estado interno q_i observando S_j, ela reage apagando S_j, escrevendo S_k em seu lugar e assumindo o estado interno q_l, resultando em β. A segunda e terceira condições são análogas à primeira, porém, na segunda condição a máquina reage movimentando-se para o quadrado adjacente à direita, e na terceira condição a máquina reage movimentando-se para o quadrado adjacente à esquerda. É importante salientar que de acordo com a definição 6, dada uma descrição instantânea, o símbolo de estado interno não pode ser o último símbolo à direita. Por esta razão acrescentamos o símbolo S_0 em β na segunda condição. Já no que diz respeito à terceira condição, acrescentamos o símbolo S_0 em β, pois quando a máquina move-se para a esquerda e assume o estado interno q_l, o quadrado examinado por q_l é vazio.

Definição 8: Dado um programa de Turing W, dizemos que α é uma *descrição instantânea inicial* se $\alpha = q_0 s_1 t_1$.

Definição 9: Dado um programa de Turing W, dizemos que α é uma *descrição instantânea terminal* se $\alpha = t_1 q_i s_j t_2$ e W não possui quádruplas começando com $q_i s_j$. Nesse caso, a máquina para.

Nota: Se fosse o caso de W possuir uma quádrupla começando com $q_i s_j$, α não seria uma descrição instantânea terminal, pois de acordo com tal quádrupla a máquina continuaria trabalhando.

Definição 10: Uma *computação de acordo com um programa de Turing* W é uma sequência finita $\alpha_1,...,\alpha_m$ de descrições instantâneas tais que α_1 é inicial, α_m é terminal e $\alpha_i \rightarrow \alpha_{i+1}$ (W), para todo i tal que $1 \leq i < m$. Nesse caso, escrevemos $Res_W(\alpha_1) = \alpha_m$, e dizemos que α_m é a resultante de α_1 com respeito a W.

Exemplo: Considere o programa de Turing $W = \{q_0 s_1 s_0 q_2,\ q_2 s_0 R q_0\}$ e a seguinte sequência finita de descrições instantâneas $\alpha_1,...,\alpha_9$:

α_1. $q_0 s_1 s_1 s_1 s_1$ (inicial)
α_2. $q_2 s_0 s_1 s_1 s_1$
α_3. $s_0 q_0 s_1 s_1 s_1$
α_4. $s_0 q_2 s_0 s_1 s_1$
α_5. $s_0 s_0 q_0 s_1 s_1$
α_6. $s_0 s_0 q_2 s_0 s_1$
α_7. $s_0 s_0 s_0 q_0 s_1$
α_8. $s_0 s_0 s_0 q_2 s_0$
α_9. $s_0 s_0 s_0 s_0 q_0 s_0$ (terminal)

Essa sequência finita de descrições instantâneas é uma computação obtida através do programa de Turing W, e portanto, dizemos que $Res_W(\alpha_1) = \alpha_9$, ou seja, $Res_W(q_0 s_1 s_1 s_1 s_1) = s_0 s_0 s_0 s_0 q_0 s_0$.

Para que uma máquina de Turing possa executar computações numéricas, faz-se necessário introduzirmos uma representação simbólica para denotar números naturais, como veremos a seguir:

1. Ao símbolo s_0 associaremos o símbolo B (de *blank*) para indicar um quadrado vazio durante a computação.
2. Ao símbolo s_1 associaremos o símbolo |.
3. A respeito da descrição instantânea inicial de uma computação, para que possamos representar um número natural n, escreveremos | em $n+1$ quadrados adjacentes. Desse modo, nós representamos o número natural 3, por exemplo, como segue: ||||.
4. Já no que diz respeito à descrição instantânea terminal de uma computação, um número natural n será representado por n traços verticais não necessariamente adjacentes.
5. Representaremos $n+1$ traços verticais (|) por $\bar{n}$.
6. Para representarmos uma n-upla $k_1, \dots, k_n$, escreveremos $\bar{k}_1\mathrm{B}...\mathrm{B}\bar{k}_n$.

Observação: Nós iniciaremos toda computação com uma descrição instantânea inicial da forma $q_0|t_1$.

Definição 11: Seja W um programa de Turing. Então, para cada $n \geq 1$, está associada a W uma única função n-ária $\psi_w^n(x_1, \dots, x_n)$, da seguinte maneira: para cada n-upla $(k_1, \dots, k_n)$, nós fazemos $\alpha_1 = q_0\bar{k}_1\mathrm{B}...\mathrm{B}\bar{k}_n$ e distinguimos dois casos:

1. Há uma computação $\alpha_1,...,\alpha_m$ de acordo com W, e, neste caso, nós colocamos $\psi_w^n(k_1, \dots, k_n) = (\alpha_m) = (\mathrm{Res}_W(\alpha_1))$, em que (α_m) representa o número de traços verticais na descrição instantânea terminal α_m;
2. Não há uma computação $\alpha_1,...,\alpha_m$ de W, e, neste caso, $\psi_w^n(k_1, \dots, k_n)$ fica indefinida.

Definição 12: Uma função numérica n-ária $f(x_1, \dots, x_n)$ diz-se *parcialmente Turing-computável*, se existe um programa de Turing W tal que $f(x_1, \dots, x_n) = \psi_w^n(x_1, \dots, x_n)$.

Entretanto, se $f(x_1, \ldots, x_n)$ é total, isto é, se está definida para todas as n-uplas dos naturais, diz-se que ela é *Turing-computável*.

Observação: Em outras palavras, dizemos que uma função é (parcialmente) Turing-computável, se existe uma máquina de Turing que a computa.

A distinção que apresentamos entre máquinas e programas de Turing é bastante intuitiva. De um ponto de vista formal, falaríamos apenas em máquinas de Turing, já que máquinas e programas denotam a mesma coisa.

1.2. ARITMETIZAÇÃO DAS MÁQUINAS OU PROGRAMAS DE TURING

Também conhecida como numeração de Gödel, a aritmetização foi inicialmente utilizada para traduzir os enunciados metalinguísticos da aritmética elementar de primeira ordem de Peano [2] na linguagem-objeto da aritmética, cujo domínio é constituído por números naturais. De forma análoga à aritmética de Peano, a aritmetização também pode ser utilizada para qualquer linguagem formal, atribuindo números aos seus componentes básicos. Com base nisso, aritmetizaremos a seguir a teoria das máquinas ou programas de Turing codificando os símbolos, expressões e sequências de expressões da linguagem de programas de Turing, fazendo uso dos números naturais. Inicialmente, para cada símbolo de programas de Turing, atribuiremos uma função $g(x)$ – que denota o número de Gödel de x – para codificá-los, como veremos a seguir:

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

[2] O leitor encontrará uma apresentação detalhada da aritmética elementar de primeira ordem de Peano na obra de MENDELSON, E. *Introduction to Mathematical Logic*, 5. ed. New York: Chapman and Hall/CRC, 2009, p.149.

Observação: Note-se que todo número natural ímpar $n > 1$, é código de um único símbolo de programas de Turing.

Agora, atribuiremos para cada expressão E da linguagem de programas de Turing, um número de Gödel denotado por g'(E). Se E for uma expressão vazia, atribuiremos para ela o número 1 como seu código. Se E é uma expressão de comprimento $\alpha_1,...,\alpha_n$, onde cada α_i é um símbolo de programa de Turing, nós a codificaremos da seguinte forma:

$$g'(E) = \prod_{i<n} p_i^{g(\alpha_{i+1})}$$

Observação: p_i denota o i-ésimo número primo e p_0 = 2.

Exemplo: Se E é a expressão $\mathrm{q_0 s_1 s_0 q_1}$, então o número de Gödel para ela é o seguinte:

$$g'(E) = p_0^{g(q_0)} \cdot p_1^{g(s_1)} \cdot p_2^{g(s_0)} \cdot p_3^{g(q_1)} = 2^9 \cdot 3^{11} \cdot 5^7 \cdot 7^{13}$$

Atribuiremos para cada sequência finita de expressões da linguagem de programas de Turing $\mathrm{S} = \mathrm{E_1},...,\mathrm{E_n}$, um número de Gödel, denotado por g''(S), da seguinte forma:

$$g''(S) = \prod_{i<n} p_i^{g'(E_{i+1})}$$

Exemplo: Considere a seguinte sequência finita de expressões da linguagem de programas de Turing:

$\mathrm{E_1} = \mathrm{q_0 s_1 s_1 s_1}$

$\mathrm{E_2} = \mathrm{q_1 s_1 s_1 s_0}$

$\mathrm{E_3} = \mathrm{s_0 q_2 s_1 s_1}$

O número de Gödel para essa sequência finita de expressões da linguagem de programas de Turing é o seguinte:

$$g''(S) = p_0^{g'(E_1)} \cdot p_1^{g'(E_2)} \cdot p_2^{g'(E_3)} = 2^{2^9.3^{11}.5^{11}.7^{11}} \cdot 3^{2^{13}.3^{11}.5^{11}.7^7} \cdot 5^{2^7.3^{17}.5^{11}.7^{11}}$$

Observação: Dado um número natural $n \geq 1$, se n for um número de Gödel, nós podemos encontrar o símbolo, a expressão ou a sequência de expressões da linguagem de programas de Turing da qual n é código, da seguinte forma:

1. Se n é ímpar maior que 1, n é código de um único símbolo da linguagem de programas de Turing.
2. Se $n = 1$, n é código de uma expressão vazia.
3. Se $n = 1$ ou n é par e sua decomposição no seguimento inicial dos números primos possui expoentes ímpares maiores que 1, então n é o número de Gödel de uma única expressão da linguagem de programas de Turing.
4. Se n é par, quando fatoramos n no seguimento inicial dos números primos, n será código de uma única sequência de expressões da linguagem de programas de Turing, se e somente se:

   I. Os expoentes de n forem iguais a 1.
   II. Os expoentes de n forem iguais a 1 e a números pares, e a decomposição desses expoentes pares em fatores primos resultarem em expoentes ímpares maiores que 1.
   III. Os expoentes de n forem pares, e a decomposição desses expoentes pares em fatores primos resultarem em expoentes ímpares maiores que 1.

Vimos até aqui como codificar símbolos, expressões e sequências de expressões da linguagem de programas de Turing. Entretanto, se W é um programa de Turing, então W é um conjunto (não vazio) de quádruplas e não uma sequência finita de quádruplas. Dessa forma, se W possui um número n de quádruplas, então W possui $n!$ números de Gödel.

1.3. TESE DE CHURCH-TURING

Como vimos, há uma classe formal de funções numéricas que é constituída de todas as funções algorítmicas, a saber, a classe das funções Turing-computáveis. A identificação dessas duas classes (classe das funções algorítmicas e classe das funções Turing-computáveis) é apresentada na famosa tese de Church-Turing. Nesta seção, apresentaremos a tese de Church-Turing em duas versões: estrita e ampla. Contudo, vale ressaltar que estamos tratando de uma tese e não de um teorema matemático, já que a mesma não é demonstrável em termos matemáticos, apenas mostra que uma versão informal corresponde a uma versão formal.

1.3.1. Tese de Church-Turing (versão estrita)

Toda função algorítmica é Turing-computável, e vice-versa.

1.3.2. Tese de Church-Turing (versão ampla)

Toda função parcialmente algorítmica é parcialmente Turing-computável, e vice-versa.

Observação: Embora esta tese não possa ser comprovada matematicamente, existem fortes argumentos para sustentá-la. Citaremos aqui dois deles:

1. Até hoje, não se conseguiu encontrar uma função parcial algorítmica ou função algorítmica que não seja parcialmente Turing-computável ou Turing-computável, respectivamente.
2. Até hoje, todas as tentativas de caracterizar formalmente as noções intuitivas de função parcial algorítmica e função algorítmica produziram exatamente as mesmas classes de funções, a saber, a classe das funções parcialmente Turing-computáveis e a classe das funções Turing-computáveis, respectivamente.

Nos capítulos que seguem, utilizaremos a tese de Church-Turing como método informal de prova para demonstrar alguns resultados.

2. FUNÇÕES RECURSIVAS PRIMITIVAS: UMA SUBCLASSE DAS FUNÇÕES TURING-COMPUTÁVEIS

Apresentaremos neste capítulo a classe das *funções recursivas primitivas* juntamente com a sua aritmetização, e mostraremos que ela constitui uma subclasse das funções Turing-computáveis.

2.1. FUNÇÕES RECURSIVAS PRIMITIVAS

No que se segue, apresentaremos algumas noções básicas concernentes à classe das funções recursivas primitivas, que vem a ser de grande importância em lógica matemática e ciência da computação. O motivo pelo qual escolhemos apresentar esta classe especial de funções numéricas deve-se ao fato de que muitas das funções conhecidas como algorítmicas (soma, multiplicação, exponenciação, entre outras), são recursivas primitivas.

Definição 1: As seguintes funções são chamadas *funções iniciais*.

1. A função nulo, $\mathrm{N}(x) = 0$ para todo x.
2. A função sucessor, $S(x) = x + 1$ para todo x.
3. A função projeção, $U_i^n(x_1, \dots, x_n) = x_i$, $1 \leq i \leq n$, para todo $x_1, \dots, x_n$.

Definição 2: As seguintes operações são chamadas operações básicas entre funções.

1. A função f (de n variáveis, $n \geq 1$) é obtida por *composição*, da função h (de m variáveis, $m \geq 1$) e das funções g_1,..., g_m (todas de n variáveis) se, e somente se:

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

2. A função f (de $n+1$ variáveis, $n \geq 0$) é obtida por *recursão primitiva*, da função g (de n variáveis) e da função h (de n+2 variáveis) se, e somente se:

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$$
$$f\big(x_1, \dots, x_n, S(y)\big) = h\big(x_1, \dots, x_n, y, f(x_1, \dots x_n, y)\big)$$

Se considerarmos $n = 0$, a função f será obtida por *recursão primitiva* apenas pela função h como segue:

$$f(0) = k$$
$$f\big(S(y)\big) = h\big(y, f(y)\big)$$

Observação: k é um número natural fixo, ou seja, uma constante.

Definição 3: Uma função f é recursiva primitiva, se e somente se existe uma sequência finita de funções $f_1, \dots, f_n$ tal que $f_n = f$ e, para $1 \leq i \leq n$, ou f_i é uma função inicial, ou f_i vem de funções anteriores da sequência por aplicação de uma das operações básicas.

Observação: Note-se que dada uma sequência finita de funções para f, podemos reconhecer se tal sequência é recursiva primitiva especificando como cada função da sequência é obtida. Dessa forma, quando especificamos cada função da sequência, obtemos uma derivação recursiva primitiva para f. Como consequência, dizemos também que uma função f é recursiva primitiva se e somente se existe uma derivação recursiva primitiva para ela.

Para melhor compreender o que acabamos de dizer, apresentaremos a seguir um exemplo de derivação recursiva primitiva para a função soma $x + y$.

Definição intuitiva:

$x + 0 = x$
$x + (y + 1) = (x + y) + 1$

Derivação:

1. $S(x) = x + 1$ função inicial
2. $U_3^3(x, y, z) = z$ função inicial
3. $g_0(x, y, z) = S\big(U_3^3(x, y, z)\big)$ 1,2 / composição
4. $U_1^1(x) = x$ função inicial
5. $g_1(x, 0) = U_1^1(x)$
   $g_1\big(x, S(y)\big) = g_0\big(x, y, g_1(x, y)\big)$ 4,3 / recursão primitiva

Uma derivação recursiva primitiva para uma função f nos fornece um conjunto finito de instruções para computá-la efetivamente. Dito isto, nós obtemos a computação a partir das funções mais interiores para as mais exteriores, no sentido das funções à esquerda para as funções à direita.

Exemplo: Seguindo as instruções fornecidas pela derivação recursiva primitiva para a função $g_1(x, y) = x + y$ apresentada acima, executaremos a seguir uma computação para $g_1(2, 1)$.

$$
\begin{aligned}
g_1(2, 1) &= g_0\big(2, 0, g_1(2, 0)\big) \\
&= g_0\big(2, 0, U_1^1(2)\big) \\
&= g_0(2, 0, 2) \\
&= S\big(U_3^3(2, 0, 2)\big) \\
&= S(2) \\
&= 3
\end{aligned}
$$

Apresentaremos a seguir, como teorema, algumas funções algorítmicas que são recursivas primitivas. Entretanto, faz-se necessário, de antemão, apresentarmos um lema que será primordial para que possamos expor estas funções.

Lema 1: Para qualquer número natural n, a função constante $C_n(x) = n$ é recursiva primitiva.

Prova:

A prova é feita por indução em n. Primeiro provamos pra base: $C_0(x) = N(x)$. A seguir, supomos então que $C_n(x)$ é recursiva primitiva. Dessa forma, $C_{n+1}(x) = S\big(C_n(x)\big)$ é recursiva primitiva, por composição.

Teorema 1: As seguintes funções são recursivas primitivas:

a) **Multiplicação**:

Definição intuitiva:

$x.0 = 0$
$x\,.\,(y+1) = (x\,.\,y) + x$

Derivação:

1. $N(x) = 0$ função inicial
2. $U_1^3(x,y,z) = x$ função inicial
3. $U_3^3(x,y,z) = z$ função inicial
4. $g_1(x,0) = U_1^1(x)$
   $g_1\big(x,S(y)\big) = g_0\big(x,y,g_1(x,y)\big)$ função recursiva primitiva
5. $g_2(x,y,z) = g_1\big(U_1^3(x,y,z), U_3^3(x,y,z)\big)$ 4,2,3 / composição
6. $g_3(x,0) = N(x)$
   $g_3\big(x,S(y)\big) = g_2\big(x,y,g_3(x,y)\big)$ 1,5 / recursão primitiva

b) **Exponenciação**:

Definição intuitiva:

$x^0 = 1$
$x^{y+1} = x^y\,.\,x$

Derivação:

1. $N(x) = 0$ função inicial
2. $S(x) = x + 1$ função inicial
3. $g_4(x) = S\big(N(x)\big)$ 2,1 / composição
4. $U_1^3(x, y, z) = x$ função inicial
5. $U_3^3(x, y, z) = z$ função inicial
6. $g_3(x, 0) = N(x)$
   $g_3\big(x, S(y)\big) = g_2\big(x, y, g_3(x, y)\big)$ função recursiva primitiva
7. $g_5(x, y, z) = g_3\big(U_1^3(x, y, z), U_3^3(x, y, z)\big)$ 6,4,5 / composição
8. $g_6(x, 0) = g_4(x)$
   $g_6\big(x, S(y)\big) = g_5\big(x, y, g_6(x, y)\big)$ 3,7 / recursão primitiva

c) **Predecessor**:

Definição intuitiva:

$$pd(x) = \begin{cases} x - 1, & se\ x > 0 \\ 0, & se\ x = 0 \end{cases}$$

Derivação:

1. $U_1^2(x, y) = x$ função inicial
2. $g_7(0) = 0$
   $g_7\big(S(y)\big) = U_1^2\big(y, g_7(y)\big)$ 1 / recursão primitiva

d) **Subtração própria**:

Definição intuitiva:

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

Derivação:

1. $U_1^1(x) = x$ função inicial
2. $U_3^3(x, y, z) = z$ função inicial
3. $g_7(0) = 0$

   $g_7\big(S(y)\big) = U_1^2\big(y, g_7(y)\big)$ função recursiva primitiva
4. $g_8(x, y, z) = g_7\big(U_3^3(x, y, z)\big)$ 3,2 / composição
5. $g_9(x, 0) = U_1^1(x)$

   $g_9\big(x, S(y)\big) = g_8\big(x, y, g_9(x, y)\big)$ 1,4 / recursão primitiva

e) **Diferença absoluta**:

Definição intuitiva:

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

Derivação:

1. $g_1(x, 0) = U_1^1(x)$

   $g_1\big(x, S(y)\big) = g_0\big(x, y, g_1(x, y)\big)$ função recursiva primitiva
2. $g_9(x, 0) = U_1^1(x)$

   $g_9\big(x, S(y)\big) = g_8\big(x, y, g_9(x, y)\big)$ função recursiva primitiva
3. $g_{10}(x, y) = g_1\big(g_9(x, y), g_9(y, x)\big)$ 1,2 / composição

f) **Sinal**

Definição intuitiva:

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

Derivação:

1. $C_1(x) = 1$ função recursiva primitiva
2. $U_1^2(x, y) = x$ função inicial
3. $g_{11}(x, y) = C_1\left(U_1^2(x, y)\right)$ 1,2 / composição
4. $g_{12}(0) = 0$
   $g_{12}\left(S(y)\right) = g_{11}\left(y, g_{12}(y)\right)$ 3 / recursão primitiva

g) **Contrassinal**

Definição intuitiva:

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

Derivação:

1. $N(x) = 0$ função inicial
2. $U_1^2(x, y) = x$ função inicial
3. $g_{13}(x, y) = N\left(U_1^2(x, y)\right)$ 1,2 / composição
4. $g_{14}(0) = 1$
   $g_{14}\left(S(y)\right) = g_{13}\left(y, g_{14}(y)\right)$ 3 / recursão primitiva

h) **Fatorial**

Definição intuitiva:

$0! = 1$

$S(y)! = S(y) \,.\, y!$

Derivação:

1. $S(x) = x + 1$ função inicial
2. $U_1^2(x, y)$ função inicial
3. $U_2^2(x, y)$ função inicial
4. $g_3(x, 0) = N(x)$
   $g_3\big(x, S(y)\big) = g_2\big(x, y, g_3(x, y)\big)$ função recursiva primitiva
5. $g_{15}(x, y) = g_3\left(S\big(U_1^2(x, y)\big), U_2^2(x, y)\right)$ 4,1,2,3 / composição
6. $g_{16}(0) = 1$
   $g_{16}\big(S(y)\big) = g_{15}\big(y, g_{16}(y)\big)$ 5 / recursão primitiva

i) **Resto da divisão de y por x**

Definição intuitiva:

$Rm(x, 0) = 0$

$Rm\big(x, S(y)\big) = S\big(Rm(x, y)\big) \,.\, sg\big(\big|x - S\big(Rm(x, y)\big)\big|\big)$

Derivação:

1. $S(x) = x + 1$ função inicial
2. $U_3^3(x, y, z) = z$ função inicial
3. $U_1^3(x, y, z) = x$ função inicial
4. $g_{12}(0) = 0$
   $g_{12}\big(S(y)\big) = g_{11}\big(y, g_{12}(y)\big)$ função recursiva primitiva
5. $g_{10}(x, y) = g_1\big(g_9(x, y), g_9(y, x)\big)$ função recursiva primitiva
6. $g_3(x, 0) = N(x)$
   $g_3\big(x, S(y)\big) = g_2\big(x, y, g_3(x, y)\big)$ função recursiva primitiva
7. $N(x) = 0$ função inicial
8. $g_{17}(x, y, z) = S\big(U_3^3(x, y, z)\big)$ 1,2 / composição
9. $g_{18}(x, y, z) = g_{10}\big(U_1^3(x, y, z), g_{17}(x, y, z)\big)$ 5,3,8 / composição
10. $g_{19}(x, y, z) = g_{12}\big(g_{18}(x, y, z)\big)$ 4,9 / composição

11. $g_{20}(x,y,z) = g_3\big(g_{17}(x,y,z), g_{19}(x,y,z)\big)$ 6,8,10 / composição
12. $g_{21}(x,0) = N(x)$
    $g_{21}\big(x,S(y)\big) = g_{20}\big(x,y,g_{21}(x,y)\big)$ 7,11 / recursão primitiva

j) **Quociente da divisão de y por x**

Definição intuitiva:

$Qt(x,0) = 0$
$Qt\big(x,S(y)\big) = Qt(x,y) + \overline{sg}\big(\big|x - S\big(Rm(x,y)\big)\big|\big)$

Derivação:

1. $U_1^3(x,y,z) = x$ função inicial
2. $U_3^3(x,y,z) = z$ função inicial
3. $N(x) = 0$ função inicial
4. $S(x) = x + 1$ função inicial
5. $g_1(x,0) = U_1^1(x)$
   $g_1\big(x,S(y)\big) = g_0\big(x,y,g_1(x,y)\big)$ função recursiva primitiva
6. $g_{14}(0) = 1$
   $g_{14}\big(S(y)\big) = g_{13}\big(y,g_{14}(y)\big)$ função recursiva primitiva
7. $g_{10}(x,y) = g_1\big(g_9(x,y), g_9(y,x)\big)$ função recursiva primitiva
8. $g_{21}(x,0) = N(x)$
   $g_{21}\big(x,S(y)\big) = g_{20}\big(x,y,g_{21}(x,y)\big)$ função recursiva primitiva
9. $g_{22}(x,y,z) = g_{21}\big(U_1^3(x,y,z), U_3^3(x,y,z)\big)$ 8,1,2 / composição
10. $g_{23}(x,y,z) = S\big(g_{22}(x,y,z)\big)$ 4,9 / composição
11. $g_{24}(x,y,z) = g_{10}\big(U_1^3(x,y,z), g_{23}(x,y,z)\big)$ 7,1,10 / composição
12. $g_{25}(x,y,z) = g_{14}\big(g_{24}(x,y,z)\big)$ 6,11 / composição
13. $g_{26}(x,y,z) = g_1\big(U_3^3(x,y,z), g_{25}(x,y,z)\big)$ 5,2,12 / composição
14. $g_{27}(x,0) = N(x)$
    $g_{27}\big(x,S(y)\big) = g_{26}\big(x,y,g_{27}(x,y)\big)$ 3,13 / recursão primitiva

Como acabamos de ver, para toda função recursiva primitiva há uma derivação recursiva primitiva, isto é, um algoritmo para computá-la. Dessa forma, dizemos que toda função recursiva primitiva é algorítmica. Portanto, via tese de Church-Turing, podemos afirmar que as funções recursivas primitivas são parcialmente Turing-computáveis. Porém, como toda função recursiva primitiva é total, ela é Turing-computável. Com base nisto, mostraremos no próximo capítulo que embora toda função recursiva primitiva seja algorítmica, há ao menos uma função algorítmica que não é recursiva primitiva. Portanto, a classe das funções recursivas primitivas constitui uma subclasse das funções Turing-computáveis.

2.2. SOMAS E PRODUTOS LIMITADOS

Definiremos a seguir algumas operações, as quais quando aplicadas à funções recursivas primitivas, geram novas funções recursivas primitivas.

Definição 4:

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

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

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

$$\prod_{y\leq z} f(x_1, \dots, x_n, y) = \prod_{y<z+1} f(x_1, \dots, x_n, y)$$

Teorema 2: Se $f(x_1, \dots, x_n, y)$ é recursiva primitiva, então $\sum_{y<z} f(x_1, \dots, x_n, y)$ é recursiva primitiva.

Prova:

1. $U_1^{n+2}(x_1, \dots, x_n, y, w) = x_1$ função inicial

⋮

n+1. $U_{n+1}^{n+2}(x_1, \dots, x_n, y, w) = y$ função inicial

n+2. $U_{n+2}^{n+2}(x_1, \dots, x_n, y, w) = w$ função inicial

n+3. $U_1^n(x_1, \dots, x_n) = x_1$ função inicial

n+4. $N(x) = 0$ função inicial

n+5. $g_1(x, 0) = U_1^1(x)$

$g_1\big(x, S(y)\big) = g_0\big(x, y, g_1(x, y)\big)$ função recursiva primitiva

n+6. $f(x_1, \dots, x_n, y)$ função recursiva primitiva por hipótese

n+7. $h_3(x_1, \dots, x_n) = N\big(U_1^n(x_1, \dots x_n)\big)$ n+4,n+3 / composição

n+8. $h_4(x_1, \dots, x_n, y, w) = f\big(U_1^{n+2}(x_1, \dots, x_n, y, w), \dots, U_{n+1}^{n+2}(x_1, \dots, x_n, y, w)\big)$ n+6,1,n+1 / composição

n+9. $h_5(x_1, \dots, x_n, y, w) = g_1\big(h_4(x_1, \dots, x_n, y, w), U_{n+2}^{n+2}(x_1, \dots, x_n, y, w)\big)$ n+5,n+8,n+2 / composição

n+10. $g_{31}(x_1, \dots, x_n, 0) = h_3(x_1, \dots, x_n)$

$g_{31}\big(x_1, \dots, x_n, S(z)\big) = h_5\big(x_1, \dots, x_n, z, g_{31}(x_1, \dots x_n, z)\big)$ n+7,n+9 / recursão primitiva

Corolário 1: Se $f(x_1, \dots, x_n, y)$ é recursiva primitiva, então $\sum_{y \le z} f(x_1, \dots, x_n, y)$ é recursiva primitiva.

Prova:

1. $U_1^{n+1}(x_1, \dots, x_n, z) = x_1$ função inicial

⋮

n. $U_n^{n+1}(x_1, \dots, x_n, z) = x_n$ função inicial

n+1. $U_{n+1}^{n+1}(x_1, \dots, x_n, z) = z$ função inicial

n+2. $S(x) = x + 1$ função inicial

n+3. $a(x_1, \dots, x_n, z) = S\big(U_{n+1}^{n+1}(x_1, \dots, x_n, z)\big)$ n+2,n+1 / composição

n+4. $g_{31}(x_1, \dots, x_n, 0) = h_3(x_1, \dots, x_n)$

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

n+5. $g_{32}(x_1, \dots, x_n, z) = g_{31}(U_1^{n+1}(x_1, \dots, x_n, z), \dots, U_n^{n+1}(x_1, \dots, x_n, z), a(x_1, \dots, x_n, z))$

n+4,1,n,n+3 / composição

Teorema 3: Se $f(x_1, \dots, x_n, y)$ é recursiva primitiva, então $\prod_{y<z} f(x_1, \dots, x_n, y)$ é recursiva primitiva.

Prova:

A prova é análoga ao teorema 2.

Corolário 2: Se $f(x_1, \dots, x_n, y)$ é recursiva primitiva, então $\prod_{y\leq z} f(x_1, \dots, x_n, y)$ é recursiva primitiva.

Prova:

A prova é análoga ao corolário 1.

2.3. RELAÇÕES E PREDICADOS RECURSIVOS PRIMITIVOS

Como sabemos, em Teoria da Computabilidade trabalhamos apenas com números naturais. Dessa forma, dizemos que uma relação numérica n-ária R ($n \geq 1$), é um subconjunto de N^n.

Definição 5: A função característica de uma relação n-ária R é a seguinte:

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

Definição 6: Uma relação $R(x_1, \dots, x_n)$ é recursiva primitiva se e somente se sua função característica $\chi_R(x_1, \dots, x_n)$ é recursiva primitiva.

Teorema 4: A classe das relações recursivas primitivas n-árias é fechada sob as operações de união, interseção e complemento.

Prova:

Dadas duas relações recursivas primitivas n-árias, a saber, R e S, e suas respectivas funções características χ_R e χ_S, temos:

1. $\chi_{R\cup S} = (\chi_R + \chi_S) \dot{-} (\chi_R \cdot \chi_S)$
2. $\chi_{R\cap S} = \chi_R \cdot \chi_S$
3. $\chi_{\overline{R}} = 1 \dot{-} \chi_R$

Já a contrapartida linguística das relações numéricas, é o que chamamos de predicado numérico. Vale ressaltar que o conjunto dos predicados numéricos é infinito enumerável, ao passo em que o conjunto das relações é infinito não-enumerável. Portanto, há relações para as quais não existem contrapartida linguística. Um predicado numérico n-ário ($n \geq 1$), é uma expressão com n variáveis (com ou sem subscrito), que torna-se uma sentença (verdadeira ou falsa) quando estas variáveis são substituídas[3] por números naturais. Para darmos um exemplo, considere a seguinte expressão: $x + y = 4$. Como sabemos, esta expressão não constitui uma sentença, entretanto, quando substituímos as variáveis por números, obtemos uma sentença que pode ser verdadeira: $2 + 2 = 4$; ou falsa: $2 + 3 = 4$.

Observação: Fazendo uso dos quantificadores ∀ e ∃, podemos também quantificar as variáveis de um predicado para obtermos uma sentença.

Definição 7: Se $P(x_1, \dots, x_n)$ é um predicado n-ário, sua extensão é a relação n-ária $\{(x_1, \dots, x_n) : P(x_1, \dots, x_n)\}$. Ou seja, o conjunto de todas as n-uplas $(x_1, \dots, x_n)$ para as quais $P(x_1, \dots, x_n)$ é verdadeiro.

[3] Uma variável que ocorre mais de uma vez na expressão, é substituída pelo mesmo número natural.

Exemplo: Considere o predicado $x + y = 4$. A sua extensão, $\{(x, y) : x + y = 4\}$, é constituída por todos os pares ordenados que o tornam verdadeiro, a saber, $(2, 2)$, $(3, 1)$, $(1, 3)$, $(4, 0)$ e $(0, 4)$.

Definição 8: A função característica de um predicado n-ário $P(x_1, \dots, x_n)$, é a função característica de sua extensão $\{(x_1, \dots, x_n) : P(x_1, \dots, x_n)\}$, e é escrita $\chi_P(x_1, \dots, x_n)$. Dessa forma, temos:

$$\chi_P(x_1, \dots, x_n) = \begin{cases} 1, & se\ P(x_1, \dots, x_n)\ \text{é}\ verdadeiro \\ 0, & caso\ contrário \end{cases}$$

Definição 9: Um predicado n-ário $P(x_1, \dots, x_n)$ é recursivo primitivo, se sua função característica é recursiva primitiva.

Dado um ou mais predicados, podemos obter novos predicados fazendo uso dos conectivos proposicionais. O teorema a seguir mostra que, se os predicados previamente dados forem recursivos primitivos, os que obteremos com auxílio dos conectivos proposicionais também serão recursivos primitivos.

Teorema 5: Se $R(x_1, \dots, x_n)$ e $S(x_1, \dots, x_n)$ são predicados recursivos primitivos n-ários, então os seguintes predicados são recursivos primitivos:

1. $\neg R(x_1, \dots, x_n)$
2. $R(x_1, \dots, x_n) \wedge S(x_1, \dots, x_n)$
3. $R(x_1, \dots, x_n) \vee S(x_1, \dots, x_n)$
4. $R(x_1, \dots, x_n) \rightarrow S(x_1, \dots, x_n)$
5. $R(x_1, \dots, x_n) \leftrightarrow S(x_1, \dots, x_n)$

Prova:

1. $\chi_{\neg R} = 1 \dot{-} \chi_R$
2. $\chi_{R \wedge S} = \chi_R \cdot \chi_S$
3. $\chi_{R \vee S} = (\chi_R + \chi_S) \dot{-} (\chi_R \cdot \chi_S)$
4. $\chi_{R \rightarrow S} = \chi_{\neg R \vee S}$

5. $\chi_{R\leftrightarrow S} = \chi_{(R\rightarrow S)\wedge(S\rightarrow R)}$

Além dos conectivos proposicionais, podemos também aplicar os quantificadores limitados a predicados recursivos primitivos $(n+1)$-ários previamente dados, com $(n \geq 0)$, para obtermos novos predicados recursivos primitivos $(n+1)$-ários. Estes quantificadores limitados serão definidos a seguir.

Definição 10:

$$\bigvee_{y<z} P(x_1, \dots, x_n, y) \leftrightarrow P(x_1, \dots, x_n, 0) \vee P(x_1, \dots, x_n, 1) \vee \dots \vee P(x_1, \dots, x_n, z-1)$$

$$\bigvee_{y\leq z} P(x_1, \dots, x_n, y) \leftrightarrow P(x_1, \dots, x_n, 0) \vee P(x_1, \dots, x_n, 1) \vee \dots \vee P(x_1, \dots, x_n, z)$$

$$\bigwedge_{y<z} P(x_1, \dots, x_n, y) \leftrightarrow P(x_1, \dots, x_n, 0) \wedge P(x_1, \dots, x_n, 1) \wedge \dots \wedge P(x_1, \dots, x_n, z-1)$$

$$\bigwedge_{y\leq z} P(x_1, \dots, x_n, y) \leftrightarrow P(x_1, \dots, x_n, 0) \wedge P(x_1, \dots, x_n, 1) \wedge \dots \wedge P(x_1, \dots, x_n, z)$$

Os símbolos $\vee$ e $\wedge$, denotam o quantificador limitado existencial, e o quantificador limitado universal, respectivamente.

Teorema 6: Se $P(x_1, \dots, x_n, y)$ é recursivo primitivo, então $\vee_{y<z} P(x_1, \dots, x_n, y)$ é recursivo primitivo.

Prova:

Seja $Q(x_1, \dots, x_n, z)$ o predicado $\vee_{y<z} P(x_1, \dots, x_n, y)$. Então:

$$\chi_Q(x_1, \dots, x_n, z) = sg\left(\sum_{y<z} \chi_P(x_1, \dots, x_n, y)\right)$$

Corolário 3: Se $P(x_1, \dots, x_n, y)$ é recursivo primitivo, então $\vee_{y\leq z} P(x_1, \dots, x_n, y)$ é recursivo primitivo.

Prova:

Seja $R(x_1, \dots, x_n, z)$ o predicado $\bigvee_{y \leq z} P(x_1, \dots, x_n, y)$. Então:

$$\chi_R(x_1, \dots, x_n, z) = sg\left(\sum_{y \leq z} \chi_P(x_1, \dots, x_n, y)\right)$$

Teorema 7: Se $P(x_1, \dots, x_n, y)$ é recursivo primitivo, então $\bigwedge_{y<z} P(x_1, \dots, x_n, y)$ é recursivo primitivo.

Prova:

Seja $S(x_1, \dots, x_n, z)$ o predicado $\bigwedge_{y<z} P(x_1, \dots, x_n, y)$. Então:

$$\chi_S(x_1, \dots, x_n, z) = \prod_{y<z} \chi_P(x_1, \dots, x_n, y)$$

Corolário 4: Se $P(x_1, \dots, x_n, y)$ é recursivo primitivo, então $\bigwedge_{y \leq z} P(x_1, \dots, x_n, y)$ é recursivo primitivo.

Prova:

Seja $T(x_1, \dots, x_n, z)$ o predicado $\bigwedge_{y \leq z} P(x_1, \dots, x_n, y)$. Então:

$$\chi_T(x_1, \dots, x_n, z) = \prod_{y \leq z} \chi_P(x_1, \dots, x_n, y)$$

Apresentaremos a seguir, em forma de teorema, alguns predicados recursivos primitivos que serão retomados adiante.

Teorema 8: Os seguintes predicados são recursivos primitivos:

a) $x = y$

Prova: Sua função característica é dada por $\overline{sg}\,|x - y|$

b) $x < y$

Prova: Sua função característica é dada por $sg(y \dot{-} x)$

c) $x \mid y$, isto é, "$x\ divide\ y$"

Prova: Sua função característica é dada por $\overline{sg}\left(Rm(x,y)\right)$

d) Primo(x), isto é, "x é $primo$"

Prova: Primo(x) se, e somente se, $x \neq 0 \wedge x \neq 1 \wedge \bigwedge_{y \leq x}\big(y \mid x \rightarrow (y = 1 \vee y = x)\big)$. Sua função característica é dada por:

$$(sg|x-0| \,.\, sg|x-1|) \,.\, \prod_{y \leq x} sg\left(sg\big(Rm(y,x)\big) + sg(\overline{sg}|y-1| + \overline{sg}|y-x|)\right)$$

Definiremos agora o μ-operador limitado, que quando aplicado a predicados $(n+1)$-ários já definidos, nos permite definir funções recursivas primitivas.

Definição 11:

$$\mu_{y<z}\, P(x_1, \dots, x_n, y) = \begin{cases} o\ menor\ y\ t.q.\ 0 \leq y < z\ e\ P(x_1, \dots, x_n, y), se \bigvee_{y<z} P(x_1, \dots, x_n, y) \\ 0, \qquad\qquad se \neg \bigvee_{y<z} P(x_1, \dots, x_n, y) \end{cases}$$

Teorema 9: Se $P(x_1, \dots, x_n, y)$, $n \geq 0$, é um predicado recursivo primitivo, então a função $\mu_{y<z}\, P(x_1, \dots, x_n, y)$ é recursiva primitiva.

Prova:

$$\mu_{y<z}\, P(x_1, \dots, x_n, y) = sg\left(\sum_{y<z} \chi_P(x_1, \dots, x_n, y)\right) . \sum_{y<z}\left(\prod_{u \leq y} \overline{sg}\chi_P(x_1, \dots, x_n, u)\right)$$

Observação: Se não existe $y < z$ tal que $P(x_1, \dots, x_n, y)$, então $sg\left(\sum_{y<z} \chi_P(x_1, \dots, x_n, y)\right) = 0$. Entretanto, se existe tal y , então $sg\left(\sum_{y<z} \chi_P(x_1, \dots, x_n, y)\right) = 1$, e o menor $y < z$ será o valor determinado pela outra função, a saber, $\sum_{y<z}\left(\prod_{u\leq y} \overline{sg}\chi_P(x_1, \dots, x_n, u)\right)$, que soma 1 cada vez que $\chi_P(x_1, \dots, x_n, u) = 0$, para $u \leq y$. Esta função é desenvolvida da seguinte forma:

$$\prod_{u\leq 0} \overline{sg}\chi_P(x_1, \dots, x_n, u) + \prod_{u\leq 1} \overline{sg}\chi_P(x_1, \dots, x_n, u) + \cdots + \prod_{u\leq z-1} \overline{sg}\chi_P(x_1, \dots, x_n, u)$$

Exemplo: Considere o predicado $P(x, y)$ como $(x < y)$. Dessa forma, aplicando o μ-operador limitado a ele e atribuindo para x e z os valores 1 e 3 respectivamente, temos:

$$\mu_{y<3}\, P(1, y) = 1\,.\sum_{y<3}\left(\prod_{u\leq y} \overline{sg}\chi_P(1, u)\right)$$

Agora fazemos os seguintes cálculos:

$$\prod_{u\leq 0} \overline{sg}\chi_P(1, u) + \prod_{u\leq 1} \overline{sg}\chi_P(1, u) + \prod_{u\leq 2} \overline{sg}\chi_P(1, u)$$

$\overline{sg}\chi_P(1,0) +$
$\overline{sg}\chi_P(1,0)\,.\,\overline{sg}\chi_P(1,1) +$
$\overline{sg}\chi_P(1,0)\,.\,\overline{sg}\chi_P(1,1)\,.\,\overline{sg}\chi_P(1,2)$

Isto é, $1 + (1.1) + (1.1.0) = 2$.

2.4. ARITMETIZAÇÃO DAS FUNÇÕES E DERIVAÇÕES RECURSIVAS PRIMITIVAS

De forma análoga à aritmetização das máquinas de Turing vista no capítulo anterior, apresentaremos aqui a aritmetização das funções e derivações recursivas primitivas. Como a classe das funções recursivas primitivas é fechada sobre as

funções básicas e sobre as operações de composição e recursão primitiva, atribuiremos inicialmente às funções básicas a seguinte codificação:

c'(N) = 11
c'(S) = 13
c'(U_i^n) = p_{i+5}^n

Observação: Atribuímos o símbolo c' para codificar as funções e o símbolo c para codificar as derivações.

Codificaremos a seguir funções obtidas pela operação de composição. Quando $f(x_1, \dots, x_n) = h\big(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n)\big)$, sendo c'($h$) = a e c'(g_i) = a_i, para $1 \leq i \leq m$, então:

$$c'(f) = 2^a \cdot 3^{p_0^{a_1} \cdot \dots \cdot p_{m-1}^{a_m}}$$

Exemplo: Dada a função $f(x) = S\big(U_1^1(x)\big)$, o número de Gödel para ela será:

$$c'(f) = 2^{c'(S)} \cdot 3^{2^{c'\left(U_1^1\right)}} = 2^{13} \cdot 3^{2^{17}}$$

No que diz respeito à codificação das funções obtidas pela operação de recursão primitiva, quando f é obtida de g e h, sendo c'(g) = b e c'(h) = d, então:

$$c'(f) = 5^b \cdot 7^d$$

Exemplo: Considere a seguinte função:

$$f(x, 0) = N(x)$$
$$f\big(x, S(y)\big) = U_3^3\big(x, y, f(x, y)\big)$$

O número de Gödel para ela será o seguinte:

$$c'(f) = 5^{c'(N)} \cdot 7^{c'\left(U_3^3\right)} = 5^{11} \cdot 7^{23^3}$$

Ao passo em que apresentamos a aritmetização para as funções recursivas primitivas, apresentaremos agora a aritmetização para as derivações recursivas primitivas. Seja β uma derivação recursiva primitiva com n passos, β = (β_1,...,β_n). Dito isto, a codificação para β é obtida da seguinte forma:

$$c(\beta) = \prod_{i<n} p_i^{c'(\beta_{i+1})}$$

Exemplo: Voltemos ao exemplo de derivação recursiva primitiva para a função soma apresentado no início do capítulo:

β1. $S(x) = x + 1$ função inicial
β2. $U_3^3(x, y, z) = z$ função inicial
β3. $g_0(x, y, z) = S\left(U_3^3(x, y, z)\right)$ 1,2 / composição
β4. $U_1^1(x) = x$ função inicial
β5. $g_1(x, 0) = U_1^1(x)$
$g_1\left(x, S(y)\right) = g_0\left(x, y, g_1(x, y)\right)$ 4,3 / recursão primitiva

Para codificarmos β, precisamos inicialmente codificar cada função de β da seguinte forma:

$c'(\beta_1) = c'(S) = 13$
$c'(\beta_2) = c'(U_3^3) = p_{3+5}^3$
$c'(\beta_3) = c'(g_0) = 2^{13} \cdot 3^{p_0^{p_{3+5}^3}}$
$c'(\beta_4) = c'(U_1^1) = p_{1+5}^1$
$c'(\beta_5) = c'(g_1) = 5^{p_{1+5}^1} \cdot 7^{2^{13} \cdot 3^{p_0^{p_{3+5}^3}}}$

Feito isto, codificaremos β como segue:

$$c(\beta) = 2^{c'(S)} \cdot 3^{c'(U_3^3)} \cdot 5^{c'(g_0)} \cdot 7^{c'(U_1^1)} \cdot 11^{c'(g_1)}$$

$$= 2^{13} \cdot 3^{p_{3+5}^3} \cdot 5^{2^{13} \cdot 3^{p_0^{p_{3+5}^3}}} \cdot 7^{p_{1+5}^1} \cdot 11^{5^{p_{1+5}^1} \cdot 7^{2^{13}} \cdot 3^{p_0^{p_{3+5}^3}}}$$

Como vimos na aritmetização das máquinas de Turing, assim como podemos codificar as funções e derivações recursivas primitivas, também podemos, dado um número natural n, decodifica-las, da seguinte forma: fatoramos n em fatores primos, depois fatoramos seus expoentes em fatores primos, e assim por diante, até obtermos uma expressão constituída somente de primos. Dessa forma, se n for um número de Gödel nós podemos encontrar a derivação recursiva primitiva da qual n é código.

3. ALGUNS RESULTADOS PARA A CLASSE DAS FUNÇÕES RECURSIVAS PRIMITIVAS

Neste capítulo, apresentaremos alguns resultados para a classe das funções recursivas primitivas, tais como: a existência de um algoritmo reconhecedor para a classe das derivações recursivas primitivas, função universal para a classe das funções recursivas primitivas, e por fim, mostraremos através de uma generalização do método da diagonal, que existe ao menos uma função que é algorítmica e não é recursiva primitiva.

3.1. ALGORITMO RECONHECEDOR PARA A CLASSE DAS DERIVAÇÕES RECURSIVAS PRIMITIVAS

Quando falamos em um algoritmo reconhecedor para uma classe de algoritmos, entendemos um algoritmo que decide se um dado conjunto de instruções pertence ou não à classe. A existência de um algoritmo reconhecedor para uma classe de algoritmos é a solução para o problema de decisão dessa classe.

Apresentaremos nesta seção, resposta para a seguinte questão: dada a classe das derivações recursivas primitivas, há um algoritmo reconhecedor para ela? Bem, de acordo com Dias e Weber (2010), sabe-se que não existe um algoritmo para decidir, dado um conjunto finito (não-vazio) de instruções, se este conjunto pertence ou não à classe de todos os algoritmos. Entretanto, existe um algoritmo reconhecedor para uma subclasse desta classe supracitada, a saber, a classe das derivações recursivas primitivas.

Provamos intuitivamente a existência de um algoritmo reconhecedor para a classe das derivações recursivas primitivas, da seguinte forma:

Seja $DRP = \{x : x \text{ é } código\ de\ uma\ derivação\ recursiva\ primitiva\}$

Sua função característica é a seguinte:

$$\chi_{DRP}(x) = \begin{cases} 1, & se\ x\ \in DRP \\ 0, & se\ x\ \notin DRP \end{cases}$$

Esta função característica é algorítmica. De fato, como foi mostrado na seção *2.4*, dado um número natural x, quando você o decodifica, você tem como saber se x é código ou não de uma derivação recursiva primitiva. Isto encerra a prova.

Observação: Embora tenhamos provado que essa função característica é algorítmica, isto é, que existe um algoritmo para computá-la, nós não sabemos como construir um algoritmo específico para ela.

Com base no algoritmo reconhecedor para a classe das derivações recursivas primitivas, podemos listar todas as derivações recursivas primitivas, definindo a seguinte função:

$$f(0) = \mu_y(\chi_{DRP}(y) = 1)$$
$$f(n+1) = \mu_y\big(\chi_{DRP}(y) = 1 \wedge y > f(n)\big)$$

Ao passo em que fatoramos os números naturais no segmento inicial dos números primos, quando encontramos o primeiro número que é código de uma derivação recursiva primitiva, este código nos dará a primeira derivação da lista, a saber, $f(0)$. O segundo nos dará a segunda derivação da lista, isto é, $f(1)$. O terceiro nos dará $f(2)$. E assim sucessivamente. Esta lista é infinita, porém, cada derivação pode ser encontrada em algum ponto finito da mesma.

3.2. FUNÇÃO UNIVERSAL PARA A CLASSE DAS FUNÇÕES RECURSIVAS PRIMITIVAS

Nesta seção definiremos uma função universal para a classe das funções recursivas primitivas, e mostraremos adiante que embora essa função nos permita listar e computar todas as funções recursivas primitivas, ela não é recursiva primitiva.

Definição 1: Uma função universal U para uma classe C de funções n-árias, com $(n \geq 1)$, é uma função $1+n$-ária que enumera as funções de C e seus argumentos. Portanto:

$$U(x, y_1, \dots, y_n) = \begin{cases} \varphi_x(y_1, \dots, y_n), & se\ \varphi_x(y_1, \dots, y_n) \downarrow \\ \uparrow, & se\ \varphi_x(y_1, \dots, y_n) \uparrow \end{cases}$$

Observação: φ_x é a função cuja derivação tem como código x.

Definindo U para a classe das funções recursivas primitivas n-árias, temos:

$$U(f(x), y_1, \dots, y_n) = \begin{cases} \varphi_{f(x)}(y_1, \dots, y_n), & se\ \varphi_{f(x)}(y_1, \dots, y_n) \downarrow \\ \uparrow, & se\ \varphi_{f(x)}(y_1, \dots, y_n) \uparrow \end{cases}$$

De posse dessa função universal, podemos listar e computar efetivamente todas as funções recursivas primitivas n-árias, como segue:

1. Após listarmos efetivamente todas as derivações recursivas primitivas (como vimos na seção anterior), podemos listar efetivamente todas as funções recursivas primitivas n-árias, da seguinte forma:

$$\begin{array}{cccc} \varphi_{f(0)}(y_1, \dots, y_{n-1}, 0), & \varphi_{f(0)}(y_1, \dots, y_{n-1}, 1), & \varphi_{f(0)}(y_1, \dots, y_{n-1}, 2) & \cdots \\ \varphi_{f(1)}(y_1, \dots, y_{n-1}, 0), & \varphi_{f(1)}(y_1, \dots, y_{n-1}, 1), & \varphi_{f(1)}(y_1, \dots, y_{n-1}, 2) & \cdots \\ \varphi_{f(2)}(y_1, \dots, y_{n-1}, 0), & \varphi_{f(2)}(y_1, \dots, y_{n-1}, 1), & \varphi_{f(2)}(y_1, \dots, y_{n-1}, 2) & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

Observação: Percebe-se que $\varphi_{f(x)}(y_1, \dots, y_{n-1}, z)$ é a função recursiva primitiva n-ária cuja derivação tem como código $f(x)$ e $(y_1, \dots, y_{n-1}, z)$ como argumentos. Os argumentos $(y_1, \dots, y_{n-1})$ são fixados aqui como parâmetros.

2. U computa todas as funções recursivas primitivas n-árias da seguinte forma:

Como vimos, $U(f(x), y_1, \dots, y_n) = \varphi_{f(x)}(y_1, \dots, y_n)$. Dessa forma, se U tem como argumentos $(f(x), y_1, \dots, y_n)$, o que temos que fazer é encontrar a $f(x)$-ésima derivação recursiva primitiva e aplicá-la aos argumentos de φ, a saber, $(y_1, \dots, y_n)$.

Quando, e somente quando, $\varphi_{f(x)}(y_1, \dots, y_n)$ tem um valor, atribua esse valor a $U(f(x), y_1, \dots, y_n)$. Entretanto, como toda função recursiva primitiva é total, sempre haverá um valor para $U(f(x), y_1, \dots, y_n)$.

Observação: Podemos reduzir todas as funções recursivas primitivas n-árias, com $(n \geq 2)$, à funções recursivas primitivas de uma variável, através da função J de Cantor[4] – a qual não apresentaremos aqui – que nos possibilita codificar n-uplas.

3.3. NEM TODA FUNÇÃO ALGORÍTMICA É RECURSIVA PRIMITIVA

Como vimos na seção *2.1*, toda função recursiva primitiva é algorítmica, e portanto, via tese de Church-Turing, é Turing-computável. Entretanto, o contrário não acontece, isto é, nem toda função algorítmica é recursiva primitiva.

Teorema 1: Para toda classe de funções recursivas primitivas n-árias $(n \geq 1)$, existe ao menos uma função n-ária que é algorítmica, e não é recursiva primitiva.

Prova:

Tendo em vista as considerações feitas nas seções *3.1* e *3.2*, vamos mostrar através de uma generalização do método da diagonal – método que pode ser aplicado a uma variedade de classes formalmente caracterizadas de funções, e que, em cada caso, a partir do traçado da diagonal a uma classe previamente dada, produz uma função que não pertence à classe –, como construir uma função algorítmica que não é recursiva primitiva.

Defina a seguinte função: $g(y_1, \dots, y_{n-1}, x) = U(f(x), y_1, \dots, y_{n-1}, x) + 1 = \varphi_{f(x)}(y_1, \dots, y_{n-1}, x) + 1$. Sabemos que g é algorítmica, pois para computá-la, basta encontrarmos $\varphi_{f(x)}(y_1, \dots, y_{n-1}, x)$ na lista de todas as funções recursivas primitivas, e somar 1 ao seu valor. Agora, suponha que g é uma função recursiva primitiva n-ária.

[4] Uma exposição detalhada a cerca dessa função, pode ser encontrada em DIAS, Matias Francisco e WEBER, Leonardo. *Teoria da recursão*. São Paulo: Ed. UNESP, 2010, p.134.

Se isto ocorre, então g é uma das funções da lista, isto é, $g = \varphi_{f(i)}$ para algum i. Portanto, $g(y_1, \dots, y_{n-1}, i) = \varphi_{f(i)}(y_1, \dots, y_{n-1}, i)$. Mas, por definição, temos que, $g(y_1, \dots, y_{n-1}, i) = \varphi_{f(i)}(y_1, \dots, y_{n-1}, i) + 1$. Portanto, temos que, $\varphi_{f(i)}(y_1, \dots, y_{n-1}, i) = \varphi_{f(i)}(y_1, \dots, y_{n-1}, i) + 1$, o que nos leva a uma contradição. Dessa forma, concluímos que, embora g seja algorítmica, ela não está na lista e portanto ela não é recursiva primitiva.

Provamos acima que nem toda função algorítmica é recursiva primitiva. Apresentaremos a seguir, como corolário, que a função universal para a classe das funções recursivas primitivas, apresentada na seção *3.2*, é algorítmica mas não é recursiva primitiva.

Corolário 1: A função universal definida para a classe das funções recursivas primitivas não é recursiva primitiva.

Prova:

Com base nos resultados anteriores, e levando em consideração que podemos reduzir todas as funções recursivas primitivas n-árias à funções recursivas primitivas de uma variável, após listarmos efetivamente todas as derivações recursivas primitivas, listamos todas as funções recursivas primitivas de uma variável da seguinte forma:

Definimos $U(f(x), y) = \varphi_{f(x)}(y)$, e portanto temos a seguinte listagem:

$$\begin{matrix} \varphi_{f(0)}(0), \varphi_{f(0)}(1), \varphi_{f(0)}(2) \cdots \\ \varphi_{f(1)}(0), \varphi_{f(1)}(1), \varphi_{f(1)}(2) \cdots \\ \varphi_{f(2)}(0), \varphi_{f(2)}(1), \varphi_{f(2)}(2) \cdots \\ \vdots \qquad \vdots \qquad \vdots \qquad \ddots \end{matrix}$$

Agora que temos uma lista com todas as funções recursivas primitivas de uma variável, mostraremos por diagonalização que a função universal para a classe das funções recursivas primitivas não é recursiva primitiva.

Hipótese: Se $U(f(x), y) = \varphi_{f(x)}(y)$ é recursiva primitiva, então $U(f(x), x) = \varphi_{f(x)}(x) = h(x)$ é recursiva primitiva por composição.

Defina a seguinte função $g(x) = h(x) + 1 = \varphi_{f(x)}(x) + 1$. Sabemos que g é algorítmica. De fato, basta encontrarmos $\varphi_{f(x)}(x)$ na lista de todas as funções recursivas primitivas, e somar 1 ao seu valor. Agora, suponha que $g(x)$ é uma função recursiva primitiva de uma variável. Se isto ocorre, g é uma das funções da lista, isto é, $g = \varphi_{f(k)}$ para algum k. Portanto, $g(k) = \varphi_{f(k)}(k)$. Mas por definição, $g(k) = \varphi_{f(k)}(k) + 1$. Portanto, temos que, $\varphi_{f(k)}(k) = \varphi_{f(k)}(k) + 1$, o que nos leva a uma contradição. Dessa forma, concluímos que, embora g seja algorítmica, ela não está na lista, e portanto, ela não é recursiva primitiva.

Percebe-se que se $U(f(x), y)$ fosse recursiva primitiva, então $U(f(x), x)$ seria recursiva primitiva, e portanto $U(f(x), x) + 1$ seria recursiva primitiva. Entretanto, temos que $U(f(x), x) + 1 = \varphi_{f(x)}(x) + 1 = g(x)$, que não é recursiva primitiva.

CONCLUSÃO

Vimos que, a classe das funções recursivas primitivas constitui uma subclasse das funções Turing-computáveis. Embora a classe das funções recursivas primitivas não seja equivalente à classe de todas as funções algorítmicas, trabalhamos com esta classe especial de funções pelo fato de que muitas das funções que conhecemos como algorítmicas, como por exemplo, a soma, a multiplicação, a exponenciação, entre outras, são recursivas primitivas. Como a classe das funções recursivas primitivas não dá conta de todas as funções algorítmicas, apresentamos, num primeiro momento, uma das mais conhecidas tentativas de se formalizar a noção intuitiva de algoritmo, a saber, a teoria das máquinas de Turing, para só então podermos trabalhar com a classe das funções recursivas primitivas e chegar a resultados importantes para esta classe.

Para que pudéssemos alcançar nossos objetivos, dividimos nosso trabalho em três capítulos porque na primeira parte precisamos apresentar através de algumas definições, a teoria das máquinas e programas de Turing, a qual constitui uma versão formal para a noção intuitiva de algoritmo. Isto, para que na segunda parte pudéssemos apresentar a aritmetização das máquinas de Turing, isto é, mostrarmos como codificar essa teoria, atribuindo números naturais aos componentes básicos de sua linguagem, com o fim de traduzir seus enunciados metalinguísticos na linguagem-objeto da aritmética. De posse desses resultados, pudemos apresentar, na terceira parte, a famosa tese de Church-Turing em suas duas versões, a estrita e a ampla.

A partir de então, foi preciso organizar o segundo capítulo em quatro partes. Na primeira parte, precisamos exibir a classe das funções recursivas primitivas por meio de definições e teoremas, e mostrarmos que ela constitui uma subclasse das funções Turing-computáveis. Em seguida, precisamos apresentar através de algumas definições, teoremas e corolários, as operações de soma e produto limitado, as quais, quando aplicadas à funções recursivas primitivas, geram novas funções recursivas primitivas. De posse disso, introduzimos, na terceira parte, relações e predicados recursivos primitivos, para, na quarta parte, apresentarmos a aritmetização da classe das funções recursivas primitivas.

Tendo exibido a classe das funções recursivas primitivas, apresentamos no nosso terceiro e último capítulo, soluções para as três questões concernentes a esta

classe de funções, que propomos como objetivo do nosso trabalho. Nossos resultados mostram que:

1. Existe um algoritmo reconhecedor para a classe das derivações recursivas primitivas, a qual constitui uma subclasse da classe de todos os algoritmos. Consideramos inicialmente o conjunto DRP e sua função característica. A partir disso, mostramos que essa função característica é algorítmica, isto é, existe um algoritmo para computá-la. Contudo, embora essa função seja algorítmica, nós não sabemos como construir um algoritmo específico para ela.
2. Existe uma função universal para a classe das funções recursivas primitivas. De fato, definimos uma função $1+n$-ária que enumera e computa todas as funções recursivas primitivas n-árias $(n \geq 1)$.
3. Toda função recursiva primitiva é algorítmica, mas nem toda função algorítmica é recursiva primitiva. Como toda função recursiva primitiva é algorítmica, concluímos, via tese de Church-Turing, na sua versão estrita, que todas as funções recursivas primitivas são Turing-computáveis. Entretanto, o contrário não vale, isto é, toda função recursiva primitiva é Turing-computável, mas nem toda função Turing-computável é recursiva primitiva. Dessa forma, provamos, através de uma generalização do método da diagonal, que existe ao menos uma função Turing-computável que não é recursiva primitiva. Portanto, se toda função recursiva primitiva é Turing-computável, mas nem toda função Turing-computável é recursiva primitiva, a classe das funções recursivas primitivas constitui uma subclasse das funções Turing-computáveis. Como consequência desse resultado, mostramos também, através do método da diagonal, que embora a função universal para a classe das funções recursivas primitivas enumera e computa todas as funções recursivas primitivas n-árias, ela não é recursiva primitiva.

Embora estas questões já tenham sido por demais discutidas, a abordagem e os resultados aos quais chegamos aqui foram apresentados diferentemente das formas como são expostos nos manuais pesquisados na literatura. A análise que fizemos das funções recursivas primitivas possibilita também uma melhor compreensão da classe das funções algorítmicas, no sentido de que esta classe possui um algoritmo reconhecedor e uma função universal, características que a

classe das funções algorítmicas não possui. De todas as maneiras, temos que nossa análise favorece a uma compreensão mais acurada da classe das funções algorítmicas.

REFERÊNCIAS

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

CICHON, Adan. A short proof of two recently discovered independence resultats using recursion theoretic methods. *Proceedings of the American Mathematical Society*, vol. 87, 1983, p. 704-706.

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

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

DIAS, Matias Francisco. On Elementary Arithmetic. *Journal of Symbolic Logic*, vol. 59, 1994, p. 716-717.

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

DIAS, Matias Francisco; WEBER, Leonardo. *Teoria da recursão*. São Paulo: Ed. UNESP, 2010.

EPSTEIN, R. L.; CARNIELLI, W. A.. *Computability*: Computable Functions, Logic, and the Foundations of Mathematics. Wadsworth & Brooks/Cole Advanced Books & Software: Pacific Grove, California, 1989.

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

HINMAN, Peter G.. Recursion-Theoretic Hierarchies. *Perspectives in Mathematical Logic*, Vol. 9. Berlin: Springer-Verlag, 1978.

JAPARIDZE, G. The Logic of the Arithmetical Hierarchy. *Annals of Pure and Applied Logic*, vol. 66, 1994, p. 89-112.

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

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

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

NIES, André. *Computability and Randomness*. Oxford University Press, 2009.

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

POST, Emil L.. Recursively Enumerable Sets of Positive Integers and Their Decision Problems. *Bulletin of the American Mathematical Society*, Vol. 50, 1944, p. 284-316. Reprinted in E. L. Post, *Solvability, Provability, Definability*: The Collected Works of Emil L. Post. p. 461-494.

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

SIPSER, Michael. *Introdução à teoria da computação*. 2. ed. São Paulo: Cengage Learning, 2012. Tradução de: Ruy José Guerra Barreto de Queiroz.

SOARE, Robert. Formalism and intuition in computability. *Philosophical Transactions Of The Royal Society Of London A: Mathematical, Physical and Engineering Sciences*. London, vol. 370, p. 3277-3304. 18 jun. 2012. Disponível em: <http://rsta.royalsocietypublishing.org/content/370/1971/3277>. Acesso em: 09 dez. 2015.

SOARE, Robert. *Recursively Enumerable Sets and Degrees*: A Study of Computable Functions and Computably Generated Sets. Berlin, Heidelberg, New York: Springer-Verlag, 1987.

TURING, A. M. Collected Works: *Mathematical Logic* (R. O. Gandy and C. E. M. Yates, Editors). Elsevier, Amsterdam, New York, Oxford, Tokyo, 2001.

_____. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, vol. 42, 1936. p. 230-265.

WOLF, Michael M.. *Lecture on Undecidability*. 2012. Disponível em: <http://www-m5.ma.tum.de/foswiki/pub/M5/Allgemeines/MA5116_2012S/lecture.pdf>. Acesso em: 15 jun. 2015.