UNIVERSIDADE DE SÃO PAULO
FACULDADE DE FILOSOFIA, LETRAS E CIÊNCIAS HUMANAS

ENTREGA DO EXEMPLAR CORRIGIDO DA DISSERTAÇÃO/TESE

Termo de Anuência do (a) orientador (a)

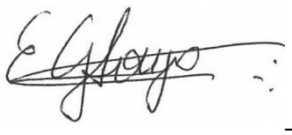
Nome do (a) aluno (a): Euclides Torres Ometto Stolf

Data da defesa: _08/09/2022

Nome do Prof. (a) orientador (a): Edelcio Gonçalves de Souza

Nos termos da legislação vigente, declaro **ESTAR CIENTE** do conteúdo deste **EXEMPLAR CORRIGIDO** elaborado em atenção às sugestões dos membros da comissão Julgadora na sessão de defesa do trabalho, manifestando-me **plenamente favorável** ao seu encaminhamento ao Sistema Janus e publicação no **Portal Digital de Teses da USP**.

São Paulo, _10/11/2022



_____ _____

(Assinatura do (a) orientador (a)

Objetos Imitadores:

teoremas limitativos pensados a partir da computação

Candidato: Euclides Torres Ometto Stolf

Orientador: Edelcio Gonçalves de Souza

Dissertação de Mestrado
Versão Corrigida

Departamento de Filosofia
Faculdade de Filosofia Letras e Ciências Humanas
Universidade de São Paulo
São Paulo
2022

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

Catalogação na Publicação
Serviço de Biblioteca e Documentação
Faculdade de Filosofia, Letras e Ciências Humanas da Universidade de São Paulo

S875o

Stolf, Euclides Torres Ometto
Objetos imitadores: teoremas limitativos pensados a partir da computação / Euclides Torres Ometto Stolf; orientador Edelcio Gonçalvez de Souza - São Paulo, 2022.
146 f.

Dissertação (Mestrado)- Faculdade de Filosofia, Letras e Ciências Humanas da Universidade de São Paulo. Departamento de Filosofia. Área de concentração: Filosofia.

1. incompletude. 2. ponto fixo. 3. computação teórica. 4. lógica. I. de Souza, Edelcio Gonçalvez, orient. II. Título.

Para meus pais

Agradecimentos

Um dia, quando comentei com um amigo que estava voltando a estudar na filosofia, ele sentenciou numa palavra: Aufhebung. A palavra ficou em minha mente, e ressoou em vários sentidos no período em que escrevi esta dissertação. Não só porque foi um reencontro com a filosofia, agora com um olhar da matemática, mas no sentido de revelar muitas ideias e discussões que apenas puderam acontecer devido a muitas pessoas brilhantes com quem convivi durante este tempo, e que contribuíram direta ou indiretamente para este trabalho. E por isto devo todo meu agradecimento a elas.

Em primeiro lugar, agradeço ao meu orientador, Edelcio, com quem, desde a primeira aula, fiquei assombrado com a didática e ao mesmo tempo profundidade. Agradeço em especial por me guiar nesta dissertação, e poder propiciar um ambiente tão acolhedor. Agradeço ao Daniel Nagase, a quem devo muito deste trabalho, e agradeço sobretudo às conversas que sempre me ajudaram muito. Agradeço aos colegas e amigos, que sempre me instigaram com as discussões, e ensinamentos, seja nas aulas, nos grupos de estudos, ou fora deles: ao Levi, Fernanda, Marco Gaiarsa, Thiago Alexandre, Pedro Falcão, Luíza Ramos, Rodolfo, Matheus, Caio, André, James, João, Luiz, Marcos, Henrique, Felipe, Vanessa.

Agradeço a todos professores com quem tive o prazer de assistir aula durante este período. Agradeço ao departamento de filosofia da USP como um todo, aos funcionários e alunos. Agradeço à CAPES, pela bolsa de estudos.

Agradeço por fim a minha família, minha mãe Regina e meu pai Hamilton, ao suporte, compreensão e paciência que sempre me deram.

Resumo

Temos como objetivo analisar fenômenos de autorreferência na lógica e na computação, estabelecendo resultados limitantes para ambas as áreas. Em especial damos uma exposição unificada para o teorema da incompletude, tanto na sua formulação construída por Gödel, quanto na versão demonstrada por Chaitin.

Abstract

Our goal is to analyze self-referential phenomena in logic and computer science, establishing limiting results for both areas. In particular, we give a unified exposition of the incompleteness theorem, both in Gödel's formulation and Chaitin's proof.

Sumário

Prefácio

Este trabalho tem como público alvo pessoas já familiarizadas tanto com lógica (possivelmente já com algum contato com os teoremas da incompletude) quanto com algum manejo de ferramentas matemáticas e de programação. Adotamos uma postura que difere de alguns textos clássicos da área que tomam como modelo computacional principal as funções recursivas ou a própria aritmética. Procuramos nos aproximar dos modelos computacionais empregados na prática da programação, em especial queremos nos aproximar de um modelo básico de linguagem de programação (imperativa) com ferramentas com que um programador está acostumado: loops e condicionais. Porém, isto não quer dizer que o texto é voltado especificamente para pessoas da área, nem que teremos alguma linguagem de programação preferida como modelo, mas queremos com isto indicar que muitas ideias que aparecem cotidianamente na área tiveram um percurso histórico que surgem destas questões teóricas. O fato de tais ideias se mostrarem úteis para esta área e o fato de estas ideias conservarem um núcleo comum ao longo dos anos talvez seja indício de que é possível ter um valor teórico – apesar da programação ser um campo de atualizações incessantes e apesar de terem manifestação e nomenclaturas muito diferente a depender da linguagem de programação.

Por isto temos como objetivo nos três primeiro capítulos abordar de forma gradual uma diversidade de modelos computacionais, partindo de uma visão simplificada da noção de número visto da perspectiva funcional (exposto no capítulo introdutório, na seção sobre a ambivalência computacional); depois fundamentaremos as ideias sobre máquinas de Turing, no capítulo 2 (Perto da máquina) ; depois apresentaremos a ideia de como passar de tal modelo computacional para o modelo das máquinas while-codificadores, no capítulo 3 (Longe da máquina), definição que esperamos tenha conexão com a ideia de loop como é apresentado no contexto de programação. Terminamos este último capítulo apresentando a última etapa de abstração: o teorema da universalidade.

Depois mostraremos no capítulo 4 (Máquinas como números e números como máquinas) alguns resultados centrais da computabilidade, focando na ideia de ponto-fixo. No capítulo 5 mostramos a incompletude a partir do teorema do ponto-fixo da teoria da computabilidade, evitando porém os detalhes iniciais de como codificar teoremas dentro de um modelo computacional e de como expressar as construções deste modelo computacional através do sistema formal. No capítulo 6 abordaremos a incompletude de Chaitin. Na conclusão indicamos

rapidamente como é possível comparar a incompletude demonstrada por Chaitin com a incompletude demonstrada com métodos gödelianos.

Nos apêndices colocamos definições de apoio à leitura e um exemplo de como simular máquinas de Turing em uma linguagem de programação específica.

Esquematicamente, percorremos os objetos computacionais da seguinte maneira:

Números como funções (Capítulo 1)

↓

Máquinas de Turing (Capítulo 2)

↓

Máquinas while-codificadoras (Capítulo 3)

↓

Máquinas universais (Capítulo 4)

↓

Aritmética (Capítulo 5)

↓

Máquinas universais aditivamente ótimas (Capítulo 6)

Capítulo 1

Introdução

Neste capítulo apresentaremos algumas motivações das ideias que serão desenvolvidas posteriormente no texto. Nas primeiras seções trataremos do conceito de imitação e computabilidade de um ponto de vista filosófico. Trataremos na seção "Ambivalência computacional" sobre a operação de *curring* adotando um ponto de vista funcional adaptado do lambda calculus. Depois, falaremos como é possível pensar o teorema do ponto-fixo em abstrato. Terminaremos aplicando o teorema do ponto-fixo em um contexto simplificado.

1.1 Imitação

Imitação é um conceito que pressupõe, entre outras, estas camadas: o imitado e a imitação. Quando vamos tentar definir algo na matemática e na lógica separamos dois tipos de níveis de texto: a linguagem e a metalinguagem. A metalinguagem seria uma espécie de linguagem inicial, e a linguagem aquela que contém os símbolos específicos da teoria que queremos trabalhar. Para explicar a linguagem usamos a metalinguagem. Porém, algo estranho ocorre ao efetuarmos esta separação, para explicar alguma operação da linguagem muitas vezes precisamos ter em germe este conceito na metalinguagem.

Por exemplo, quando queremos explicar o que é uma prova formalmente, podemos explicar isto na metalinguagem utilizando a noção de conjunto, caso contrário não conseguiríamos explicar o que é o conjunto dos teoremas de uma teoria. Mas o que é conjunto? Poderíamos pensar que este conceito é dado por uma teoria, mas uma teoria pressupõe o conceito de prova. Assim poderíamos pensar em uma metametalinguagem da metalinguagem e assim por diante, poderíamos pensar em um metametametalinguagem. Há um jogo de imitação operando neste exemplo, a linguagem imitando determinados aspectos da metalinguagem, como a noção de conjunto, e vice-versa.

A situação pareceria circular caso não soubéssemos que de fato nós conseguimos nos comunicar. Ou seja, o início da linguagem existe. De alguma forma as

crianças conseguem – talvez por imitação – adquirir linguagem. Porém não sabemos como este fato da aquisição da linguagem ocorre e tal questão permanece dentro do campo da psicologia do desenvolvimento infantil. Não adentraremos tais questões, e para prosseguir a discussão diremos que existe uma prática inicial.

A situação parece sob controle se assumirmos que de fato existe algo chamado metalinguagem, onde há determinados conceitos primordiais sobre os quais todos estão de acordo. Ou seja, se assumirmos que há várias camadas de linguagem na matemática, uma mais primordial, e outra derivada, não parece que teríamos problemas. Porém, isto é uma ilusão. O jogo de imitação ocorre dentro da própria linguagem.

Para ver claramente isto, basta pensarmos que símbolos matemáticos como 0 e 1 podem ser utilizados para codificar letras. E assim, podemos formar sequências de 0 e 1 para representar sentenças e fórmulas da própria linguagem. Uma teoria suficientemente complexa poderia utilizar isto para se referir a si mesma. Por exemplo, a teoria poderia se referir a quais fórmulas da teoria são demonstráveis ou não são demonstráveis. E este é o germe do teorema da incompletude demonstrado por Gödel. Em resumo, ele nos dá uma forma de codificar na teoria uma sentença que diz: "esta sentença não é demonstrável". Este é um dos pontos principais deste trabalho, cujos ingredientes principais vamos abordar.

Por hora, vamos pensar em outro campo em que a imitação opera. Na computação ocorre um fenômeno interessante, existem programas[1] que reproduzem o seu próprio código. Tais programas são chamados *quines*, é o que iremos ver na proposição 11. Eles imprimem o próprio código. Em analogia com seres vivos, isto equivale a dizer que existem programas que se reproduzem como a procriação de seres vivos. Mas mais do que isso, podemos criar programas que além de se reproduzirem, evoluem, no sentido de, além de produzir seu próprio código, modificam alguma coisa do seu próprio código para produzir um novo programa. Ou seja, estes programas podem modificar a si mesmos de acordo com alguma regra.

Porém, da mesma forma como na matemática, isto gera alguns problemas. Em analogia com a noção de prova na matemática, no âmbito da computação temos o o problema da parada de um programa. Tanto a parada de um programa quanto uma prova de um teorema determinam se uma sequência de operações chega ao fim ou não. E da mesma maneira como na matemática, não é trivial determinar se um programa pára. Não podemos construir um programa para determinar de antemão se outro programa irá parar ou continuará operando para sempre. Pois, da mesma maneira de algumas teorias matemáticas, podemos construir um código de um programa que tem sua parada condicionada pela sua não parada. Analisaremos tal problema mais a frente, chamado problema da parada, no teorema 8.

De maneira mais abstrata, um computador (no sentido abstrato) é uma máquina programável e deste modo ele pode imitar o que qualquer outro computa-

[1] Veremos mais a frente a definição formal de programa.

dor faz, inclusive si mesmo. Podemos assim imitar o funcionamento do próprio computador, e isto no próprio computador. Podemos criar um programa que faz tudo o que o próprio computador faz, e operarmos este programa nele mesmo. Esta possibilidade é o que permite criarmos linguagens de programação que independem da forma como um computador é construído. Assim computadores são universais, no sentido de que qualquer programa de qualquer outro computador pode ser imitado dentro de qualquer outro computador. Computadores são objetos imitadores, assim como algumas teorias matemáticas o são. É isto que está por trás da construção que iremos fazer no teorema da universalidade 5.

Pensando deste modo, podemos fazer o seguinte questionamento. E se, ao invés de termos um computador complexo, com mais componentes, criarmos um computador bem simples, com poucos componentes. Tudo que este computador primeiro realiza, também o computador simples realizará. É verdade que a questão do tempo pode entrar, mas vamos deixar isto de lado, não nos importamos com quanto tempo cada computador realiza a operação pretendida. Deste ponto de vista, podemos dizer que o computador complexo possui componentes a mais que não são necessários; em resumo, que ele é redundante. Tal observação, e a possibilidade de podermos tirar redundâncias sem comprometer a imitação, fez surgir o campo de estudo chamado complexidade de Kolmogorov[2], em homenagem a este que foi um dos pioneiros da área.

Em especial, Chaitin mostrou utilizando o paradoxo de Berry que a noção de complexidade pode ser usada para criar uma sentença paradoxal que diz:

> O_1 menor_2 número_3 que_4 não_5 pode_6 ser_7 definido_8 com_9 menos_{10} de_{11} vinte_{12} palavras_{13}.

Acabamos de defini-lo com menos de vinte palavras.

Veremos este problema no teorema 17. A novidade de tal sentença é que ela não fala de si mesma, como outras sentenças paradoxais. A pergunta que fica é: será que em algum momento a autorreferência é usada neste tipo de paradoxo ou em construções teóricas baseadas nele?

Ora, a situação que temos é a seguinte: a imitação é um fenômeno geral que surge em muitas linguagens. Algumas teorias ou objetos possuem uma complexidade tamanha que podem utilizar isso para criar sentenças paradoxais que falam de si mesmas. O propósito desta monografia é tentar explicitar como esta ideia surge tanto no contexto da incompletude, tanto na forma como foi demonstrada tanto originalmente por Gödel, quanto demonstrado por Chaitin.

1.2 Computabilidade

A computabilidade é uma área que estuda problemas teóricos de funções computáveis. Mas o que significar dizer que algo é computável? É interessante fazer

[2]Tal campo de estudo se iniciou, por um lado, como uma tentativa de formalizar a teoria da probabilidade, além de outras perguntas teóricas, tendo sido construído simultaneamente por vários pesquisadores: Chaitin, Kolmogorov, Martin-Löf.

análise filosófica neste ponto, expondo algumas teses sobre a conexão que há entre a ideia de computabilidade e a ideia de determinismo.

No sentido contemporâneo, dizer que um processo é determinístico significa dizer que o processo é determinado inteiramente por suas causas. Ou seja, que há ausência de possibilidades de escolha. Neste sentido dizemos que um agente é determinístico se ele não tem o poder de tomar decisões. Embora este nomenclatura seja tardia, para os filósofos do racionalismo havia um dilema: seria a natureza determinística? Quando analisamos as opiniões dos filósofos racionalistas neste quesito, podemos ver diversas variações deste tema: cartesianos, que argumentavam que a distinção corpo e alma, dito de modo simplificado, representava o conflito entre um mundo determinístico e um não determinístico, cuja complexidade de interação resultava na complexidade das paixões da alma e na possibilidade de interferir no mundo com livre-arbítrio; espinosistas que acreditavam que a mente, assim como o corpo, operava seguindo causas necessárias que ela mesma não podia acessar, dada sua limitação, e portanto postulando que o livre-arbítrio seria uma ilusão; e outros, que também tomavam este questionamento como um dos temas centrais da filosofia.[3]

É possível localizar no racionalismo cartesiano, indiretamente, a ideia de tomar a certeza da aritmética como um guia para regras mais gerais de condução do conhecimento. Mais tarde, no século XX, se tentou dar um caráter mais radical a esta afirmação, mostrando que, no campo específico da lógica, não somente era possível conservar as verdades de uma disciplina na outra, mas as regras de dedução mesmas poderiam ser aritmetizadas.

Por outro lado, a história da matemática como um todo apontava para esta direção, desde a criação de ábacos a humanidade procurava por um método sistematizado de resolução de problemas. O ponto é que todo este contexto histórico e matemático lentamente levou o século XX a criar constructos mentais que conseguiam seguir regras explícitas para o problema da calculabilidade. Manteve-se a ideia de que os problemas complicados devem ser divididos em pedaços cada vez menores. E, ao mesmo tempo, levou os matemáticos a se perguntarem quais problemas estes constructos mentais poderiam resolver.

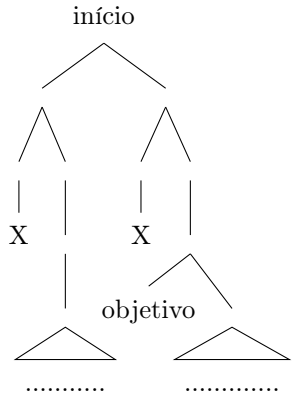
Coincidência ou não, contemporaneamente a teoria computacional moderna utilizou conceitos semelhantes a estes, porém interpretou de forma diferente esta dicotomia entre determinismo e não-determinismo. Determinismo e não-determinismo na computação contemporânea não estão mais associados à noção

[3] Não à toa, a questão do método era importante para Descartes. Regras que permitiriam bem conduzir o raciocínio mostrariam que o homem, apesar de limitado, possuiria em potência a mesma capacidade de, pelo menos no âmbito da matemática, conseguir resolver qualquer problema que fosse passível de solução. Ao mesmo tempo, tentava-se evitar qualquer tipo de discordância com a fé cristã, pois neste sentido o homem estaria em posição semelhante a Deus: enquanto Deus conseguiria acessar imediatamente a verdade de tudo, o homem deveria chegar até lá através da análise, da divisão do problema em problemas menores, e assim por diante. Assim, Descartes dizia:

> Pois, enfim, o método que ensina a seguir a verdadeira ordem e a enumerar exatamente todas as circunstâncias do que se procura contém tudo o que dá certeza às regras da aritmética. [**p.25**; Des96]

de escolha. Vamos ilustrar isto em um exemplo: o de que chamaremos *agente determinístico*, inspirado no conceito de Pushdown Automata.

Suponha que um agente determinístico deve andar um caminho, partir do início e achar o objetivo. Este caminho possui bifurcações, como as descritas na árvore abaixo, a título de exemplo. Em cada bifurcação ele pode tomar ou o caminho da direita ou (exclusivo) o da esquerda (ou seja, não pode tomar ambos os caminhos ao mesmo tempo). (Para facilitar a compreensão, sempre representamos o caminho da direita pensando da perspectiva do leitor, ou seja o caminho da direita seria aquele que está à direita na página. Respectivamente para representar o caminho da esquerda.) Há caminhos que não terminam, que se dividem em infinitas bifurcações, na árvore abaixo isto é simbolizado pelos triângulos com base pontilhada. E há caminhos fechados, simbolizados pelo X.



Nós devemos dar comandos para este agente determinístico. Os comandos se resumem a escolher a cada bifurcação o caminho da direita (D) ou o caminho da esquerda (E). Por exemplo, poderíamos dar o seguinte comando: sempre tome o caminho da direita. Nesta árvore específica acima isto significaria que o agente percorreria um caminho infinito e nunca iria parar. Por outro lado, caso disséssemos o comando sempre tome o caminho da esquerda, isto significaria que ele iria parar em caminho fechado (X).

Não é difícil ver que achar o objetivo neste caso, para este tipo de agente, é questão de pura sorte.

Porém, neste pequeno jogo, já podemos retirar algumas ideias. É interessante analisar isto da ótica deste agente determinista. Para ele, não há escolhas (é aqui que a associação com o determinismo do passado falha[4]). Ele não conhece as regras do jogo. Somos nós, que damos as ordens para ele, que sabemos que há escolhas a tomar a cada bifurcação. Esta separação entre aquele que dá as ordens (nós) e o agente que as opera é crucial. Nós sabemos que há bifurcações, e sabemos que estas bifurcações podem se repetir indefinidamente.

[4]Ou poderíamos dizer, falha até em certo nível. Há talvez um resquício da divisão entre alma e corpo em operação, aqui. O programador seria uma espécie de alma, o corpo seria a máquina. Mas não seguiremos tal análise, que envolveria uma análise mais fina dos problemas envolvidos.

Os comandos que damos ao agente estariam em analogia com um programa, o agente estariam em analogia com uma máquina específica, e nós estaríamos em uma analogia com um programador. Tal separação de níveis facilita a compreensão de que o programador estaria em posse do conhecimento das tomadas de decisão a se fazer. O programador teria uma visão global das possibilidades. Apesar de não saber qual árvore específica estaria trabalhando, ele teria uma visão "onisciente" de como todas estas árvores funcionam.

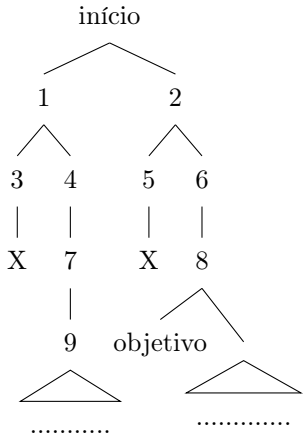
Agora introduziremos a descrição do funcionamento do agente não determinístico. Um agente não determinístico é aquele que percorre todos os caminhos originado pelas bifurcações ao mesmo tempo. Ele percorreria todos os caminhos em paralelo. Assim, por exemplo, um agente não determinístico percorreria todas as subárvores (Em qualquer árvore, e a cada nível, de cima para baixo, há uma subárvore associada), como se multiplicasse a cada intersecção. E assim, um agente não-determinístico acharia o objetivo sempre, desde que sempre haja um ramo com a palavra objetivo.[5]

Há uma diferença grande diferença entre agentes determinísticos e não-determinísticos neste caso. Ou seja, o poder computacional dos constructos não determinísticos é muito maior: é sempre possível achar o objetivo. Já no caso dos agentes determinísticos, não necessariamente.

Mas a situação muda de figura quando trabalhamos com um tipo de agente que aceita instruções mais sofisticadas. Suponhamos que agora um agente determinístico, além de receber a instrução de ir para esquerda ou para direita, também pode voltar qualquer caminho tomado, e em acréscimo também pode marcar cada nó da árvore. Por exemplo, poderíamos dizer à máquina que ela deve seguir sempre à esquerda, e se encontrar algum caminho fechado deve voltar para a última bifurcação e continuar o mesmo procedimento. Na árvore em questão, estas instruções novamente fariam o agente não chegar ao objetivo, pois o agente cairia em uma árvore infinita.

Porém, este novo agente determinístico é de configuração muito mais complexa que o agente determinístico inicial. É possível dar uma instrução que, ao contrário da anterior, faça com que ele encontre seu objetivo com toda certeza (dado que toda árvore tem um objetivo a ser alcançado). Em resumo tal instrução diz ao agente que ele deve ordenar cada nó da árvore (digamos da esquerda para direita). Na árvore em específica isto equivale a fazer a seguinte ordenação:

[5] Apesar da associação com a concepção de determinismo no sentido filosófico também falhar aqui, talvez seja possível dizer que tal possibilidade de tomar vários caminhos ao mesmo tempo seja um compartilhamento do ponto de vista do programador com o agente.



Uma possível instrução que resultaria em tal ordenamento (usamos indução):

- Na primeira bifurcação, vá a esquerda e marque o símbolo 1, em seguida volte ao início e vá para direita e marque 2. Caso encontre o objetivo neste primeiro nível, pare.
- Para cada nível da árvore já marcada, percorra cada nó em ordem crescente, de modo que, se $n-1$ é o último símbolo marcado:
  - caso haja bifurcação vá à esquerda e marque n, volte, e vá a direita e marque $n+1$, volte;
  - caso não haja bifurcação siga em frente, marque n e volte;
  - caso encontre um caminho fechado (X), volte;
  - caso encontre o objetivo, pare.

Há várias formas de realizar alguma ordenação nesta árvore binária de forma sistemática. O ponto importante é ver que, com este novo agente mais sofisticado, a diferença entre agente determinístico e não determinístico desaparece para os efeitos de encontrar o objetivo. Pois, um agente não-determinístico deste modelo mais sofisticado percorreria simultaneamente todos os caminhos. Obviamente encontraria o objetivo, se ele existir. Mas as vantagens a mais que obtemos ao passar para esta versão não determinística não implicariam em uma melhora daquilo que o agente determinístico já fazia, para efeitos de encontrar o nosso objetivo. A única melhora seria apena na questão de tempo. [6]

[6] Tempo aqui se refere a quantos passos o agente deve dar até achar o objetivo. Para efeitos práticos de computação, tal pergunta adquire importância primeira. Pois, em muitos casos, achar um algoritmo para fazer alguma tarefa não é suficiente para resolver o problema, não temos a nossa disposição tempo infinito. Problemas complicados tendem a ter tempo de resolução muito longos. Por exemplo, o xadrez é um jogo que possui uma árvore de possibilidades de jogadas que cresce exponencialmente. Achar um algoritmo para a melhor jogada requer tempo exponencial relativo ao número de jogadas que se quer prever. Tal

A discussão que fizemos aqui é uma ilustração do conceito formal de um *Pushdown Automaton*, conceito equivalente a nosso agente determinístico inicial; e por outro lado, *máquinas de Turing*, conceito formal que faz analogia ao agente determinístico posterior e mais avançado. Apresentaremos uma definição formal deste conceito na definição 5. O ponto desta comparação é salientar a diferença de nível que há de um conceito para outro, ou seja: máquinas de Turing não determinísticas possuem o mesmo poder computacional que máquinas de Turing determinísticas, e por outro lado Pushdown Automata determinísticos **não** possuem o mesmo poder computacional que Pushdown Automata não determinísticos. Esta diferença de nível mostra que máquinas de Turing são mais complexos que estes autômatos. Dito de outro modo, máquinas de Turing absorvem a complexidade envolvida no conceito de não determinismo. E isso é um indicador de que máquinas de Turing conseguem abarcar não só máquinas do tipo não-determinístico, mas todos os fenômenos computacionais possíveis. Ou seja, é um indicador de que podemos tomar a própria definição de máquina de Turing como definição do que seria intuitivamente calculável.

Tal último passo porém é apenas um indício, uma evidência, não é um argumento que mostra absolutamente que a definição de máquina de Turing absorve todos os possíveis fenômenos computacionais. Isto porque a definição do que seria intuitivamente calculável, ou dito de outro modo, *efetivamente computável*, é uma noção como o nome diz, não formal. Ela designa os casos paradigmáticos sobre o que seria um método para decidir se um problema pode ser calculado com lápis e papel (ou qualquer outro meio que possa ser reduzido a este).

Em uma formulação contemporânea dizemos que um *método de decisão* para um conjunto A é um método que mostra, para algum objeto a, se é possível decidir em um número finito de passos se a está em A ou não.[7] Similarmente, existe um método de decisão para uma função $f : A \to B$ se é possível decidir em um número finito de passos se dado um a, podemos obter $f(a)$. Assim, um conjunto ou uma função seriam *efetivamente computáveis* se exite um método de decisão para eles.

---

análise é inviável para qualquer tipo de computador prático. Do ponto de vista abstrato, abstraindo o tempo, é possível encontrar tal jogada. Ou seja o problema da decisão para o xadrez é decidível. Mas isto, na prática, não tem importância, pois seu tempo de resolução é exponencial.

O problema da decisão para se determinar se uma fórmula da lógica proposicional é satisfatível ou não foi demonstrado ser de tempo polinomial para um agente não-determinístico (mais especificamente para uma máquina de Turing não determinística). A grande pergunta é se é possível encontrar um algoritmo também polinomial deste problema para agentes determinísticos. Isto é equivalente a afirmar que $P = NP$, ou seja, dito em outros termos, que qualquer problema verificado em tempo polinomial pode também ser resolvido em tempo polinomial.

Interessante notar que computadores hoje em dia possuem frequência por volta de 2Ghz (cada núcleo). Isto é equivalente a $2 \times 10^9 \approx 10^{10}$ operações por segundo. Um problema de ordem de n^{10}, para uma entrada de $n = 10$, seria realizado por volta de um segundo. Para $n = 100$, seria realizado por volta de $\frac{(10^2)^{10}}{10^{10}} = 10^{10}$ segundos. Isto é aproximadamente 300 anos. Inviável. Apesar de problemas como $P = NP$ serem importantes, muitos problemas ainda permaneceriam inviáveis em termos práticos caso a sua solução seja positiva.

[7] Aqui não enunciamos, mas está subentendido que A é subconjunto de um conjunto universo U e a está neste conjunto universo.

Claro, estas definições não resolvem o problema. Pois estamos jogando o problema para o que seria um *método*. E qualquer outra tentativa de definir melhor estes conceitos parece ser circular. Por isso, é mais fácil dar exemplos do que entendemos por este conceito.

Para ilustrar, voltando para nossa ilustração inicial, não há um método de decisão que garanta que o nosso primeiro agente determinístico chegue ao objetivo. Por outro lado, para nosso segundo agente, há um método de decisão. É possível mostrar que há um método de decisão de problemas de árvores binárias semelhantes a este para máquinas de Turing também. Porém, não há um método de decisão que garanta que uma máquina de Turing compute se outra máquina de Turing pára ou não, como falamos, este é o famoso problema da parada. Outro problema importante que se mostrou indecidível por máquinas de Turing, é o problema de decisão de determinar se uma fórmula da lógica de primeira ordem é teorema ou não: o problema da decidibilidade da lógica da primeira ordem. Tal problema também não é decidível por máquinas de Turing.

Um outro ponto importante a ser abordado sobre a noção intuitiva de computabilidade efetiva é que ela deve ser formulada, para ser completa, no ambiente de funções parciais. Vamos dar um exemplo para ilustrar. Funções parciais são funções não definíveis em alguns elementos do domínio. Um exemplo é a divisão. Não podemos aplicar a função divisão quando o dividendo for zero. Similarmente, funções que surgem naturalmente de modelos computacionais não estão definidos em pontos que funções associadas dão erro. Por exemplo, se dermos uma árvore sem nenhum objetivo para nosso segundo agente determinístico (ou não determinístico), nunca encontraremos o objetivo. Dizemos que a função associada não está definida para este input.

1.3 Tese de Church

O modelo de máquinas de Turing não foi nem o primeiro nem será o último modelo computacional criado. Por exemplo, havia antes dele o lambda calculus criado por Church 1933. E também antes dele havia uma discussão sobre funções recursivas, primeiramente em sua versão inicial dada por Dedekind, e posteriormente um aperfeiçoamento para sua versão equivalente à máquina de Turing (utilizando a μ-recursão). Todos estes modelos se mostraram equivalentes.

Vejamos um pequeno resumo histórico destas definições (ordenados por data de conceitualização da definição seguido da prova de equivalência com outras definições que basicamente ocorreu em 1936):

I (Dedekind, 1888) - Funções recursivas primitivas (nota: classe menor, não é equivalente às seguintes);

II (Herbrand, Gödel, 1931) - Funções Computáveis por sistema de equações, Herbrand-Gödel computáveis (equivalente à recursiva por Kleene);

III (Church, 1933) - Lambda Calculus (equivalente à recursiva por Church e Kleene);

IV (Kleene, 1936) - Funções recursivas

V (Turing, Post, 1936) - Funções computáveis por máquina de Turing-Post (equivalente à recursiva por Turing);

VI (Gödel, Tarski, 1936) - Funções fracamente representáveis em um sistema formal que contém aritmética (equivalente à recursiva por Church e Gödel)

Detalhes sobre a equivalência destas definições no contexto de funções recursivas (totais) podem ser conferidos em Odifreddi [Odi99] capítulo 1.

A pergunta que se segue imediatamente ao observar este percurso histórico é: por que consideramos máquinas de Turing como padrão neste texto em detrimentos de outros modelos computacionais?

Em primeiro lugar, máquinas de Turing abordam claramente a distinção entre determinismo e não determinismo de suas operações. É fácil mudar a definição de máquina de Turing para que ela se torne uma máquina de Turing não determinística. Como falamos, esta mudança não implica em uma mudança de poder computacional. Ambas máquinas calculam as mesmas coisas. Porém, para se pensar em complexidade temporal (não confundir com a complexidade de Kolmogorov, um dos tópicos que tratamos aqui), a facilidade de tal mudança se mostrou crucial. Assim, muitos adotam máquinas de Turing como padrão, pela facilidade de adaptação.

Em segundo lugar, máquinas de Turing em geral são fáceis de serem modificadas e continuarem computáveis. Há muitas mudanças que podem ser feitas como: trabalhar com mais de uma fita, adicionar contadores, adicionar símbolos, trabalhar com um modelo mais próximo do modelo RAM dos computadores usuais, trabalhar com entradas de um certo tipo.

Em terceiro lugar, máquinas de Turing não são imediatamente funções. Elas se "tornam" funções depois fazermos uma associação entre strings e números (e definirmos as entradas e saídas destas funções de um jeito usual). Este ponto é interessante especialmente porque esta visão é mais próxima da visão do programador imperativo (isto é aquele que trabalha com linguagens imperativas como C, $Python$, etc...). Neste ponto de vista, há um objeto que é o computador, há as funções que são objetos abstratos, existem as entradas dadas por algum usuário, e há ele, o programador, que pensa tudo isto em conjunto, para criar algo usável. Da mesma forma, aqui há as máquinas de Turing com seus programas escritos no papel, há as funções computáveis associadas a estas funções, existem as entradas que podemos colocar nas máquinas, e há nós que pensamos tudo isto em conjunto. Em modelos como o Lambda Calculus, em que tudo é função, não conseguimos separar claramente todos estes componentes. O componente maquínico se mistura com a própria definição de função, e se escrevem juntos quando as representamos no papel. Isto pode ser uma vantagem em muitos aspectos, pois nos permite pensar já desde o início que o computador nada mais é que uma função, ponto de vista que será abordado na próxima seção. O ponto de vista das máquinas de Turing nos permite separar melhor o ponto de vista do programador, do ponto de vista do constructo. E isto é importante para

lidar com questões de complexidade. Pois queremos medir os programas das máquinas de Turing, não medir as funções.

Uma curiosidade histórica também talvez seja interessante aqui com relação a intuitividade da definição de máquina de Turing. No resumo histórico acima, muitas provas de equivalência de definições de computabilidade foram pensadas por Kleene, Church e Turing. Gödel primeiramente não estava convencido da tese de Church, ou seja, que a noção de computabilidade que podemos dar intuitivamente aparentemente sempre se mostra equivalente a alguma destas definições. Gödel tinha dúvidas da universalidade, e mesmo inter-equivalência das definições. Ele tinha dúvidas até mesmo se a definição II, que hoje possui o seu nome, era definível a partir de outras. Foi apenas com a definição proposta por Turing que Gödel se mostrou convencido desta convergência de definições. Talvez podemos dizer que Gödel achou esta definição intuitiva suficiente para se convencer de uma tese consideravelmente importante.

É certo que as outras definições são também intuitivas em determinados sentidos, porém talvez seja interessante perceber a intuição destas definições no sentido de facilitarem ou não a nossa compreensão da tese de Church-Turing. O fato de uma definição ser mais intuitiva que outra obviamente não pode ser medida pelo fato de uma ou mais pessoas a acharem mais intuitiva, mas não deixa de ser curioso que esta pessoa seja Gödel, quem dedicou parte de seu trabalho a considerações sobre por exemplo Principia Mathematica de Russell e Whitehead, reconhecidamente um dos constructos mais crípticos da história da lógica do século vinte.

Há diversas razões para aceitar que tudo que pode ser efetivamente computado pode ser computado por algum dos modelo computacionais acima. Vamos enumeração estas razões:

1. Muitas mudanças na definição de máquinas de Turing, que a torne mais sofisticada, ou mesmo não determinística, têm como consequência uma máquina com poder equivalente à definição original.

2. Todos os modelos computacionais alternativos se mostraram equivalentes entre si.

3. A noção de efetivamente calculável não é definível; é um conceito intuitivo que reflete a percepção do homem sobre o que é computável.

Isto nos leva diretamente à tese de Church-Turing, que pode ser anunciada deste modo, seguindo Soare [Soa16] p.7:

Tese de Church-Turing. *Uma função é efetivamente calculável por um ser humano se, e somente se, pode ser computada por uma máquina de Turing. Dito de outro modo, toda a noção de algoritmo e função algorítmica informal pode ser captada pela definição formal de máquina de Turing e função parcial computável por máquina de Turing, e vice-versa.*

A tese possui um gosto metafísico especial de consequências extremamente filosóficas. Não exploraremos a fundo esta questão, pois requer um outro direcionamento do assunto. Porém é interessante salientar que obviamente a tese não

é para todos gostos filosóficos. Rogers[8] diz que esta tese não pode ser demonstrada. Deve ser aceita ou não com base em evidências empíricas, argumentos pragmáticos ou argumentos filosóficos. Não obstante, ela é praticamente um consenso no campo de estudos da computabilidade devido aos resultados clássicos que mostraram a equivalência da noção de computabilidade por máquina de Turing a noções mais diversas possíveis. Estas equivalências foram tomada como a maior evidência de que qualquer outro tipo de mecanismo abstrato possível também se tornaria equivalente ao modelo da máquina de Turing.

Para evitar os choques que podem eventualmente acontecer por alguém com tendências críticas mais ou menos radicais, sugerimos a divisão feita entre a tese no sentido abstrato dado a cima (em que os realistas podem considerar), e uma versão instrumental mais democrática que possibilita uma pacificação da situação e ao mesmo tempo também é imediatamente aceita pelo crítico. Esta parece ser a postura por exemplo de Odifreddi [9] [Odi99], p.103, e Rogers [Rog87] p.23. Dito com palavras do senso comum esta postura está em assumir que:

Tese de Church-Turing versão instrumental ou preguiçosa: *é possível reservar ao leitor o direito de perguntar se algum algoritmo ou função algorítmica tem uma contraparte formal na definição de computabilidade empregada, caso no qual aquele que está se utilizando da tese de Church-Turing tem o ônus da prova. Independente disso é possível assumir a tese por motivos metodológicos de exposição e praticidade do campo de estudo em questão.*

Ou seja, todo leitor mais ou menos indignado pode se debruçar sobre a literatura considerável deste campo de estudos. Rogers em seu livro *Theory of Recursive Functions and Effective Computability* descreve um procedimento de prova por tese de Church, que é basicamente considerar a tese de Church-Turing instrumental como um método de prova de funções claramente computáveis.

Intuitivamente, a tese de Church-Turing nos permite operar aproximadamente como se tivéssemos à nossa disposição uma equipe de programadores que fazem as tarefas tidas como acessórias para a prova de um teorema, como por exemplo fazer com que uma máquina de Turing consiga computar uma vasta gama de funções primitivas, manipular strings, etc...

Empregaremos esta tese no texto a seguir. Com isto podemos nos concentrar nos pontos em que achamos mais importantes para compreensão dos conceitos de universalidade, ponto-fixo e incompletude. Faremos uma exposição das definições, começando de conceitos maquínicos (mas não menos abstratos) como a ideia de Máquina de Turing, e passaremos por uma versão abstrata de uma linguagem de programação moderna (como C ou $Python$), e depois vamos em direção ao ponto de vista abstrato sobre as funções parciais computáveis, em que não importa o modelo computacional sobre a qual elas estão definidas. Esta passagem do que consideramos mais maquínico (definição de máquina de Turing) para o ponto de vista abstrato (funções parciais computáveis) possui muitas provas. Como temos a nossa disposição a tese de Church, podemos deixar muitas destas provas e nos concentrarmos na ideia geral que nos permite esta

[8] [Rog87], p.20

[9] Segundo Odifreddi, op. cit. p. 103, esta é uma divisão feita por Kreisel.

passagem. E esta passagem será o conceito de universalidade exposto no teorema 5 da existência de uma máquina universal; especificamente a possibilidade de simular máquinas de Turing a partir de outras máquinas de Turing.

A construção desta simulação pode ser feita de muitas maneiras. Porém, em muitos textos não é salientado como este procedimento ocorre diariamente nas linguagens de programação modernas. O conceito subjacente é o mesmo que está subjacente à ideia de compilador, ou seja, um programa escrito com a própria linguagem que traduz para o computador o que está sendo dito por uma linguagem de mais alto nível. Como tentativa de aproximar os conceitos, decidimos criar uma versão abstrata de uma linguagem de programação moderna, a que denominamos *máquina while codificadora*.[10] Baseamos este conceito em outros modelos computacionais, como a computação por flowchart exposta em [Odi99]. Também, este conceito possui analogia com o conceito de funções recursivas, modelo computacional normalmente empregado para provar a existência de uma máquina de Turing universal.

Acreditamos que este teorema pode ser ilustrado com o seguinte exercício: escolha a linguagem de programação de sua preferência e simule algum modelo computacional abstrato (máquinas de Turing, lambda calculus, etc...). Como qualquer linguagem de programação pode ser reduzida ao próprio modelo abstrato que se quer simular (esta passagem não é óbvia, aqui usamos a tese de Church), então estamos simulando o próprio modelo computacional no próprio modelo computacional.

Faremos um parêntesis aqui para defender o uso de linguagens de programação em uso atual para fazer uma analogia com estes resultados, em especial linguagens imperativas comuns em cursos de introdução à programação que possuem estruturas com laços de repetição do tipo *while*. Há um motivo porque linguagens de programação com estruturas de repetição sejam consideradas preferidas, e, por mais que as linguagens mudem, tais laços de repetição continuem sendo importantes. Tais formas de estruturar o programa são simples de definir, possuem uma estrutura linear[11], são fáceis de testar, e talvez o mais importante, permitem visualizar facilmente quando o loop irá rodar para sempre. Ou seja, o conflito entre parar e não parar, entre uma função parcial e total, é melhor administrado. Se compararmos com funções parciais recursivas, esta situação deve ser criada a partir da introdução da μ-recursão. Fechamos o parêntesis.

Apesar de escolhermos alguns modelos computacionais específicos, máquinas de Turing, com suas idiossincrasias específicas, a ideia de tal prova vai justamente na contramão de que estamos vinculados a este modelo computacional específico, pois a ideia da prova pode ser reproduzida para qualquer modelo

[10] Loops de tipo while restringem a repetição a um conjunto de instruções. Tal conjunto de instrução vai ser repetida somente se uma determinada condição for verdadeira. Caso a condição seja falsa, o programa sai do loop e continua a execução.

[11] Por exemplo o loop de tipo while é diferente de loops do tipo jump. Loops jumps são loops que não possuem uma estrutura linear, mais parecidos com os que encontramos em máquinas de Turing. Em resumo eles falam que é possível voltar a qualquer linha do programa e repetir o procedimento. Por referenciar qualquer parte do programa, este tipo de loop gera alguns problemas, porque qualquer parte do programa se torna passível de ser repetido, e não uma parte específica.

computacional. Isto reforça ainda mais a tese de Church como ponto de partida. Ou seja, as peculiaridades de definições formais podem ser esquecidas. Assim, quando falamos de funções parciais computáveis, queremos nos referir a qualquer modelo computacional que consiga fazer esta operação de imitação dentro de si mesmo, assim como no caso em que a descrição de uma curva em geometria analítica é invariante por mudança no sistema de coordenadas. E da mesma forma como na geometria analítica, há descrições mais simples e menos simples de tais curvas, a depender de onde colocamos o sistema de coordenada, assim há fenômenos mais simples e menos simples de serem descritos no universo da computação a depender do modelo computacional empregado. Desse modo, nossa intenção não é advogar o uso de uma linguagem computacional específica — máquinas de Turing ou qualquer outra — mas justamente, através do teorema da universalidade, perceber que a escolha é supérflua para os objetivos que temos em mente.

1.4 Finitismo e transfinitismo

Uma questão infatigável da filosofia é a pergunta do porquê nossas construções mentais correspondem a objetos da "realidade". No âmbito da filosofia da matemática tal questão pode ser bem ilustrada. Uma primeira aproximação a esse problema seria a observação de que os matemáticos criam teorias e modelos com certa coerência interna. Por outro lado o mundo físico também seria regido por leis coerentes. Assim, poderíamos fazer previsões sobre os eventos físicos através do aparato matemático e testar se tais previsões se confirmam. Ou seja, por falsificação de hipóteses, nós poderíamos aprimorar nossas previsões.

Porém, ao analisarmos simbolicamente os objetos matemáticos, nos deparamos com uma observação incontornável. Do ponto de vista dos símbolos, objetos como $\mathbb{R}$, o conjunto dos números reais, são apenas um símbolo no papel.[12] Não há nada contido neste símbolos, vistos isoladamente, que nos indique que se trata de um conjunto infinito. Como símbolo, não temos muito como argumentar contra a existência física de $\mathbb{R}$, eles têm uma existência determinada pela existência física do papel e das marcas de tinta. Porém, sua interpretação pretendida, de que se trata de um conjunto infinito, tem uma existência muito mais difícil de ser aceita do ponto de vista físico.

A situação é diferente quando analisamos conjuntos de cardinalidade finita. Por exemplo, supondo a definição de Von Neumann, o conjunto 3, além de ser um símbolo no papel, tem uma interpretação pretendida que também pode ser escrita com símbolos no papel $\{0, 1, 2\}$. Ou seja, deste ponto de vista conjuntos finitos parecem ter existência garantida, tanto do ponto de vista de seu símbolo, quanto de sua interpretação pretendida.

[12] Falamos de símbolo no papel por simplicidade, qualquer meio físico de representação, digital, sonoro ou de outro tipo, também poderia ser considerado. Em geral qualquer meio físico que grava informação e que pode ser reduzido a representação como a que temos em uma folha de papel e marcas de tinta.

Porém, vemos que símbolos como $\mathbb{R}$ são muito úteis para descrever o funcionamento de objetos físicos. Pensemos nas técnicas de otimização de funções diferenciáveis, resultados que são imensamente úteis para achar máximos e mínimos de funções. Aplicações como achar o ponto em que um carro atinge maior velocidade, otimização logística, otimização de estratégias de jogos, etc...

O problema da decidibilidade de sistemas formais, não por acaso, se concentra no uso dos quantificadores. A parte proposicional da lógica de primeira ordem é decidível, ou seja, para analisar se proposições complexas são ou não verdadeiras usamos a tabela de verdade ou qualquer outro método. Assim, podemos determinar rapidamente quando sentenças como $p(a) \vee q(a)$ ou $p(a) \wedge q(a)$ são verdadeiras ou falsas, dado que sabemos se $p(a)$ e $q(a)$ são verdadeiras ou falsas. Porém, como determinar se uma "sentença infinita" da seguinte forma é verdadeira ou falsa:

$$p(1) \vee p(2) \vee ... \vee p(n) \vee ...?$$

Suponha que verificamos que todas sentenças até a sentença $p(n)$ são falsas, nada nos garante que a $n+1$-ésima sentença não será verdadeira. É exatamente a mesma coisa que está acontecendo com sentenças do tipo $\exists x p(x)$. Se um computador achar apenas uma sentença $p(a)$ verdadeira, $\exists x p(x)$ será verdadeira. Mas se, neste processo, apenas verificar que todas as sentenças até a sentença $p(n)$ são falsas, nunca terá garantia de que a $p(n+1)$ não será verdadeira. Denotamos este processo por quantificadores limitados desta forma: $(\exists x < n)p(n)$ pode ser falsa para algum n, mas $\exists x p(x)$ verdadeira. Ou seja, caso tenhamos certeza que existirá alguma testemunha para verdade de $p(x)$, este programa parará, caso não tenhamos esta certeza, o programa pode não parar. Podemos fazer portanto a seguinte correlação:

> $\exists x p(x) \iff$ programa que pára se soubermos que a está em p, e pode não parar se não soubermos.

Chamamos da parte da matemática que pode ser desenvolvida apenas usando quantificadores limitados $(\exists x < n)$ e $(\forall x < n)$ de parte *finitista* da matemática. Ou seja é parte da matemática que temos total certeza que se deixarmos um computador com tempo suficiente, este computador conseguirá dar um resultado ao problema. Qualquer parte que não é finita é considerado como parte *transfinita da matemática*. Em especial a parte que pode ser desenvolvida usando pelo menos fórmulas do tipo $\exists x p(x)$ com apenas uma quantificação existencial é chamada de parte computacionalmente enumerável da matemática.

O programa de Hilbert na sua formulação inicial foi desenvolvido com a ideia de reduzir a consistência da parte transfinita da matemática à consistência da parte finitista da matemática. Tal ideia não apenas pode ser vista como um esforço de proteção dos objetos matemáticos contra críticas matemáticas, mas também uma proteção contra críticas filosóficas. Em especial, seria uma espécie de garantia de que as ideias abstratas da matemática não se corrompem em relação a nossa experiência concreta do mundo, pois enquanto a matemática

conservasse uma coerência interna, manteria um paralelismo com a suposta coerência do mundo.

Como se desenvolveu tal programa?

Segundo Smorysnki [Smo88, p.6], Hilbert, em seu livro *Grundlagen der Geometrie*, olhando para a discussão sobre a axiomática da geometria, deu um novo caráter ao trabalho axiomático da matemática: para que nossas construções e axiomas tenham força, é necessário não somente que eles sejam simples e independentes uns dos outros, mas também é preciso provar a consistência destes objetos, ou seja, em última análise, temos que tomar o cuidado de provar *a priori*, isto é, sem nos preocuparmos com o que eles significam, que não cairemos em contradição. Os axiomas não eram mais vistos como verdades evidentes, mas teriam sua verdade condicionada à consistência do sistema. Na correspondência a Frege, Hilbert explica seu ponto: "Se os axiomas arbitrariamente dados não se contradizem com todas suas consequências, então eles são verdadeiros e as coisas definidas por estes axiomas existem. Este é para mim o critério de verdade e existência" [Smo88, p.7].

Esta visão pretendia responder a dúvida sobre a existência e a verdade de objetos que não só eram contraintuitivos para a época, tais como uma geometria não-euclidiana, mas também fundamentar os passos da matemática nas direções em que o raciocínio finitista da matemática não teria permissão de entrada, isto é, em nossa terminologia, lugares proibidos ao método. Isto porque Hilbert teve sucesso em condicionar a consistência dos axiomas da geometria não-euclidiana e euclidiana [13] à consistência da análise, e a consistência desta à consistência da aritmética. O que levou naturalmente à questão de se a consistência desta aritmética e todos seus enunciados transfinitos (enunciados não finitos tais como qualquer um que tenha um quantificador universal ou existencial) poderiam ser solucionados usando métodos que nenhum matemático colocaria em questão, isto é, apenas usando métodos finitistas de prova. A esta visão ficou associada o epíteto "*o programa de Hilbert*": demonstrar a consistência de sentenças finitas obtidas por métodos transfinitos de prova usando apenas métodos de prova finitos.

A ideia de Hilbert era que se pudéssemos levar a cabo seu programa, teríamos fortes indícios para convencer matemáticos da escola intuicionista de Brouwer, que não aceitavam métodos de provas não construtivas (isto é, métodos de prova em que se usa o axioma do terceiro excluído: $\neg x \vee x$) da existência e a verdade dos enunciados da parte transfinita da matemática. A razão disso é que sabemos que ser demonstrado por métodos finitos implica ser demonstrado por métodos construtivos, e se demonstrarmos a consistência da aritmética por este método, teríamos mostrado a consistência da análise e da geometria, dando indícios de que os objetos trabalhados ali (o número π, por exemplo) existem no sentido finitista do termo. Aqui falamos de 'fortes indícios' porque não conseguiríamos convencer alguém que não acredita na afirmação: a consistência de uma teoria matemática implica a existência e verdade de seus objetos.

[13] Outros matemáticos condicionaram a consistência da geometria não-euclidiana à consistência da euclidiana.

Isto nos leva a Hermann Weyl. Smorysnki mostra que Weyl, em alguns momentos de seu trabalho, tenta "fazer a justiça para os dois lados" [Smo88, p.37], trazendo argumentos tanto para a visão de Brouwer quanto para a visão de Hilbert, embora também tenha críticas duras a ambos. Weyl vê a tentativa de Hilbert de formalizar a matemática como um simples "jogo de fórmulas", algo como o jogo de xadrez, em que, "uma vez que os axiomas e as regras de inferência estejam prontos, apenas precisamos seguir as regras. A prova de consistência seria semelhante a provar que não podemos ter mais de 10 rainhas da mesma cor em um jogo" [Smo88, p.37]. Assim, Brouwer estaria livre desta problemática ao atribuir um conteúdo e evidência verdadeira intuitiva para os axiomas logo de saída. Por outro lado, Weyl via no formalismo de Hilbert a possibilidade de tratar problemas extrínsecos aos problemas matemáticos que não poderiam ser alcançados caso adotássemos uma perspectiva como a de Brouwer, pois teríamos que nos limitar no momento em que tentássemos descrever um objeto exterior à matemática. Por exemplo, se acreditássemos que a lei do terceiro excluído realmente não vale, teríamos como conclusão uma série de resultados em análise, entre eles que o teorema do valor intermediário não vale, e consequentemente a noção de máximos e mínimos de uma função seria diferente, o que sabemos ser de máxima importância na aplicação à predição de fenômenos da natureza.

Weyl teria dito em uma audiência:

> Minha opinião pode ser resumida como se segue: se a matemática for tomada por si mesma, devemos nos restringir a nós mesmos com Brouwer às verdades intuitivamente cognoscíveis ... nada nos compele a ir além. Mas nas ciências naturais nós estamos em contato com uma esfera que é impermeável à evidência intuitiva; aqui cognição necessariamente se torna construção simbólica. Portanto nós não precisamos mais exigir que quando a matemática for tomada no processo de construção teorética na física, deveria ser possível separar o elemento matemático como um domínio especial no qual todos julgamentos são intuitivamente certos; deste ponto de vista mais alto que faz o todo da ciência aparecer como unidade, eu considero que Hilbert esteja certo [Smo88, p.44].

Os teoremas da incompletudes têm como consequência a impossibilidade de levar tal programa iniciado por Hilbert. Em primeiro lugar, o primeiro teorema da incompletude implica que não pode haver nenhuma teoria que conserva os teoremas básicos da aritmética e que ao mesmo tempo é completa no sentido que dado uma fórmula A da teoria, ou a teoria prova ou refuta tal fórmula. O teorema nos impede de reduzir fórmulas transfinitas destas teorias a até mesmo fórmulas do tipo $\exists x A$, com A uma fórmula sem quantificadores, o que torna reduzir para fórmulas do tipo $(\exists x < n)A$, parte finitária da matemática, ainda mais impossível. Em segundo lugar, o segundo teorema da incompletude diz que se assumirmos que uma teoria é consistente, ela não pode provar que ela mesma é consistente. Isto implica na impossibilidade de tomar a consistência como uma pedra de toque para determinar a existência dos objetos de uma teoria.

1.5 A ambivalência computacional

Em 1921, Wittgnstein no seu Tractatus Logico-Philosophicus reflete sobre a natureza dos números: "o número é o expoente de uma operação".[14] Com isto ele quer fazer de cada símbolo de número $0, 1, 2, etc...$ nada mais que uma abreviação de uma função composta consigo mesma determinadas vezes. Em uma notação mais atual isto quer dizer[15]:

$$
\begin{aligned}
0 &=_{def} (f, x) \mapsto x \\
1 &=_{def} (f, x) \mapsto f(x) \\
2 &=_{def} (f, x) \mapsto f \circ f(x) \\
3 &=_{def} (f, x) \mapsto f \circ f \circ f(x) \\
&... \\
n &=_{def} (f, x) \mapsto \overbrace{f \circ f \circ ... \circ f}^{n \text{ vezes}}(x)
\end{aligned}
$$

Isto é, dado uma função (operação) f e um elemento de um conjunto qualquer, 0 significa nem sequer operar f sobre o objeto, 1 significa operar f sobre x uma vez, 2 significa operar f duas vezes sobre x, e assim em diante. Assim o conceito de número se torna uma função, porém uma função que opera sobre outras funções. Ou seja, o número nada mais é que uma função definida em $\mathcal{F} \times X \to A$, onde $\mathcal{F}$ é um conjunto de funções de $X \to A$. Pensado deste modo, o conceito de número nada mais é que uma função que avalia outras funções repetidamente.

Vamos pensar em exemplos cotidianos para tal interpretação. Imaginemos que temos uma máquina que produz uma xícara de café ($f(x)$) toda vez que apertarmos um botão (f). Se não apertarmos nenhuma vez o botão, teremos um copo sem nenhuma xícara de café. Se apertarmos uma vez, teremos um copo com uma xícara de café. E assim sucessivamente. A ideia de número generaliza esta ideia para qualquer tipo de operação. Assim, ela é uma função que se encarna em vários processos realizados no mundo. Pensado do ponto de vista do computador, um número pode ser pensado como um programa onde tal encarnação é possível.

Porém, ao mesmo tempo que podemos pensar números como funções, também podemos pensá-los como inputs para outras funções. Por exemplo, a função sucessor pode ser descrita da seguinte forma:

$$suc(h, f, x) =_{def} f(h(f, x)).$$

[14] Proposição 6.021, [Wit08] .

[15] Esta seguinte análise foi originalmente feita por Peano e Wittgnestein e posteriormente adotada pelo lambda calculus de Church, na codificação dos numerais para o cálculo lambda. No caso $0 := \lambda fx.x$, $1 := \lambda fx.fx$, e assim por diante. Não utilizamos o cálculo lambda aqui para não aumentar o número de definições. Uma análise disto a partir do cálculo lambda pode ser encontrada em [Odi99] no capítulo respectivo sobre cálculo lambda, capítulo I.6, "Functions as Rules".

Assim, suponhamos que $h = 3$, temos:

$$suc(3, f, x) = f(3(f, x)) = f \circ f \circ f \circ f(x) = 4$$

Igualmente, podemos definir a soma da seguinte forma:

$$+(h, g, f, x) =_{def} h(f, g(f, x)).$$

Supondo $h = 2$ e $g = 3$ temos:

$$+(2, 3, f, x) = 2(f, 3(f, x)) = f \circ f(3(f, x)) = f \circ f \circ f \circ f \circ f(x) = 5.$$

Assim de acordo com este ponto de vista, o número é um objeto no qual outras funções agem. Ou dito de outra forma, número é visto como input para outros programas. Mas o que são estes outros programas que operam números, no final das contas? Também podem ser vistos como números. Isto pode ser observado quando realizamos as operações até o final. De outra forma, podemos ver que *suc* está definido da seguinte forma:

$$suc : \mathcal{H} \times \mathcal{F} \times X \to A,$$

onde $\mathcal{H}$ são funções definidas em $X \to A$ e $\mathcal{F}$ também são definidas assim. E notemos que $\mathcal{H} \times \mathcal{F}$ opera em X, da mesma forma que um único conjunto de funções $\mathcal{G}$ operaria em um conjunto X. Mais sobre isso será discutido adiante nesta seção.

A moral da história é que há uma associação entre números e programas. Números são programas e programas são números. Se esquecermos o que falamos aqui, e pensarmos na associação com computadores, esta afirmação é corroborada. Programas são interpretados por computadores como sequências de zero e uns, que correspondem a ausência e presença de circuito elétrico em seus componentes. Estes componentes trabalham com estes impulsos elétricos e podemos interpretá-los como números. Logo programas são números. Da mesma forma, podemos utilizar estes números para criar programas. Ou seja, números são programas.

Vamos adentrar um pouco mais nesta comparação. O hardware de um computador pode ser pensado como o conjunto de circuitos, memória, e todo o aparato físico que vemos. O software pode ser pensado como um conjunto de instruções que este computador opera, ou seja, como um conjunto de instruções digitadas e guardadas na memória de tal forma que sempre que precisarmos de tal programa, o computador faz o trabalho de realizar as determinadas instruções correspondentes. Mas deste ponto de vista, software é nada mais que um input. Ou seja, não precisamos abrir a máquina e modificar seus constituintes físicos e hardware para criarmos uma nova máquina que faz outra operação, caso que ocorreria por exemplo se quiséssemos transformar uma calculadora em um dispositivo que faz outra coisa. Neste sentido o computador é algo programável.

Por que isto ocorre? Por que não precisamos mudar o hardware de cada computador para modificar os programas? A resposta é porque de um ponto de

vista mais abstrato o software é hardware. Ou seja que podemos simular a partir do software máquinas de uso específico, sem termos que realmente construir tais máquinas. A analogia com os números e funções retorna novamente. Da mesma forma como números podem ser pensados como programas, o software pode ser pensado como hardware. E da mesma forma como programas podem ser pensados como números, o hardware pode ser pensado como software.

Para analisar melhor esta ideia, vamos voltar novamente à ideia inicial desta seção: vimos que números são funções com esta estrutura:

$$\phi : X \times T \to A,$$

onde T é pensado como um conjunto de funções. Se abstrairmos T e pensarmos como um conjunto qualquer, então podemos pensar que esta ϕ é uma função definida assim:

$$\phi(x, t) = a,$$

para x, t, a membros dos respectivos conjuntos X, T, A. Notemos que sempre podemos passar desta definição de ϕ para uma ϕ' da seguinte forma:

$$\phi(x, t) = (\phi'(x))(t),$$

onde $(\phi'(x))(t)$ é o resultado de aplicar primeiro a função $\phi' : X \to (T \to A)$ a x,[16] gerando assim uma função $\phi'' : T \to A = (\phi'(x))$, e por conseguinte aplicar esta função ϕ'' a t, nos dando $\phi''(t) \in A$. Dito de outra forma, ϕ' faz o seguinte:

$$x \mapsto (t \mapsto a),$$

onde $t \mapsto a$ é a função ϕ'' elemento de A^T.

Em resumo, ao afirmarmos que tal transformação sempre pode ocorrer, (i.e. que dado ϕ sempre existe ϕ' e vice-versa) temos as seguinte bijeção entre o conjunto de funções definidas nestes respectivos conjuntos, isto é [17]:

$$X \times T \to A \cong X \to A^T.$$

Vamos enunciar e provar que, no contexto de conjuntos, sempre podemos fazer esta bijeção:

PROPOSIÇÃO **1.** *Existe uma função bijetora* $\Phi : C^{A \times B} \to (C^B)^A$.

Demonstração. Para isto mostraremos que existe uma função inversa

$$\Psi : (C^B)^A \to C^{A \times B}$$

tal que $\Psi \circ \Phi = Id$, e $\Phi \circ \Psi = Id'$, onde $Id : C^{A \times B} \to C^{A \times B}$ é a função identidade, e $Id' : (C^B)^A \to (C^B)^A$, também é a função identidade.

[16] A notação $T \to A$ denota as funções de T em A, em outra notação isso significa A^T.

[17] Utilizamos a notação $\cong$ para signficar que existe uma função bijetora entre os dois conjuntos de funções. Tal notação não é usual, porém faz referência a um resultado mais forte em teoria das categorias.

Dado $f : A \times B \to C$, definimos:[18]

$$g(a) := f(a, -)$$

E definimos:

$$\Phi(f)(a)(b) := g(a)(b)$$

Analogamente, dado $j : A \to C^B$, definimos:

$$i(a, -) := j(a)$$

E definimos:

$$\Psi(j)(a, b) := i(a, b).$$

Assim:

$$\Psi \circ \Phi(f)(a, b) = \Psi(\Phi(f))(a, b) = \Phi(f)(a)(b) = f(a, b).$$

A primeira igualdade pela definição de composição, a segunda pela definição de Ψ, a terceira pela definição de Φ.

Portanto, $\Psi \circ \Phi = Id$.

Por outro lado:

$$\Phi \circ \Psi(j)(a)(b) = \Phi(\Psi(j))(a)(b) = \Psi(j)(a, b) = j(a)(b).$$

A primeira igualdade pela def. de composição, a segunda pela def. de Φ, a terceira pela def. Ψ.

Portanto, $\Phi \circ \Psi = Id'$.

□

Esta bijeção é o que está na base do método de *currying* presente em diversos fenômenos da computação (o nome currying é em homenagem ao lógico Haskell Curry). A ideia é que sempre podemos pensar programas definidos em mais de uma variável como programas que operam com cada variável de cada vez. Com esta correspondência podemos por exemplo postergar a execução dos programas definidos em $A \to T$, pensando primeiramente nos programas definidos em $X \to A^T$. A passagem inversa, de $X \to A^T$ para $X \times T \to A$ é chamado de *uncurrying*, e justamente permite pensar dois programas definidos separadamente como um programa só definido no par ordenado correspondente.

Se nos referirmos à reflexão feita anteriormente, vemos que esta bijeção na realidade é a essência da ambivalência: número vs programa. O número t é pensado no conjunto de funções $X \times T \to A$ como um input, como software, já t em $X \to A^T$ está associado à função $t \mapsto a$, isto é o programa, o hardware. Vamos voltar a nossa análise da função sucessora:

[18] Usamos a notação $f(a, -)$ para denotar que a é um elemento fixo de A e a função $f(a, -)$ por seguinte se torna uma função elemento de B^C.

$$suc : \mathcal{H} \times \mathcal{F} \times X \to A,$$

que pode ser pensada depois do currrying como

$$\mathcal{H} \times \mathcal{F} \times X \to A \cong \mathcal{F} \times \mathcal{H} \to (A^X),$$

a expressão da direita se comporta como um número, basta observar que este conjunto de funções tem a mesma estrutura da definição que demos antes para números como funções definidas em $\mathcal{F} \times \mathcal{H} \to B$, onde B neste caso é A^X. Já a expressão da esquerda se comporta como a função sucessor. Mais uma vez, vemos que todo programa pode ser pensado como número ao fazermos operações de currying, e todo número pode ser pensado como um programa, caso façamos o uncurrying. Deste ponto de vista, respondemos à pergunta: por que não precisamos abrir um computador para mudar o que ele opera? Ora, porque o computador pode ser pensado como uma máquina que encarna esta ideia de número/programa. Dito de outro modo, os componentes do computador possuem complexidade suficiente para fazer as operações de currying e uncurrying.

Dizer que algo é enumerável quer dizer que podemos indexá-lo por números. Isto quer dizer que podemos representar $(\phi'(x))(t)$ da seguinte forma: $\phi_t(x)$ e portanto estamos afirmando que, para toda ϕ definida em $X \times T \to A$, existe uma ϕ_t definida em $X \to A$ tal que:

$$\phi(x, t) = \phi_t(x).$$

Aqui de forma mais evidente ainda, transformamos cada programa ϕ de duas variáveis, em um programa de uma variável ϕ'. Isto quer dizer que todo programa pode ser enumerado. Por exemplo o programa que faz a operação de sucessor suc será um determinado programa ϕ_n que opera apenas em X. Da mesma fora, a soma, após algumas operações de currying, pode chegar a uma forma $\phi(x, t)$, e que por sua vez, com mais uma operação de currying, pode ser pensado como um programa que opera apenas em X.

A computabilidade é um ramo de estudo da computação que se interessa nos aspectos teóricos da computação. Como todo campo de estudo, há determinados teoremas que podem ser pensados como ponto de partida. Estas ideias que esboçamos aqui nesta seção, de que é possível realizar um processo de currying e uncurrying no contexto computacional, são teoremas deste gênero, e a eles é dado o nome de teorema do parâmetro (currying) e enumeração (uncurrying). Porém, não é muito claro como estes teoremas são possíveis se de fato partirmos de definições mais simples. Um dos objetivos nossos é tentar fazer um trajeto que forneça esta ideia. É o que faremos ao abordar o teorema 6 da enumerabilidade e o teorema 7 do parâmetro.

1.6 Ponto-fixo: fenômeno geral de auto-referência

Vimos nesta introdução que a ideia de imitação é fundamental para a computação. Utilizamos várias palavras para nos referirmos a este conceito: emulação,

referência, reprodução, etc... O fato de que podemos fazer isto no contexto computacional é em grande parte dado pela ambivalência abordada na seção anterior. Em particular, se podemos imitar algo, podemos gerar fenômenos de auto-referência, ou auto-imitação, etc... Basta pensar no seguinte resultado:

$$\phi(x, x) = \phi_x(x).$$

O que isto diz é que o programa ϕ_x está operando o número que corresponde a si mesmo em uma enumeração dada.

Porém, como é possível determinar fenômenos como estes em geral? Para isto é necessário entender que fenômenos de auto-referência estão ligados a ideia de ponto-fixo. Vamos esboçar uma teoria geral sobre isto partindo do resultado de auto-referência historicamente fundamental: o teorema de Cantor. As ideias a seguir são adaptadas do exposto por Yanofksy em [Yan03].

Um ponto-fixo de uma função $f : X \to X$ de uma maneira geral é apenas um ponto c tal que

$$f(c) = c.$$

Ou seja, se pensarmos no contexto dos números reais, é simplesmente uma função que intercepta a função identidade. Em conjuntos, se X for um conjunto com cardinalidade maior que ou igual a 2, sempre é possível criar tanto funções específicas que possuem ponto-fixo, quanto funções específicas que não possuem ponto-fixo. Por exemplo, no caso do conjunto com dois elementos que denotaremos $2 = \{0, 1\}$, temos a seguinte função (negação) sem ponto-fixo:

$$0 \mapsto 1$$
$$1 \mapsto 0$$

e a seguinte função com ponto-fixo no 0:

$$0 \mapsto 0$$
$$1 \mapsto 0$$

E o que isto tem a ver com o teorema de Cantor? Vamos primeiramente dar uma prova do teorema de Cantor em sua forma mais tradicional.

TEOREMA **1.** ***Teorema de Cantor***
Não existe uma função sobrejetora f definida nos seguintes conjuntos:

$$\mathbb{N} \to \wp(\mathbb{N}).$$

Demonstração. Suponha que existe uma função f sobrejetora entre $\mathbb{N}$ e seus subconjuntos. Isto significa que podemos enumerar cada um dos subconjuntos de $\mathbb{N}$:

$$S_0, S_1, S_2, ...$$

Defina o seguinte subconjunto de $\mathbb{N}$:

$$D = \{x \in \mathbb{N} : x \notin S_x\}.$$

Como D é um elemento de $\wp(\mathbb{N})$, ele deve ser algum S_n. Assim:

$$n \in D \Leftrightarrow n \notin S_n \Leftrightarrow n \notin D.$$

Porém, isto é uma contradição. Portanto, f não pode ser sobrejetora. □

O primeiro ajuste a ser feito para pensarmos a relação com o teorema do ponto-fixo é transformarmos toda esta argumentação em uma argumentação do ponto de vista das funções características envolvidas. Podemos pensar $\wp(\mathbb{N})$ como o conjunto das funções características dos subconjuntos de $\mathbb{N}$. Ou seja, pensamos nas funções definidas $\mathbb{N} \to 2$. Por exemplo o conjunto $\{2, 3\}$ tem como função característica associada a função f que leva $2 \mapsto 1$ e $3 \mapsto 1$ e o restante dos números no zero. Todo subconjunto de $\mathbb{N}$ possui uma função característica. Portanto existe uma bijeção: $\wp(\mathbb{N}) \cong 2^{\mathbb{N}}$, onde $2^{\mathbb{N}}$ é o conjunto de funções de $\mathbb{N}$ para o conjunto 2. Isto quer dizer que o teorema de Cantor pode ser enunciado: não existe f sobrejetora em:

$$\mathbb{N} \to 2^{\mathbb{N}}.$$

Lembremos da seção passada, que pelo método de currying, sabemos que o conjunto das funções f definidas acima tem a mesma cardinalidade das funções definidas em:[19]

$$\mathbb{N} \times \mathbb{N} \to 2.$$

Notemos que apesar disto, existe uma função sobrejetora no conjunto de funções acima (basta fazer $f((0,0) = 0$ e $f((0,1)) = 1$). Para entender como enunciar o teorema de Cantor neste contexto, vejamos como podemos enunciar a função que desempenha f no contexto da prova original:

$$f(x, y) = \begin{cases} 1, \text{ se } x \in S_y \\ 0 \text{ se } x \notin S_y. \end{cases}$$

Como podemos ver, $f(-, m) : \mathbb{N} \to 2$ desempenha o papel de função característica de uma enumeração possível de $\wp(\mathbb{N})$. [20] Um enunciado equivalente ao teorema de Cantor é dizer que as funções definidas em

[19] Para ver isto basta checar que para cada função definida em $\mathbb{N} \to 2^{\mathbb{N}}$ do tipo $(f(n))(m)$ é possível encontrar pelo processo de uncurrying uma função do tipo $f(n, m)$.

[20] Usamos a notação $f(-, m)$ para indicar que m está fixo e a função é de apenas uma variável, com a variável definida no conjunto original. Podemos fazer isto porque podemos fazer currying na função original definida em $\mathbb{N} \times \mathbb{N} \to 2$.

$$f(-, m) : \mathbb{N} \to 2$$

não podem ser enumeradas.

Vamos pensar agora qual função característica do conjunto $D = \{x \in \mathbb{N} : x \notin S_x\}$ da prova que fizemos acima do teorema de Cantor. Se tomarmos a função $f : \mathbb{N} \times \mathbb{N} \to 2$ definida sobre os elementos (x, x) de $\mathbb{N} \times \mathbb{N}$ (isto é, a dupla ordenada que possui os duas coordenadas iguais, como $(0, 0)$, $(1, 1)$, e assim por diante) chegaríamos na função característica do conjunto:

$$\{x \in \mathbb{N} : x \in S_x\}$$

A única coisa que temos que fazer é tomar a negação deste conjunto. Definimos assim uma função *neg*:

$$\begin{aligned} neg : 2 &\to 2 \\ 1 &\mapsto 0 \\ 0 &\mapsto 1 \end{aligned}$$

Notemos que esta é uma função sem ponto-fixo. Com esta função definimos a função:

$$\begin{aligned} d : \mathbb{N} &\to 2 \\ x &\mapsto neg(f(x, x)). \end{aligned}$$

Esta função faz o papel de função característica do conjunto D. Vejamos diagramaticamente o que está ocorrendo:

$$\begin{array}{ccc} \mathbb{N} \times \mathbb{N} & \xrightarrow{\quad f \quad} & 2 \\ {\scriptstyle \Delta}\uparrow & & \downarrow{\scriptstyle neg} \\ \mathbb{N} & \xrightarrow[\quad d \quad]{} & 2 \end{array}$$

onde $\Delta(x) = (x, x)$ é a função que duplica os valores recebidos, e o restante das funções já foram definidas. Dizemos que este diagrama *comuta* se:

$$d = neg \circ f \circ \Delta,$$

ou seja, que para todo $x \in \mathbb{N}$:

$$d(x) = neg(f(\Delta(x))) = neg(f(x, x)).$$

Que é equivalente à nossa definição original.[21]

Apenas nos resta estabelecer mais uma notação para podemos enunciar novamente o teorema de Cantor nesta versão apenas para funções. Dizemos que d *é representável* por f se existe um m tal que:

$$f(-, m) = d(-).$$

Notemos que neste contexto esta condição é equivalente a dizer que há uma enumeração das funções $f(-, m)$, que representam subconjuntos de $\mathbb{N}$. Queremos mostrar o contrário disso, ou seja que não há tal enumeração possível, isto é:

TEOREMA **2.** ***Teorema de Cantor segunda versão:***

Toda função $f : \mathbb{N} \times \mathbb{N} \to 2$ *existe uma função* $d : \mathbb{N} \to 2$ *que não é representável. Quer dizer, para todo* m*:*

$$d(-) \neq f(-, m).$$

Demonstração. No contexto das definições anteriores sabemos que o seguinte diagrama comuta.

$$\begin{array}{ccc} \mathbb{N} \times \mathbb{N} & \xrightarrow{\quad f \quad} & 2 \\ {\scriptstyle \Delta}\uparrow & & \downarrow{\scriptstyle neg} \\ \mathbb{N} & \xrightarrow[\quad d \quad]{} & 2 \end{array}$$

Como vimos, $f(-, m)$ é a função característica dos conjuntos $S_1, ..., S_m, ...$ de $\mathbb{N}$, e d é a função característica de $D = \{x \in \mathbb{N} : x \notin S_x\}$.

Afirmamos que $d(-) \neq f(-, m)$.

De fato, suponha o contrário, logo temos:

$$f(m, m) = d(m) = neg(f(m, m)),$$

a primeira igualdade pois supomos o contrário, a segunda pelo diagrama. Mas isto é um absurdo, pois neg não tem ponto-fixo.

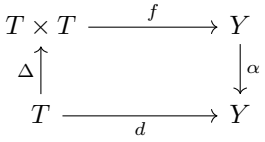
Fica provado o teorema. E além disso, podemos ver que $d(-)$ não está na enumeração esperada dos subconjuntos de $\mathbb{N}$. □

O interessante desta prova surge quando olhamos sua contrapositiva:

[21] Como é esperado, tal linguagem diagramática é inspirado na linguagem mais geral das teoria das categorias. Com efeito, tal discussão pode ser generalizada no teorema do ponto-fixo de Lawvere, onde trabalhamos com categorias cartesianas fechadas no lugar de conjuntos. Tal trabalho foi feito originalmente por Lawvere em[Law06].

TEOREMA **3.** ***Teorema do ponto-fixo versão abstrata:***[22]

Se Y e T são conjuntos e existe uma função $f : T \times T \to Y$ tal que para toda função $d : \mathbb{N} \to T$, d é representável por f (quer dizer, existe m tal que $d(-) = f(-, m)$), então toda função $\alpha : Y \to Y$ possui ponto-fixo.



Tomamos a liberdade de generalizar esta versão, tomando T no lugar de $\mathbb{N}$ e Y no lugar de 2. A prova segue-se igualmente à prova do anterior.

Aparentemente, tal versão do teorema de Cantor não parece adicionar muita informação. Porém, ao mudarmos de estrutura e pensarmos nas funções com mais estrutura, como as funções computáveis, isto está dizendo que existe uma maneira computável de passar de uma função computável ϕ_n para uma outra função $\phi_{f(n)}$. Veremos com detalhe como fazer isto no teorema do ponto-fixo para funções computáveis 9, em especial veremos como os teoremas explorados na sessão anterior, do parâmetro e da enumeração, implicam neste teorema. A seguir aprofundaremos este teorema no contexto abstrato do teorema da incompletude.

1.7 Teorema da incompletude pela perspectiva do ponto-fixo

Adaptaremos a ideia geral do teorema da incompletude da forma como foi exposto por Smullyan em seu capítulo inicial de [Smu92]. Criaremos uma teoria simplificada. Suponha que temos a linguagem L com os seguintes símbolos:

$$\neg, P, N, 1, 0$$

Para cada letra podemos associar um número em sua representação binária da seguinte forma (não converteremos para decimal, mas temos em vista que $11 = 3$ e assim por diante):

[22] Este teorema é o correspondente ao teorema da diagonal em [Yan03]. Este teorema deve ser tomados com cuidado quando pensado na teoria dos conjuntos, pois é falso se $|Y| \geq 2$. Como vimos ele é falso em geral exatamente pelo teorema de Cantor: para qualquer conjunto não-trivial $|Y| \geq 2$ é possível construir uma $\alpha : Y \to Y$ sem ponto-fixo (pensar em uma variante da função negação, que manda todos elementos diferentes de x para x e manda x para algum elemento diferente de x). O teorema do ponto-fixo só valerá portanto para conjuntos triviais com $|Y| = 1$. A situação muda de figura quando pensamos em categorias com mais estruturas, como a estrutura das funções computáveis ou fórmulas de uma teoria. Neste último caso $|Y|$ pode ser maior.

$$\begin{aligned} \ulcorner \cdot \urcorner : L &\to \mathbb{N} \\ \neg &\mapsto 10 \\ P &\mapsto 100 \\ N &\mapsto 1000 \\ 1 &\mapsto 10000 \\ 0 &\mapsto 100000 \end{aligned}$$

Podemos induzir a operação efetuada pela função codificadora $\ulcorner \cdot \urcorner$ para o conjunto dos *strings* (*palavras* de 0 ou mais letras, isto é sequências formadas pela concatenação de letras). Para cada string podemos associar o *número de Gödel* correspondente a partir de uma concatenação dos números binários correspondentes. Por exemplo:

$$\ulcorner PN\neg \urcorner = 100100010$$

Notemos que esta associação é sobrejetora, porém não bijetora. Isto não nos atrapalhará. Há diversas soluções de codificação, inclusive bijetoras [23].

Uma *norma* de um string é um string seguido de seu número de Gödel, por exemplo:

$$PN\neg 100100010$$

Uma *sentença* ($Sent$) é um string escrito das seguintes formas:

$$PX, PNX, \neg PX, \neg PNX,$$

onde X é um número qualquer em notação binária.

Definimos um predicado ($Pred$) como os strings da seguinte forma:

$$P, PN, \neg P, \neg PN.$$

Utilizamos uma função interpretação $\mathcal{A}$ para as sentenças:

$$\mathcal{A} : Sent \to 2,$$

tal que (escrevemos 1 para significar verdade, 0 para falsidade):

- $\mathcal{A}(PX) = 1$ sse X é demonstrável no sistema;
- $\mathcal{A}(PNX) = 1$ sse a norma de X é demonstrável;
- $\mathcal{A}(\neg PX) = 1$ sse X não é demonstrável;

[23] Trataremos de codificações canônicas na definição 17. Porém é interessante notar que a codificação utilizada neste exemplo tem uma propriedade que nos interessará posteriormente: ela não gera ambiguidade. Sempre sabemos quando uma letra acaba e outra começa: quando temos um 1.

- e $\mathcal{A}(\neg PNX) = 1$ sse a norma de X não é demonstrável.

Assumimos que estes sistema que descrevemos acima é *correta*, ou seja, ele não prova sentenças falsas, nem refuta sentenças verdadeiras.

O problema que Smullyan pede é:

> Qual sentença que é verdadeira mas não é demonstrável?

Com algumas tentativas e erros conseguimos chegar na seguinte sentença:

$$\neg PN101001000$$

ou:

$$\neg PN\ulcorner\neg PN\urcorner$$

Se esta sentença é falsa, então a norma de $\neg PN$ é demonstrável. Porém a norma disto também é a própria sentença inteira, $\neg PN\ulcorner\neg PN\urcorner$. Mas então ela é falsa e demonstrável, uma contradição com a correção.

Se a sentença é verdadeira, então a norma de $\ulcorner\neg PN\urcorner$ não é demonstrável. Porém a norma disto é a própria sentença. Então ela é verdadeira e não demonstrável. A única possibilidade.

A sentença também não pode ser refutável, pois vimos que é verdadeira, e pela correção não é possível refutar fórmulas verdadeiras.

Portanto, achamos uma sentença cuja verdade implica sua própria não demonstrabilidade. Achamos o teorema da incompletude abstrato para esta pequena teoria. Pois dizemos que uma teoria é *incompleta* se existe uma sentença que a teoria não prova nem refuta. Sabemos que o teorema não prova a sentença, mas também não pode refutar, pois pela correção a teoria apenas refuta sentenças falsas, e já vimos que tal caso leva à contradição. Assim, temos um pequeno teorema da incompletude para este sistema primitivo.

Qual o problema desta prova? Por que os livros não param por aí e tem que descrever uma teoria mais sofisticada, como a da aritmética? Bom, um primeiro problema é que nós partimos do pressuposto que existe um P que é um predicado de provabilidade, não provamos que este predicado realmente é construtível dentro da teoria. Alguém poderia dizer que tal coisa é impossível. Para construir tal predicado é preciso uma teoria como a da aritmética.

Até aqui expomos na íntegra o sistema criado por Smullyan com algumas modificações. Agora vamos pensar como o teorema do ponto-fixo entraria em cena.

Podemos definir o conjunto $Pred^*$ como aquele dos números de Gödel dos predicados da teoria. Isto é:

$$Pred^* = \{\ulcorner P\urcorner, \ulcorner PN\urcorner, \ulcorner\neg P\urcorner, \ulcorner\neg PN\urcorner\}$$

Podemos definir o conjunto das classe de equivalência $[Sent]$ cuja relação de equivalência subjacente é a seguinte: duas sentenças estão em relação sse são logicamente equivalentes. Por exemplo, as seguintes sentenças são equivalentes:

$$PN\ulcorner P\urcorner \iff P\ulcorner P\urcorner,$$

pois se $PN^{\ulcorner}P^{\urcorner}$ é verdadeira, então a norma de P é demonstrável, e isto significa dizer que $P^{\ulcorner}P^{\urcorner}$ é verdadeira. Da mesma forma, se supormos que $PN^{\ulcorner}P^{\urcorner}$ é falsa, $P^{\ulcorner}P^{\urcorner}$ é falsa.

Nós criamos uma relação de equivalência para o conjunto de sentenças tal que, para X e Y sentenças:

$$X \sim Y \text{ sse } (X \iff Y).$$

E definimos assim $[Sent]$ como as classes de equivalências induzidas por esta relação. Denotamos os elementos destas classes de equivalência, para facilitar a notação, da mesma forma como denotamos as sentenças. Ou seja:

$$P^{\ulcorner}P^{\urcorner} = \overline{P^{\ulcorner}P^{\urcorner}} \in [Sent]$$

A partir destes conjuntos podemos agora utilizar o teorema do ponto-fixo. Definiremos as funções utilizadas na hipótese do teorema:

Definimos uma função f da seguinte maneira:

$$\begin{aligned} f : Pred^* \times Pred^* &\to [Sent] \\ f(x, y) &\mapsto Y^{\ulcorner}x^{\urcorner}, \text{ onde } y = {\ulcorner}Y^{\urcorner} \text{ e } Y \text{ é algum predicado.} \end{aligned}$$

Definimos a função α como uma função em:

$$\alpha : [Sent] \to [Sent]$$

Definimos a função d:

$$\begin{aligned} d : Pred^* &\to [Sent] \\ d({\ulcorner}X^{\urcorner}) &\mapsto X^{\ulcorner}X^{\urcorner} \end{aligned}$$

E isso de tal forma que o seguinte diagrama comuta (Δ é a função diagonal definida como na seção anterior):

$$\begin{array}{ccc} Pred^* \times Pred^* & \xrightarrow{\quad f \quad} & [Sent] \\ {\scriptstyle\Delta}\uparrow & & \downarrow{\scriptstyle\alpha} \\ Pred^* & \xrightarrow[\quad d \quad]{} & [Sent] \end{array}$$

Afirmamos que d é representável por f, quer dizer, existe um m tal que $f(-, m) = d(-)$. De fato:

$$d({\ulcorner}X^{\urcorner}) = X^{\ulcorner}X^{\urcorner} = f({\ulcorner}X^{\urcorner}, {\ulcorner}X^{\urcorner}).$$

A primeira igualdade se segue da definição de d e a segunda da definição de f.

Logo, pelo teorema do ponto-fixo vai existir um ponto-fixo para α, em especial a seguinte sentença é um ponto-fixo:

$$X\ulcorner X\urcorner$$

onde X é um predicado qualquer. Notemos que esta sentença está bem definida. Para mostrar que é um ponto-fixo fazemos as seguintes contas:

$$\begin{aligned}
\alpha(X\ulcorner X\urcorner) &= \alpha(f(\ulcorner X\urcorner, \ulcorner X\urcorner)) && \text{def. } f \\
&= d(\ulcorner X\urcorner) && \text{pela composição do diagrama} \\
&= f(\ulcorner X\urcorner, \ulcorner X\urcorner) && \text{pela representabilidade.} \\
&= X\ulcorner X\urcorner && \text{def. } f
\end{aligned}$$

Ou seja, para qualquer função α temos um ponto-fixo do aspecto indicado acima. Em especial, definimos a seguinte α, para F um string qualquer:

$$\alpha(X) = F\ulcorner X\urcorner$$

A função $\alpha : [Sent] \to [Sent]$ é uma função definida da forma que se enquadra no teorema do ponto-fixo. Mas como estamos trabalhando com representantes de classes de sentenças equivalentes, isto induz a seguinte equivalência:

$$F\ulcorner X\urcorner \iff X$$

Com isto podemos dar uma intuição para o teorema do ponto-fixo para lógica:

Teorema **4.** *Para toda fórmula F de uma variável, existe uma sentença X tal que T :*

$$F\ulcorner X\urcorner \iff X.$$

Podemos interpretar este teorema do ponto-fixo pensando que X é uma sentença que diz que ela mesma possui a propriedade F.

Se fizermos $F = \neg PN$, temos:

$$\neg PN\ulcorner X\urcorner \iff X.$$

Quem será esta X em específico? Vimos que X pode ser escrito como: $\neg PN\ulcorner\neg PN\urcorner$. Ou seja, $\neg PN\ulcorner\neg PN\urcorner$ é exatamente a sentença que achamos anteriormente, e que provamos ser indecidível sem usar o ponto-fixo, em nossa versão inicial do problema. A sentença $\neg PN\ulcorner\neg PN\urcorner$ diz, dela mesma, que ela não pode ser provada. Podemos assim usar o teorema para não só provar o teorema da incompletude, mas para gerar as sentenças que não podem ser provadas no sistema (e justamente por isso são verdadeiras).

Pode parecer um desvio desnecessário, porém tais argumentações valem igualmente para sistemas formais mais complexos como o da aritmética. E além

disso, nos permite mostrar construtivamente que tais sentenças não prováveis existem.

Capítulo 2

Perto da máquina

> Neste capítulo introduziremos o conceito de máquina de Turing. Nos próximos capítulos, referenciaremos a este conceito de forma não tão precisa, porém a ideia geral de que se trata de um mecanismo no sentido mais concreto possível continuará no horizonte de fala.

A história do trabalho matemático está familiarizada com a noção de mecanização. Vamos dar um argumento de porque esta familiaridade existe. Intuitivamente, e numa conversa informal, poderíamos dizer que uma atividade humana pode ser mecanizada quando pode ser substituída por uma máquina ou um mecanismo. Assim, por exemplo, atividades que vivenciamos diariamente como lavar a louça ou fazer café podem ser imaginadas como sendo feitas por máquinas — uma lavadora de pratos e uma cafeteira são exemplos. As atividades matemáticas podem parecer, em um primeiro momento, mais resistentes à mecanização, porém nossa experiência com calculadoras e procedimentos algorítmicos como manejar um ábaco não corroboram com esta hipótese. E a computação, unida à linguagem da matemática, levou seu sucesso operacional simbólico a tal ponto, adquirindo tecnologias cada vez mais sofisticadas, que a tese da manipulação de símbolos como atividade exclusiva do pensamento parece a cada dia encontrar mais contra-exemplos.[1] Mas, neste ponto é possível que façamos a pergunta: o que é uma máquina?

Quando rascunhamos em um papel uma conta de multiplicação de dois números grandes, utilizamos um algorítimo, e sabemos que se seguirmos os procedimentos que aprendemos em algum ponto de nossa vida escolar, chegaremos a um resultado correto, independente do tamanho dos números multiplicados ou do tempo que levamos para efetuar a conta. Assim, damos um primeiro passo para pensar máquinas no sentido abstrato. Não estamos mais pensando em máquinas concretas, mas máquinas que lidam com tempo infinito, imateriais, e cuja

[1] Porém a tese recíproca, de que a mente é exclusivamente manipulação de símbolos, faz parte, no mínimo, de uma polêmica muito mais delicada. Sem mencionar que a querela mecanicismo vs platonismo foi ressuscitada a partir dos teoremas da incompletude, e Gödel mesmo foi um articulador a favor do último. Para um resumo ver Odiffredi [Odi99], p. 113, Franzèn (2005).

imagem estaria mais próximo de um agente mental. Seja qual for a qualidade do agente que efetua tais procedimentos mentais, mesmo que desviando de nossa ideia intuitiva do que seria uma máquina, não parece completamente descabido chamar de máquina esse conjunto de procedimentos efetivos simples que após o fornecimento de uma entrada nos dá uma saída determinada ou então nunca para.

Máquinas no sentido abstrato (mecanismos) possuem a característica de simplificar muitas tarefas que seriam impossíveis de serem efetuadas devido a nossas muitas limitações humanas, como atenção, memória, tempo, dedicação, tamanho de uma folha de papel — e no caso de máquinas ideais, o objeto do estudo da computabilidade, a limitação física propriamente dita.

Desta perspectiva, fica claro que a história da experiência matemática está familiarizada com o conceito de máquina, e também de que ela viu áreas cada vez mais extensas da matemática serem suscetíveis à mecanização: cálculos, resolução de equações, plot de gráficos, aproximações de valores de funções. A noção de máquina de Turing nos auxiliará nesta tarefa de identificar o componente maquínico presente no dia-a-dia do matemático, já que ela foi considerada por muitos (até pelo próprio Gödel) o constructo que melhor representa intuitivamente a ideia do que seria esta mecanização.

2.1 Máquina de Turing

2.1.1 Ideia Intuitiva de Máquina de Turing

Antes de iniciarmos nossa conceitualização, tentemos colocar a definição de máquina de Turing da forma mais genérica possível. Isto nos será útil porque há várias definições equivalentes de máquinas de Turing, logo, procuraremos utilizar da imprecisão para abarcar o maior número delas para depois restringirmos a uma definição mais específica.

Uma *máquina de Turing* é um tipo de modelo abstrato de mecanismo com determinadas regras que lê ou escreve um número finito de símbolos em uma lista de tamanho infinito. Esta definição prévia não dá conta do que queremos dizer, por esta razão, analisaremos ponto por ponto o que queremos dizer por cada uma destas palavras.

- **(...) é um tipo de modelo abstrato de mecanismo:** com isto queremos dizer que este mecanismo não tem preocupações mundanas como, por exemplo, pagar a conta da energia, nem se preocupar com o espaço-tempo. As operações do mecanismo se realizam como se a máquina em questão pudesse estar em movimento perpétuo, com espaço ilimitado e realizando também suas tarefas com tempo ilimitado. Este mecanismo é aproximadamente a parte do hardware que denominamos, na computação atual, de processador (ou CPU). Obviamente, as considerações abstratas não se aplicam quando pensamos nos exemplos "reais", a energia se dissipa e aparentemente não há conclusões finais sobre o tamanho nem a duração do universo. Quando formalizarmos a definição, esta parte da máquina de

Turing se manifestará no poder notacional da definição matemática que daremos.

- **(...) com determinadas regras:** com isto queremos dizer que este modelo abstrato segue um programa dado de antemão gravado no hardware da máquina, que diz quais serão as instruções do mecanismo. Esta lista de regras deve estar de acordo com as configurações possíveis da máquina de Turing. Por exemplo, imaginemos que determinamos que nosso mecanismo vive em apenas uma dimensão, ele só pode ir para direita ou para esquerda, não faz sentido dizer para ele ir para cima.

  O mecanismo que descreveremos terá de antemão três operações possíveis. Ele opera como se fosse um trem passando por um trilho que contém células. O maquinista pode acionar apenas três alavancas, que fazem as seguintes operações:

  a Ler a cédula em que o trem está em cima;

  b Escrever na cédula algum símbolo;

  c Ou pular para próxima cédula da direita ou esquerda.

  A este maquinista é dado um conjunto de instruções, que estipulam quais botões ele deve apertar para cada estágio da computação. Estes estágios podem ser pensados como itens a serem cumpridos, ou estados. Assim por exemplo, podemos dar instruções do tipo: no primeiro estado, leia o símbolo que está na célula, se for o símbolo \$, retorne ao primeiro estado, caso contrário vá para o segundo estado. No segundo estado, leia etc...

  O maquinista é alguém que faz tudo isto em velocidade infinita. Como pode ser visto, estaremos limitando a máquina de Turing à unidimensionalidade, porém é possível mostrar que modelos bidimensionais ou tridimensionais são equivalentes a esta definição. Nos computadores atuais esta parte do hardware também pode ser pensada como localizada no conjunto de partes que constituem o processador, algo como um programa que vêm de fábrica com a máquina de Turing. Que este pedaço do hardware pode ser interpretado como um programa (um software) por uma outra máquina de Turing será importante para nós posteriormente, pois é um outro modo de enunciar o teorema da universalidade que temos como meta (ver teorema universalidade da 5). Para designar este conjunto que constitui tanto estas regras do hardware, quanto o mecanismo abstrato que descrevemos no ponto anterior, usaremos o nome *unidade de controle finito*.

- **(...) que lê ou escreve um número finito de símbolos em uma lista de tamanho infinito:** escolhemos um alfabeto finito de símbolos, como por exemplo os caracteres binários $\{0, 1\}$, ou outros, como por exemplo todos os símbolos ASCII ou todos os mais de um milhão de símbolos Unicode. A fita pode ser pensada também como uma fita cacete, e o maquinista que descrevemos no item anterior como a cabeça de leitura da

fita cacete. Na analogia do hardware dos computadores atuais, esta fita seria a memória disponível, tanto a RAM quanto a HD, que como sabemos possuem mecanismos mais sofisticados de acesso do que a fita cacete.

2.1.2 Definição de Máquina de Turing

Prosseguiremos agora na definição do arsenal sintático necessário para a nossa definição de máquina de Turing. Aqui entram os elementos abstratos de nossa máquina, basicamente sustentada por uma teoria de conjuntos[2] como metateoria.[3]

Definição **1. Alfabeto:** Um *alfabeto* é qualquer conjunto finito

$$A = \{a_0, a_1, \ldots, a_n\},$$

com $n \geq 2$. Chamaremos seus elementos de *letras* (ou *símbolos*). Desta forma fica convencionado que a variável do tipo a_i varia sobre o alfabeto que estamos lidando, assim como b_i ao alfabeto B, e assim por diante.

Definição **2. Concatenação:** Definimos a operação binária de *concatenação* $(*)$ sobre um alfabeto A, tal que ela obedece as seguintes operações: sejam a_i, a_j, a_k letras do alfabeto A, e λ uma letra que denota o *string vazio* então:

i $a_i * a_j$ é uma concatenação.

ii $(a_i * a_j) * a_k = a_i * (a_j * a_k)$.

iii $a_i * \lambda = a_i = \lambda * a_i$.

No que se segue, o símbolo $*$ será suprimido. Assim, escreveremos $a_1a_2...a_n$ para $a_1 * a_2... * a_n$. Denotaremos a palavra resultante de uma concatenação de n strings a por a^n, isto é, : $a^0 = \lambda$ e $a^n = a^{n-1}a = aaa....a$ (n vezes).

Definição **3. Strings (palavras ou expressões):** Seja A um alfabeto, definiremos um *string* de A (ou, dito de outro modo, uma *palavra*) como o resultado de concatenarmos qualquer sequência de letras de $A \cup \{\lambda\}$. O conjunto formado por todos os strings possíveis de um alfabeto A é denotado por A^*. As variáveis que representam strings são denotadas por um traço superior: $\overline{a}$.[4]

Se, por exemplo, $A = \{a, b, c\}$, teremos:

$$A^* = \{\lambda, a, b, c, ab, ac, ba...\}$$

[2] Adotamos a teoria dos conjuntos por facilidade, porém é possível enfraquecer esta hipótese e mesmo trabalhar com lógica intuicionista ao invés de uma lógica clássica de fundo.

[3] Seguiremos nesta seção especialmente [Bek06] p.10, mas também os livros de [Cal02] p 1-2, Soare [Soa16], p.7-10, de Lewis e Papadimitriou [LP98], p.180-190, e de Li e Vitányi [LV19] p.28. As notações são uma miscelânea das apresentadas nestes livros.

[4] Isto é, estas duas definições podem ser resumidas: A^* é o conjunto fechado pela operação de concatenação sobre o conjunto $A \cup \{\lambda\}$, e esta operação forma um monoide com elemento neutro λ.

Observemos também que A^* é um conjunto infinito, mais precisamente infinito enumerável.[5] Strings servirão na nossa teoria tanto para representar os mecanismos da máquina de Turing, quanto para representar os mecanismos da computação. O string vazio λ será útil, por exemplo, quando quisermos dar à máquina "nenhum" input.

Definição **4. String Binário:** Definimos o conjunto $\mathcal{B} = \{0, 1\}$ como o *alfabeto de elementos básicos*. $\mathcal{B}^*$ é o conjunto dos *strings binários*. Como assumimos que nosso alfabeto tem pelo menos duas letras, podemos assumir sem perda de generalidade que sempre podemos convencionar[6] duas letras em um alfabeto A tal que $a_0 = 0$ e $a_1 = 1$.

Assim, $\mathcal{B}^* = \{\lambda, 0, 1, 00, 10, 11, 000, 100, 101, ...\}$. Utilizaremos os binários como alfabeto padrão onde a máquina de Turing operará. Daremos a definição central do mecanismo abstrato, o processador de nosso maquinário.

Definição **5. Máquina de -Turing(M.T.):** Uma máquina de Turing é uma dupla $\langle P, Q\rangle$, onde:

- Q é um conjunto finito de letras $\{q_0, q_1, q_2, ..., q_n\}$ que chamaremos estados. Chamamos q_0 de *estado inicial* e q_n de *estado final*;
- P é um conjunto finito contendo quádruplas.[7] As quádruplas são necessariamente da forma: (i) $q_i a_j a_k q_l$, (ii) $q_i a_j R q_l$, ou (iii) $q_i a_j L q_l$, onde q_i e q_l estão em Q, a_j e a_k estão em $\mathcal{B} \cup \{B\}$, R e L são letras que **não** estão em Q, e todas as quádruplas são tais que para todo $q_i a_j$ existe no máximo um $a_j q_l$ correspondente (pode ser que não exista nenhum).[8] Chamaremos P de *programa* da máquina, e suas quádruplas chamaremos de *instruções*.

Dada esta definição formal, queremos significar com cada quádrupla o seguinte:

1. com $q_i a b q_j$ se a máquina de Turing ler o símbolo a no estado q_i, então ela apagará o símbolo a, escreverá na mesma célula o símbolo b e irá para o estado q_j;
2. com $q_i a R q_j$ se a máquina de Turing ler o símbolo a no estado q_i, então ela moverá a cabeça da máquina à direita e irá para o estado q_j;
3. com $q_i a L q_j$ se a máquina de Turing ler o símbolo a no estado q_i, então ela moverá a cabeça da máquina à esquerda e irá para o estado q_j;

[5] Ver proposição 6.

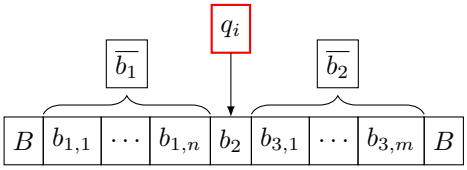
[6] Isso nos será útil para simplificar notação. Nos permitirá no meio de um raciocínio sobre um alfabeto genérico A, afirmar que $\mathcal{B} = \{0, 1\} \subseteq A$, e trabalhar com 0 e 1 como se fossem elementos de A. Notemos ainda que é sempre possível escolher estes elementos de A de forma a manter eventual ordem dos elementos de A, de modo que fixamos sempre que $0 < 1$. Ver a respeito disso a definição de ordem quase-lexicográfica 11.

[7] Strings de quatro letras.

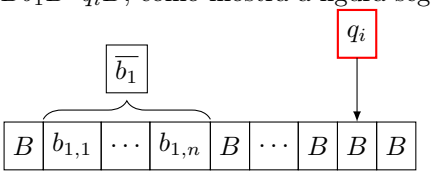
[8] Podemos generalizar esta definição adicionando um alfabeto não necessariamente binário para que a máquina de Turing possa trabalhar: $T = \{P, A, Q\}$. Além disso, note que P pode também ser pensado como uma função parcial $\delta : Q \times (\mathcal{B} \cup \{B\}) \to (\mathcal{B} \cup \{B, R, L\}) \times Q$.

Notemos que a definição de máquina de Turing é incompleta se pensada sozinha, ainda não nos referimos à fita. Para isto precisaremos das duas definições a seguir, de configuração e passo, que mostram o que a máquina T pode fazer com a sua fita. Basicamente é ter o poder de escrever, apagar e mexer nas células da fita. Isto estará implícito na notação que daremos a seguir para configuração e passo.

DEFINIÇÃO **6. Configuração:** Seja $T = \{P, Q\}$ uma máquina de Turing, definimos a *configuração* C de T no estado q_i como um elemento de $(\mathcal{B} \cup \{B\} \cup Q)^*$ da forma $\overline{b_1} q_i \overline{b_2}$, onde $\overline{b_1}, \overline{b_2}$ são strings em $(\mathcal{B} \cup \{B\})^*$. Chamamos $\overline{b_1 b_2}$ de *string significativo da fita*. Além disso, nesta configuração dizemos que a máquina T está com *a cabeça da máquina* sobre o primeiro símbolo de $\overline{b_2}$ no estado q_i (caso $\overline{b_2}$ seja o string vazio, dizemos que é B).



Notemos que nesta definição a máquina pode estar na configuração $Bq_iBB^n\overline{b_1}B$ ou $B\overline{b_1}B^nq_iB$, como mostra a figura seguir.

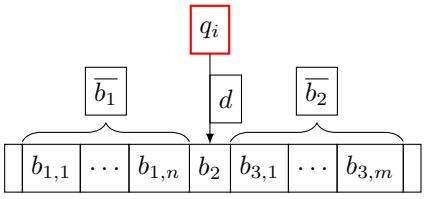


DEFINIÇÃO **7. Passo:** Seja $T = \{P, Q\}$ uma máquina de Turing e C_1 e C_2 duas configurações. Dizemos que C_2 é o próximo passo de C_1, em símbolos:

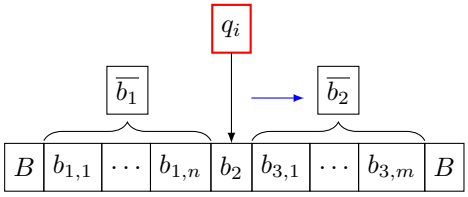
$$C_1 \longrightarrow C_2$$

se umas das condições se seguem (d denota uma letra qualquer do alfabeto):

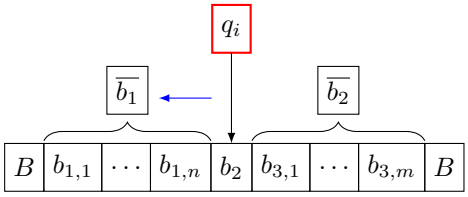
1. $C_1 = \overline{b_1} q_i b_2 \overline{b_3}$ e existe uma instrução $q_i b_2 d q_j \in P$, caso no qual $C_2 = \overline{b_1} q_i d \overline{b_3}$;



2. $C_1 = \overline{b_1} q_i$ e existe uma instrução $q_i B d q_j \in P$, então $C_2 = \overline{b_1} q_j d$;
3. $C_1 = \overline{b_1} q_i b_2 \overline{b_3}$ e existe uma instrução $q_i b_2 R q_j \in P$, então $C_2 = \overline{b_1} b_2 q_i \overline{b_3}$;



4. $C_1 = \overline{b_1}q_i$ e existe uma instrução $q_iBRq_j \in P$, então $C_2 = \overline{b_1}Bq_i$;
5. $C_1 = \overline{b_1}b_2q_ib_3\overline{b_4}$ e existe uma instrução $q_ib_3Lq_j \in P$, então $C_2 = \overline{b_1}q_ib_2b_3\overline{b_4}$;



6. $C_1 = \overline{b_1}b_2q_i$ e existe uma instrução $q_iBLq_j \in P$, então $C_2 = \overline{b_1}q_jb_2$.
7. $C_1 = q_ib_1\overline{b_2}$ e existe uma instrução $q_ib_1Lq_j \in P$, então $C_2 = q_iBb_1\overline{b_2}$.
8. $C_1 = q_i$ e existe uma instrução $q_iBLq_j \in P$, então $C_2 = q_jB$.

Dadas estas duas definições, agora estamos prontos para pensar como a máquina começa a computar e pára — ou então, pior, nunca pára. Todo esse processo definirá onde começamos e até onde seguiremos os passos contidos nas instruções do hardware de nossa máquina de Turing.

Definição **8. Configuração Inicial:** Seja $T = \{P, Q\}$ uma máquina de Turing. Um *input* é um string $\overline{b} \in \mathcal{B}^*$. A *configuração inicial* é a formada por $q_0\overline{b}$, onde $\overline{b}$ é um input.

Definição **9. Estado acessado e parada:** Dizemos que uma configuração C_2 (ou configuração) é *acessada* se existe uma configuração C_1 e um passo tal que

$$C_1 \longrightarrow C_2.$$

Defina $\xrightarrow{*}$ como o fecho[9] transitivo e reflexivo de $\longrightarrow$. Dizemos que a máquina de Turing T *gera* a configuração C se

$$\xrightarrow{*} C.$$

Uma máquina T *pára no estado* q_k se T consegue acessar o estado q_k e não há outras configurações a acessar.

[9] Ver definição fecho p. 129, o conjunto de todas as configurações que são *acessadas* pela M.T.

DEFINIÇÃO **10. Computação** Dizemos que uma máquina de Turing T *computa* o string significativo $\overline{c}$, correspondente à configuração C, se a máquina gera C, em algum estado q_f, e se a máquina pára neste estado (que chamamos de final). Se T computa $\overline{c}$, então o string significativo resultante $\overline{c}$ é chamado de *output* (ou saída). Usamos a notação $T(\overline{a}) = \overline{b}$ para dizer que uma máquina de Turing gera o output $\overline{b}$ dado um input $\overline{a}$. Utilizamos o símbolo $T\downarrow$ para denotar que a máquina T pára (ou *converge*) e $T\uparrow$ caso ela nunca pare (ou *diverge*).

2.1.3 Exemplo e Composicionalidade

Por exemplo, seja $T = \{P, Q\}$ tal que:[10]

$$
\begin{aligned}
Q =& \{q_0, q_1, q_2, q_3, q_4\} \\
P =& \; q_0 0 R q_0,\; q_0 1 R q_0,\; q_0 B L q_1 && \text{(“Pula cela” até o primeiro B.)} \\
& \; q_1 B 0 q_4,\; q_1 01 q_4 && \text{(O sucessor sem resto.)} \\
& \; q_1 10 q_2\; q_2 0 L q_3,\; q_3 10 q_2,\; q_3 01 q_4,\; q_3 B 0 q_4 && \text{(Levando o resto.)}
\end{aligned}
$$

É possível ver que T satisfaz as condições de uma máquina de Turing para uma fita qualquer. Seja o input desta máquina de Turing o string 11. Vejamos o que ela faz. Para isto usaremos uma tabela 2.1 na qual associaremos cada instrução a cada configuração da máquina de Turing.

nº de passos	instrução	configuração (o string significativo)
0	$q_0 1 R q_0$	$q_0 11$
1	$q_0 1 R q_0$	$1 q_0 1$
2	$q_0 B L q_1$	$11 q_0$
3	$q_1 10 q_2$	$1 q_1 1$
4	$q_2 0 L q_3$	$1 q_2 0$
5	$q_3 10 q_2$	$q_3 10$
6	$q_2 0 L q_3$	$q_2 00$
7	$q_3 B 0 q_4$	$q_3 B 00$
7		$q_4 000$

Tabela 2.1: máquina que faz o sucessor de um número

Adiantando o assunto um pouco, dado um input que é uma sequência de uns, ela acha o próximo número na ordem lexicográfica (veremos este conceito na definição 11). Ou seja, ela computa a função sucessora se definirmos funções computáveis através desta ordem (ver 16). Cada passo é unicamente determinado, em coerência com o que esperávamos da definição de passo. Por exemplo, da segunda para terceira linha:

$$11 q_0 \longrightarrow 1 q_1 1,$$

[10] A máquina a seguir computa o sucessor, mas **não** na representação binária padrão, na codificação que daremos a seguir.

pois na instrução correspondente à configuração da segunda linha temos um L, caso no qual, movemos a cabeça da máquina para a esquerda e trocamos q_0 por q_1.

Como $\stackrel{*}{\to}$ é o fecho transitivo, e todos os passos estão corretos, temos que

$$\stackrel{*}{\to} q_4000,$$

e a máquina de Turing computa esta última configuração. Como o string significativo é 000, temos que ele será o output. Logo, $T(11) = 000$.

Com este exemplo observamos uma coisa. Gostaríamos de falar desta máquina como uma função parcial, porque queremos cobrir o fato de que a máquina pode rodar para sempre, caso no qual a função não estará definida para este valor. E mais, interpretar esta máquina como função que opera com o conjunto de números naturais, já que nossa pretensão é vermos como esta noção de máquina opera no contexto matemático. Temos que estabelecer uma representação dos símbolos de uma máquina de Turing em termos dos nossos familiares números naturais. E isto faremos na seção seguinte.

2.2 Funções Parciais Computáveis e Códigos

Na sessão passada falamos apenas de strings, letras, e como mecanizar estes elementos primordiais. Não mencionamos, até agora, em nenhum momento, como associamos números a esses strings. Queremos falar de funções computáveis como as funções que existem em uma calculadora: $+, \times, exp, \log$. Ou seja queremos trabalhar não apenas com uma função parcial computável definida em strings: $\phi : (\mathcal{B} \cup \{B\})^* \to (\mathcal{B} \cup \{B\})^*$, mas uma função p.c. numérica: $\phi : \mathbb{N}^n \to \mathbb{N}$.

Mas antes de fazer isso, façamos um interlúdio para discutir as possibilidades de como isso pode ser feito.

O que é óbvio em todos os casos é que precisamos traduzir o resultado que aparece como input ou output de uma máquina de Turing. Um jeito fácil de fazer isso, e que é feito na maioria de livros sobre computabilidade, por exemplo nos livros [BJB12], [CE09] e [Odi99], é simplesmente contar o números de uns no input e output.[11] Uma outra alternativa é lermos o input e o output em ordem lexicográfica, como é feito nos livros específicos sobre complexidade, por exemplo em [LV19][12] e [Cal02].

Aqui adotaremos a segunda alternativa, a representação lexicográfica.

[11] Mais precisamente: Uma função parcial conta-um $\phi(x_1, ..., x_n) = y$ seria computável por uma máquina de Turing T se ela é tal que $\phi : \mathbb{N}^n \to \mathbb{N}$, e as seguintes condições acontecem:

1. se o input da máquina de Turing T é da forma $B\overline{x_1}B\overline{x_2}....B\overline{x_n}B$, onde cada $\overline{x_i}$ é um string de $x_i + 1$ uns;
2. e além disso, o output é da forma $B\overline{y}B$, sendo tal que $\overline{y}$ é também um string de $y + 1$ uns.

[12] Porém, Li e Vitaní usa uma representação lexicográfica binária autolimitada.

Por quê? Isto pode parecer completamente arbitrário. Mas deste modo criaremos uma medida de complexidade já, de início, eficiente. Adiantando um pouco, isso repercutirá essencialmente na facilidade de provarmos o limite superior da função complexidade que daremos no corolário 20. Ou seja, como a representação binária é mais curta do que o método de contagem de uns, isto impactará no menor tamanho de um programa que gera um determinado string.[13]

Mas, independente disso, os resultados da teoria seriam essencialmente os mesmos. Pois podemos criar um subprograma que, por exemplo, converte inputs do tipo bloco de uns para inputs do tipo que utilizaremos aqui. E vice-versa. Ou seja, os resultados dispostos nos livros destes autores se transferem para cá, e os nossos, para lá. E, de forma geral, isto se aplica a qualquer definição de função parcial computável que se comporte de forma parecida com estas que citamos.

Assim, resumindo, as seguintes coisas serão importantes:

- Construirmos uma função total bijetora $bin : \mathbb{N} \to \mathcal{B}^*$ que nos permitirá falar de strings binários como se fossem números, e inversamente para $bin^{-1} : \mathcal{B}^* \to \mathbb{N}$, para falar de números como se fossem strings binários;
- Construirmos uma função total bijetora chamada numeração de Gödel tal que $g : A \to \mathbb{N}$, ou seja que transformará qualquer letra em um número, e por extrapolação qualquer palavra, sequência ou texto como se fossem números; (observação: a diferença deste item para o anterior é que estamos falando de um alfabeto qualquer, não apenas o alfabeto $\mathcal{B} = \{0, 1\}$.)
- Outras codificações do tipo $f : \mathbb{N}^n \to \mathbb{N}$.

O primeiro ponto é uma base para chegarmos no segundo ponto, pois utilizaremos a função *bin* para passar de strings binários para números em geral.

O segundo ponto é uma condição para mostrarmos como podemos construir uma máquina de Turing universal. Isto porque queremos mostrar que todos os componentes de uma máquina de Turing, os estados e o programa, podem ser codificados em símbolos que uma máquina de Turing universal entende. E é, em resumo, o que um computador faz quando precisa "entender" as linguagens mais complexas com que trabalhamos.

Por exemplo, este texto. Ele está sendo escrito em um conjunto de caracteres em um editor de texto. Estes símbolos que usamos para a comunicação humana não constitui o alfabeto com que este computador foi composto. Usamos os números binários zero e um para simbolizar as mudanças de corrente elétrica, e esta linguagem seria o mais próximo do "alfabeto" que este computador consegue processar. Porém, para que o computador entenda o que estamos dispondo neste editor de texto agora, ele tem que de alguma forma codificar estes caracteres e transformá-los em strings binários (ou equivalentemente números binários).

[13] Enquanto na contagem de uns temos $|x| = x$ na outra temos $|x| = \log x$, como veremos à frente.

Há vários modos de convertermos strings em números. Ficou convencionado pela tradição que uma codificação que serve este propósito é chamada numeração de Gödel. A conversão para strings binários poderia ser feita em código Morse,[14] por exemplo, porém esta codificação é complicada, apesar de ser eficiente probabilisticamente. Poderíamos pensar em números naturais em uma base prima qualquer, e considerar cada letra do alfabeto que pretendemos trabalhar como um número diferente desse sistema numérico, é o que faz Smullyan [Smu92] p. 23. Poderíamos representar cada símbolo por um número de tal forma que palavras sejam sequências de símbolos que podem ser decodificadas por uma função beta baseada no teorema Sun Tzu do Resto, como foi originalmente pensado por Gödel, isto é explorado em [BJB12] p. 247, e pode ser considerado a codificação mais padrão para se provar a incompletude.

A escolha da codificação não é importante de maneira geral para estabelecer os teoremas principais da teoria da computabilidade. Pois veremos mais a frente no teorema 11 de Rogers que os resultados gerais se seguem independentes da codificação adotada (desde que ela sirva para provar o teorema da enumerabilidade e parâmetro): códigos lexicográficos, os números originais de Gödel e sua função beta, todos são equivalentes para nossos propósitos. Soluções mais elegantes e menos elegantes, com mais ou menos requerimentos, são possíveis. Porém, não nos deteremos em detalhes a respeito disso.

Adotaremos uma codificação lexicográfica de comprimento fixo $\ulcorner \cdot \urcorner$, qualquer das codificações citadas funcionaria tão bem quanto, esta codificação é computável, se baseia na possibilidade de ordenar strings de um determinado tamanho, como veremos a seguir.

2.2.1 Ordem e Funções Parciais Computáveis

Seguiremos Calude [Cal02] p.2.

Definição **11. Ordem Quase-Lexicográfica (ou ordem do dicionário):** Seja um conjuntos de letras A com uma ordem total estrita $<$, dizemos que esta ordem *induz uma ordem quase-lexicográfica* das palavras em A^* se, dado que a ordem total estrita é $a_1 < a_2 < ... < a_n$ no conjunto A, ordenamos os strings de A^* da seguinte forma:

$$\begin{aligned}&\lambda < a_1 < ... < a_n < \\ &< a_1a_1 < ... < a_1a_n < a_2a_n < ... < a_na_1 < ... < a_na_n < ... \\ &< a_1a_1a_1 < ... < a_na_na_n < ...\end{aligned}$$

Além disso fixamos que, em $\mathcal{B}$, sempre temos $0 < 1$ (nos referiremos a esta ordem também por ordem lexicográfica *tout court*).

Mas precisamos fazer uma distinção cuidadosa. Quando falamos de representar um numero decimal na base numérica binária nos referimos à seguinte

[14] Versão numérica binária do código Morse, basta colocar um 1 a mais na frente de todos os strings e pensá-lo como representação binária.

função cujo domínio é a representação dos números naturais em base decimal e a imagem é a representação em base binária: $\{(0,0),(1,1),(2,10),(3,11),...\}$. Esta representação não é de strings, mas sim de representantes de números, neste sentido $01 = 1$, o que não aconteceria no caso de strings. Não podemos usar esta ideia para fazer uma bijeção $g : \mathbb{N} \to \{0,1\}^*$ porque, caso contrário, como 01 e 1 são os mesmos números, teríamos $(1,1)$ e $(1,01)$, e a função estaria mal definida.

Como faremos então? Uma maneira é fazer uma bijeção dos elementos ordenados pela relação 'menor que' do conjunto dos inteiros positivos aos elementos ordenados quase-lexicograficamente do conjunto $\mathcal{B}^*$. Ou seja, fazer a seguinte bijeção que estipularemos na próxima definição:

Definição **12. Representação binária:** A representação lexicográfica binária de um número natural n será denotada pela função bijetora $bin : \mathbb{N} \to \mathcal{B}^*$ e decorre do pareamento dos números inteiros positivos (ordenados pela relação 'menor que') com os elementos de $\mathcal{B}^*$ ordenados pela ordem lexicográfica induzida.

Ou seja, $bin(n)$ é dada da forma a seguir:

$$\{(0,\lambda),(1,0),(2,1),(3,00),(4,01),(5,10),(6,11),(7,000),...\}$$

Assim, por exemplo, $bin(6) = 11$.

De modo heurístico, podemos pensar que $bin(n)$ será a expansão em base binária tradicional (i.e. $\{(0,0),(1,1),(2,10),(3,11),...\}$) do número $n+1$ após 'apagarmos' o primeiro algorismo da esquerda para direita. Ou seja, dado n um número natural, a expansão binária de $n+1$ será:

$$n+1 = 2^{k+1} + a_k 2^k + a_{k-1}2^{k-1} ... + a_1 2^1 + a_0 2^0 = 2^{k+1} + \sum_{x=0}^{k} a_x 2^x,$$

onde $a_i \in \{0,1\}$ são os dígitos da expansão binária tradicional de $n+1$, então $bin(n) = a_k ... a_0$ e k é o tamanho do número $n+1$ em sua expansão binária tradicional.

Por exemplo, se quisermos saber a representação lexicográfica binária de 16, fazemos a representação binária tradicional do número 17, isto é 10001, e tiramos o primeiro algorismo um da esquerda para direita, o que nos dará $bin(16) = 0001$.

Definição **13. Tamanho de um string:** A função que nos dá o tamanho de um string em A^* será indicada por $|x|$ e definida no domínio A^* e contradomínio nos números naturais, em especial $|\lambda| = 0$, e tal que $|a_1 ... a_n| = n$ para $a_i \in A$. Definamos também A^n conjunto de todos strings de tamanho n. O tamanho de um string binário possui uma nomenclatura de unidade de medida especial que estipularemos *bits*.

Por exemplo $|aaa| = 3$, $|a^n| = n$, etc...

É possível saber o tamanho do string que está no n-ésimo lugar da ordem lexicográfica que demos acima. Mas, para deixar claro, a nomenclatura da seguinte função log é usada de maneira peculiar neste contexto, ligeiramente diferente da usual e é standard na área. Salientaremos isso aqui, já que usaremos a seguir:

DEFINIÇÃO **14.**

$$\log(n) = \lfloor \log_2(n+1) \rfloor$$

PROPOSIÇÃO **2.** ***Tamanho do n-ésimo string binário:***

$$|bin(n)| = log(n).$$

Demonstração. Seja o tamanho de $bin(n) = k$, logo pelo método heurístico para achar $bin(n)$, que estipulamos acima, $n + 1 = 2^k + \sum_{k=0}^{k-1} a_x 2^x$, onde a_xs são dígitos da expansão binária tradicional de $n + 1$ (i.e. são dígitos 0 ou 1). Portanto:

$$\log(n) = \lfloor \log_2(n+1) \rfloor = \lfloor \log_2(2^k + \sum_{x=0}^{k-1} a_x 2^x) \rfloor = \log_2 2^k = k.$$

A somatória desaparece porque ela é menor que 2^k, e a função piso a elimina. □

A comparação entre números e representação binária é tão próxima que coincidiremos o sentido de número inteiro positivo ($\mathbb{N}$) e sua representação binária ($\mathcal{B}^*$). Assim usaremos da ambiguidade para falarmos coisas do tipo: $000 = 7$ e $|7| = 3$. Obviamente tem-se que ter em consideração que na realidade estamos falando $000 = bin(7)$ e $|bin(7)| = 3$.

DEFINIÇÃO **15. Notações e convenção sobre a identificação entre strings e números:** Escrevemos ambiguamente $x \in \mathbb{N}$ para denotar $bin(x) \in \mathcal{B}^*$, e vice-versa. Além disso: $|x_i| = n$ para $|bin(x_i)| = n$. A não ser que o contexto exija uma definição mais precisa.

No que se segue não utilizaremos mais a notação $\overline{a}$ para strings a não ser que seja necessário.

Assim chegamos à definição importante de função *parcial* computável por máquina de Turing. Notemos que é importante que a função seja parcial e não total, isto é, ela *pode* não estar definida para todos os elementos do domínio, ao contrária da função *total*, que necessariamente estará definida em todo o domínio. Porque queremos englobar também com isso os casos em que a máquina de Turing não pára, ou seja queremos englobar as funções parciais que não estão definidas em determinados elementos do domínio.

Para explicar um pouco melhor este ponto, tomemos, por exemplo, a função divisão $\div : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ definida nos números racionais. Ela não está definida no ponto 0. Para solucionarmos este problema podemos tirar este ponto do

domínio $\div : \mathbb{Q} \times (\mathbb{Q} - \{0\}) \to \mathbb{Q}$, então teremos uma função total, mas também podemos simplesmente dizer que esta função é uma função parcial que não está definida neste ponto 0. Programas da mesma forma podem ser pensados como funções, podemos tirar todos os pontos de indeterminação para garantir que todos os programas serão totais, ou seja sempre pararão. Por exemplo, podemos tornar qualquer linguagem de programação total ao estipular que cada programa gerado por esta linguagem deve parar em um determinado número de passos (por exemplo, se o programa passar de um mês rodando e não parar, então forçamos que pare e dê algum resultado parcial). Em geral os programas gerados pela maioria das linguagens de programação não vêm com esta parada forçada, ou seja, eles continuam a rodar indefinidamente, e, quando sabemos que ele entrou em um loop infinito, dizemos que este programa, como função parcial, não está definido para o input em questão. Será importante pensarmos em máquinas deste ponto de vista parcial e não total, porque é exatamente o caso de determinar quando uma máquina não pára em geral que é a fonte de problemas como o problema da parada 8.

Na teoria da computabilidade essa diferença entre *função computável parcial* e *função computável total* (ou função computável *tout court*) é crucial. Notemos que apesar disso, do ponto de vista puramente conjuntista as funções totais são apenas um subconjunto das funções parciais, a saber, aquelas funções parciais que estão definidas em todo domínio. Do ponto de vista da computabilidade no entanto, esta diferença é grande, e como veremos a teoria vai gerar resultados diferentes para esses dois conjuntos de funções, o que se deve, entre outras coisas, ao problema da parada e a dificuldade de diagonalizar funções parciais.

DEFINIÇÃO **16. função parcial computável:** Seja $T = \langle P, Q \rangle$ uma máquina de Turing. Dizemos que $\phi(x_1, ..., x_n)$, $\phi : \mathbb{N}^n \to \mathbb{N}$, é uma função parcial *computada* por T se, para x_i números naturais

$$\phi(x_1, ..., x_n) = bin^{-1}\big(T\big(bin(x_1)\, B\, bin(x_2)\, B\, ...\, B\, bin(x_n)\big)\big),$$

caso T compute este input (escrevemos $\phi\downarrow$) e indefinida caso contrário ($\phi\uparrow$).[15]

2.2.2 Números de Gödel

Usaremos a codificação lexicográfica de tamanho fixo como nossa numeração de Gödel. Como falamos, a escolha de uma numeração específica não é importante no que se segue.

Podemos pensar nesta ideia da seguinte maneira. Imaginemos que nosso alfabeto tem três letras $\{a, b, c\}$. Um jeito de codificar este alfabeto seria atribuirmos um string binário de tamanho dois para cada letra. Por exemplo, 00 para a, 01 para b, 10 para c. Por que dois dígitos para cada letra? Porque se escolhermos uma codificação do tipo $(a, 0)$, $(b, 1)$ e $(c, 10)$, não saberíamos dizer se 1001 é o código de *cab* ou *baab*. Já se escolhermos a primeira codificação,

[15] Note que ϕ pega o output de T e converte em números naturais novamente, e T pega números convertidos em strings. Convencionamos que o input de mais de uma variável serão separados por Bs. Ou seja, a definição faz sentido.

podemos dividir o código em grupos de dois dígitos e daí saberemos que apenas uma letra será determinada para cada grupo. 1001 será *ba* (além disso com esta codificação saberíamos que codificações de tamanho ímpar não podem ser codificações válidas).[16] Este tipo de codificação se chama na literatura *codificação de tamanho fixo*. E expor um tipo de codificação de tamanho fixo que nos seja útil será um dos objetivos dessa seção.

Este tipo de codificação acontece diariamente na comunicação do computador com o usuário. A codificação ASCII original foi desenvolvida como uma padronização para comunicação à distância para o alfabeto latino das palavras inglesas na década de 60. Ela originalmente trabalha com uma codificação em um alfabeto binário de comprimento fixo de sete dígitos. O que permite codificar $2^7 = 128$ caracteres. Se tivéssemos uma letra a mais além destas disponíveis pela ASCII teríamos que usar uma codificação de oito dígitos. E é o que acontece com a codificação para a língua portuguesa, precisamos de uma codificação ASCII estendida. Pelo mesmo raciocínio concluímos que se tivermos um alfabeto A de n letras, então o menor tamanho possível de strings desta codificação binária de tamanho fixo terá que ser $\log(n)$. Mas alguém poderia reclamar que a cultura humana é vasta, e que ela possui muitos mais símbolos do que os usados pelos alfabetos centralizados nas línguas latinas. Talvez, tragicamente, seja possível dizer que o comportamento exponencial das codificações binárias seja ainda mais vasto. Com aproximadamente 32 bits muitos dos símbolos de culturas do presente e do passado podem ser codificados em uma codificação única, chamada de Unicode,[17] e ainda possuir espaço livre para futuros novos alfabetos.

Assim, generalizando esta temos a seguinte definição:

Definição **17. Codificação lexicográfica de tamanho fixo** Seja A um alfabeto, disponha em ordem lexicográfica as letras de A, $a_1 < a_2 < ... < a_n$, e seja uma função bijetora $g : A \to \mathcal{B}^*$ de todas estas letras assim ordenadas nas palavras também ordenadas lexicograficamente (pela ordem induzida das letras de B) de $\mathcal{B}^*$ de tamanho $\log |A|$.[18] Para todo $a_i \in A$, a concatenação pode ser feita da seguinte maneira:

$$g(a_1...a_n) = g(a_1)...g(a_n)$$

Assim, usamos a nomenclatura:

$$\ulcorner \overline{x} \urcorner = g(\overline{x}).$$

E chamamos a codificação g ou $\ulcorner \cdot \urcorner$ assim caracterizada de codificação *lexicográfica de tamanho fixo* ou *números de Gödel*.

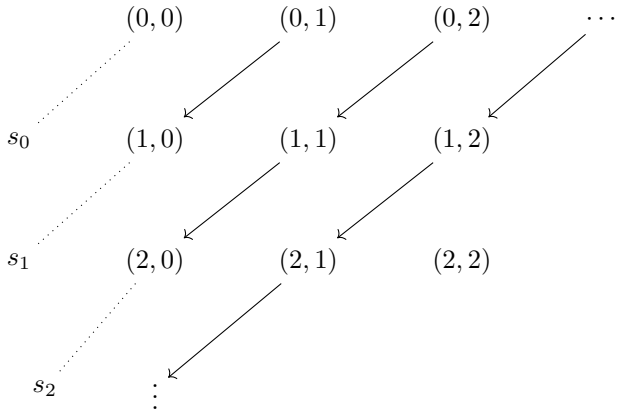
[16] Porque toda letra é associada a grupos de dois símbolos, de forma que palavras codificadas possuem número par de letras.

[17] Embora esta codificação não é de tamanho fixo, como por exemplo a UTF-8. Atualmente, talvez devido a internet, este seja o código que está caminhando para o padrão internacional.

[18] Isto é possível porque temos pelo menos $|\mathcal{B}|^{\log |A|} = |A| + 1$ palavras em $\mathcal{B}^*$ de tamanho $\log |A|$.

2.2.3 De várias dimensões à reta

Codificação de pareamento standard: Agora falaremos como é possível codificar bijetivamente pares ordenados de naturais nos naturais i.e. $f : \mathbb{N}^2 \to \mathbb{N}$. Construiremos a bijeção a partir da definição de ordem lexicográfica. Disponha os pares de números naturais da seguinte maneira:



Para cada setas representadas na figura temos uma soma parcial $s_0, s_1, ..., s_n$. Para todo (x, y) elemento das setas, temos que $x + y = n = s_n$.

Logo, podemos criar uma ordem induzida no conjunto de duplas ordenas $\mathbb{N}^2$ ordenando lexicograficamente os elementos de cada linha:

$(x, y) < (x', y')$ sse ou $x + y < x' + y'$, ou $x + y = x' + y'$ e neste caso $x < x'$.

Além disso podemos dar uma fórmula fechada para a função de pareamento $c : \mathbb{N}^2 \to \mathbb{N}$. Basta observar que o total de duplas ordenadas anteriores a s_n é a soma $1 + 2 + ... + n$. Logo a posição na s_n seta será determinado por x:

$$c(x, y) = (1 + 2 + .. + (x + y)) + x = \frac{(x + y)(x + y + 1)}{2} + x = \frac{(x + y)^2 + 3x + y}{2}$$

Logo isto enseja a seguinte definição:

Definição **18. Codificação de pareamento standard :** Defina a *codificação de pareamento standard* por:

$$c(x, y) = \frac{(x + y)^2 + 3x + y}{2}$$

Além disso para qualquer sequência $(x_1, ..., x_n)$ de números naturais defina:

$$c(x_1, ..., x_n) = c(x_1, c(x_2, c(...c(x_{n-1}, x_n)...)$$

No que se segue, podemos assumir sem perda de generalidade que todos os inputs das máquinas de Turing são de uma variável. A não ser que seja importante estabelecer uma distinção.

Capítulo 3

Longe da Máquina

> No capítulo anterior falamos sobre um modelo concreto de computação. Este modelo possui uma analogia física muito clara, a de uma fita e um processador fazendo operações muito simples.
>
> Neste capítulo vamos gradualmente nos afastar deste modelo até acharmos um modelo mais abstrato que nos permita construir de forma simples uma máquina de Turing universal.

Não é difícil imaginar uma máquina realizando o procedimento de duas ou mais máquinas. De fato, o desenvolvimento de computadores e celulares que possuem cada vez mais funcionalidades parece ser um exemplo definitivo de que isso é possível. Não nos é estranho por exemplo imaginar que possamos ter todas funcionalidades de uma calculadora, de um relógio, de um editor de texto e de outras máquinas reunidas em uma máquina só. Um resultado um pouco menos intuitivo é o de que quaisquer máquinas que existam ou possam vir a existir podem ser reunidas em uma máquina só. Mas mesmo aí a nossa experiência com máquinas com uma capacidade potencialmente infinita de programação, como os computadores, parece indicar que, se não nos impedissem os limites físicos do hardware e do tempo, nada nos impediria de pensar em um computador que consiga potencialmente programar todos possíveis programas. A versão formal desta ideia é chamado de teorema da universalidade, em que provamos que existe uma função parcial computável universal.

3.1 Enriquecendo a máquina: o programa-while

3.1.1 Composição

Uma qualidade que queremos que nossa máquina de Turing tenha é a composicionalidade. Isso vai nos permitir emendar máquinas de Turings uma nas outras, como se fossem peças, para exibirmos a possibilidade da existência de máquinas de Turings cada vez mais complexas sem ter que de fato defini-las a partir da nossa definição inicial por quádruplas. Provaremos a seguir esta propriedade.

PROPOSIÇÃO **3.** ***Composição de máquinas de Turing:*** *Seja T_1, ..., T_n máquinas de Turing que computam as funções p.c. $\phi_1, ..., \phi_n$. Então existe uma máquina de Turing T que computa a função ϕ tal que ela é a composição:*

$$\phi(x) = \phi_n \circ ... \circ \phi_1(x)$$

Provaremos por indução.

De fato, seja $n = 1$, temos que a afirmação vale trivialmente. Suponha que já temos uma ψ que faça o trabalho pedido pelo teorema para T_{n-1} e temos uma máquina de Turing T_n cuja função p.c. correspondente é ϕ_n.

Por um lado, vejamos que tal composição de funções parciais é possível. Podemos fazer com que o output de ψ seja o input de ϕ_n e também notemos que quando compomos desse modo a funções, as máquinas de Turing correspondentes começarão a computação na primeira casa à esquerda de todos os strings que não estão em branco, pela definição de computação. Logo, o output calculado pelas funções p.c. destas duas máquinas de Turing será $\phi_n \circ \psi$.

Por outro lado construamos a máquina de Turing que faz tal composição. Suponha que a máquina de Turing T_{n-1} possui estado final q_i, e suponha que a máquina T_n possui o estado final q_j. Em primeiro lugar, apague todas as quádruplas que não são acessadas pelas instruções de T_{n-1}. Crie uma máquina auxiliar X com instruções que reconduzem a cabeça da máquina para a primeira letra do string significativo do output. Sempre é possível fazer tal máquina com um número x de estados e unir as instruções ao fim das instruções da máquina T_{n-1}. Faça uma máquina de Turing T que é a união das quádruplas de T_{n-1}, X e T_n, renumerando os estados convenientemente. Ou seja, de forma que o estado inicial q_0 de T seja o estado inicial de T_{n-1}; o estado inicial de de X seja o estado q_{i+1} de T; e o estado inicial e final de T_n sejam os estado q_{x+i+1} e $q_{x+i+j+1}$ respectivamente de T. Logo, existe uma máquina de Turing T que computa a função ϕ tal que $\phi = \phi_n \circ \psi$. E isto completa a prova por indução. □

3.1.2 Máquinas while

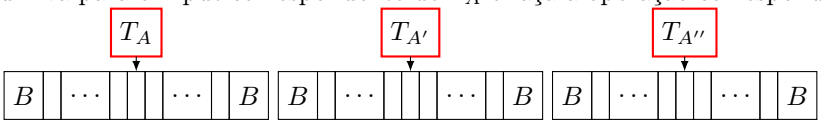
Uma consequência destas definições e este método composicional para achar novas máquinas de Turing é que podemos formar máquinas de Turing que fazem operações cada vez mais próximas daquelas que achamos em linguagens de programação usuais. Podemos emendar uma máquina na outra até podermos formar operações aritméticas, definir os racionais, definir operações de manipulação de strings, e muitas coisas mais. Porém, ainda não podemos falar de uma única máquina de Turing programável que faz tudo isso, por mais que esta composicionalidade já enseje a ideia de que máquinas de Turing menos gerais funcionam como sub-programas de máquinas de Turing que as englobam. Ainda estamos falando das propriedades do hardware das máquinas de Turing, a passagem para um ponto de vista de fato de programação apenas pode ser feita no momento em que identificarmos hardware (instruções de máquinas de Turing) com software (o input de uma máquina de Turing): ou seja, quando provarmos

que há uma máquina de Turing que tem o poder de simular outras máquinas de Turing. Isto é o que faremos na seção em que provamos que existe uma máquina de Turing universal.

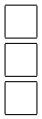
Um caminho que poderíamos seguir é provar explicitamente que existem máquinas de Turing cada vez mais 'potentes', ou seja, que podem fazer cada vez mais operações, todas unidas em um dispositivo só, até estarmos prontos para mostrar que também temos o poder de criar uma máquina de Turing que simula outras máquinas de Turing. Porém fazer isso explicitamente demoraria mais tempo do que gostaríamos. Poderíamos até enunciar a próxima proposição como um exercício para o leitor, porém seria um exercício descomunalmente grande, e não queremos com isto desconstruir a própria definição de 'exercício para o leitor' (embora alguns dos itens sejam razoáveis). Por esta razão, vamos apenas indicar que este caminho existe, e é o que é de fato feito no livro de Jeffrey, Boolos e Burgess, *Computabilidade e Lógica*, [BJB12]. Apenas ilustraremos intuitivamente o resultado.

Basicamente a construção depende de mostrar que é possível criar uma máquina de Turing que consegue administrar espaços da fita, fornecendo assim a familiar noção de espaço de memória aleatoriamente acessada (o que parece muito mais com a nossa atual arquitetura de nossos computadores).

Vejamos o seguinte desenho. Nele representamos três máquinas de Turing T_A, $T_{A'}$ e $T_{A''}$. Podemos pensar nessas três máquinas como se fossem independentes, ou seja com definições separadas, ou podemos imaginar que todas elas estão compostas em uma única máquina de Turing T que de alguma forma consegue manter separado por Bs o conteúdo significativo que cada máquina de Turing opera. Intuitivamente, não é difícil ver que esta administração de espaços é possível, basta ver que é possível fazer um programa que recorta e cola o conteúdo de uma fita da máquina de turing, arbitrariamente para direita ou para esquerda. Assim, suponha que queiramos utilizar a máquina T_A com input x, basta para isto que a máquina T que compõe estas máquinas coloque suficientemente a esquerda este input,[1] e que toda operação de T_A seja antecedida de um vá para o input correspondente de T_A e faça a operação correspondente.



A descrição acima, embora insuficiente, nos indica que é possível olhar para as fitas acima de maneira mais simplificada e esquemática. Podemos pensar que cada fita acima são informações que podemos guardar em uma tabela:



[1] Toda vez que digamos as operações de T_A forem misturar o conteúdo da fita com a fita de $T_{A'}$, colocamos a fita de T_A mais para esquerda.

Por exemplo, suponha que a máquina T_A é a função que dado um input i que representa um número nos dá o sucessor deste número $i+1$, isto é $T_A = suc$. Suponha que $T_{A'} = +$ é a função que dado dois número n e m retorna a soma $n + m$. Suponhamos que $T_{A''} = id$, isto é a função identidade.

A máquina T pode utilizar estas três funções da forma que for mais conveniente, simplificando cálculos para ela. Por exemplo, suponha que T recebe apenas dois inputs, suponhamos que neste caso, 0 e 10. O programa de T pode 1) aplicar o primeiro input, neste caso 0, na função sucessora, 2) utilizar o output de $suc(0)$ e fazer $+(suc(0), 0)$ e após 3) guardar a informação no espaço da terceira máquina $T_{A''}$. Lembremos que tudo isto é permito pelo teorema da composicionalidade (3). O resultado será o seguinte (a leitura é $suc, +, id$ respectivamente, de cima para baixo):

1
1
1

Mas a máquina T pode ser usada para realizar mais operações ainda. Pode por exemplo, 4) atualizar a função suc como um contador, i.e. dando o output da função suc novamente para a função suc; 5) fazer $+(n, m)$, com n sendo o output do item anterior e m o output da última fita; 6) armazenar o resultado do item anterior na última aplicando id ao item anterior. O resultado será o seguinte:

2
3
3

Uma coisa importante a se notar é que a partir daí podemos iterar este processo indefinidamente.

Notemos que para iterar qualquer máquina de Turing indefinidamente, apenas adicionamos estados posteriores a todas instruções de parada com instruções falando para voltarmos ao estado inicial e com o input atualizado. Por exemplo, se quiséssemos iterar indefinidamente o exemplo que demos da máquina sucessora (2.1) basta adicionar ao estado final $q_4 00 q_0, q_4 11 q_0, q_4 BB q_0$, assim, iremos compor indefinidamente a função sucessora:

$$suc \circ suc \circ suc ...(x)$$

Da mesma forma, neste exemplo que estamos mostrando, podemos repetir indefinidamente o passo 4) em diante, apenas adicionando instruções apropriadas. A segunda iteração a partir daí será a disposta na imagem a baixo.

Assim, vemos que esta máquina T repete a iteração somar números de 1 até infinito. A função por assim dizer entra em um loop. A primeira função, (a que está no primeiro item da tabela), suc, exerce a função de contador, a segunda

exerce o papel da função que queremos iterar, e a terceira é um armazenador de variáveis. Porém não parece interessante termos uma máquina que não pára. Se adicionarmos mais uma função a esta máquina, $K_{=}$, que verifica se dois números são iguais (uma função característica) e a utilizarmos para verificar se o contador em questão já chegou ao número 10, teremos uma máquina que pára. Ou seja temos a essência de um loop estruturado utilizado em linguagens de programação imperativas.

Acabamos de descrever como implementar um loop while em uma máquina de Turing específica. Um loop while é uma estrutura de iteração que repete algo enquanto alguma condição for verdadeira, no caso acima, enquanto o número i do contador (i.e. no output da função *suc*) for tal que $i \neq 10$. Ou seja, enquanto o número for diferente de 10, a máquina vai continuar a fazer instruções. A máquina de Turing que criamos é equivalente ao seguinte código (intuitivo):

```
i = 0
n = 10
x= 0
while (i ! = n):
     i = i +1
     x = x+ i
```

Estabelecemos determinadas variáveis (no caso das máquinas de Turing, reservamos determinadas partes correspondentes fita de forma que elas fiquem sempre separadas do resto da fita): $i = 0, n = 10, x = i$.

Utilizamos uma estrutura de repetição que repete as atribuições de variáveis enquanto $i \neq n$, isto é, enquanto o conteúdo da fita que está em i for diferente de 10. Quando entramos no loop, a máquina realizará as operações de atualização das variáveis, utilizando a composição da função *suc*, no caso de $i = i + 1$, que representa que o valor que estava em i vai ser atualizado pela função $i+1$; e pela função $+$, no caso $x = x + i$, que representa que o valor que estava em x será atualizado por $x + i$, isto é o valor que estava na parte da fita correspondente a i e na parte correspondente a x.

Podemos criar máquinas genéricas através deste procedimento, estabelecendo uma estrutura de repetição, atribuições de variáveis e algumas operações aritméticas. O procedimento vai ser exatamente o mesmo para outras máquinas de Turing quaisquer, outras funções quaisquer. A essência deste loop é que a máquina repete determinadas instruções até cumprir determinada condição (no nosso caso que $i \neq 10$).

Assim, a máquina de Turing administra os espaços de tal forma que pode atribuir valores a esses espaços: temos a noção de atribuição de variáveis (função id), temos um núcleo aritmético básico (função soma, e outras possíveis de serem

construídas a partir de máquinas de Turing), temos composição de funções, e temos um núcleo de repetição genérico a que chamamos de máquina while. Portanto temos uma máquina de Turing bastante sofisticada. Quanto sofisticada? Veremos a frente que o bastante para poder simular qualquer outra máquina de Turing. Ou seja, o suficiente para ser denominada um modelo computacional. Odifreddi [**p.68**; Odi99], se refere a ela como a base do paradigma computacional estruturado de linguagens de programação como *C*, *Python* etc... Aqui enriqueceremos esta máquina com mais algumas utilidades (a sintaxe é a igual a sintaxe de Python ou qualquer outra linguagem de programação imperativa e estruturada):

PROPOSIÇÃO **4.** ***Máquina while:*** *Existe uma máquina de Turing que dado um ou mais inputs faz as seguintes operações :*

1. *tem as funcionalidades de uma calculadora. Isto é, dados os inputs a, b, c, números naturais* [2] *(trabalharemos com inteiros não negativos) e predicados-enunciados p e q temos:*

   (a) *As seguintes funções aritméticas (comentamos após o # qual o significado de cada operação):*

   ```
   a + 1          # (sucessor)
   a -' 1         # (antecessor truncada ( i.e. 0-'1 = 0) )
   a + b          # (soma)
   a -' b         # (subtração truncada)
   a * b          # (multiplicação)
   a//b           # (função divisão inteira)
   pow(a,b)       # (a à potência de b)
   log'(a)         # (retorna o tamanho em bits de a (log))
   ```

   (b) *predicados-enunciados e operações booleanas:*

   ```
   a==b    #(a é igual a b)
   a!=b    #(a é diferente de b)
   a > b   #(a é maior que b)
   p and q #(operação booleana e )
   p or q  #(operação booleana ou)
   not p   #(operação booleana não)
   ```

2. *Tem a funcionalidade de criar espaços de memória. Ou seja pode criar enunciados de atribuição do tipo:*

   ```
   x = f(a_1, ... , a_n, x_1, ...,x_n)
   ```

   onde f é alguma função, $a_1, .., a_n$ *números,* $x_1, ..., x_n$ *variáveis (x está possivelmente entre elas).*

[2] Ver operações matemáticas p. 129. As funções == e ! = são definidas em valores booleanos 0 ou 1.

3. *Pode compor a execução de linhas do programa:*

```
P_1
P_2
...
P_n
```

, onde $P_1, ...P_n$ são construídos ou a partir de criação de espaços de memória, ou a partir dos procedimentos descritos nos itens a seguir.

4. *Pode fazer enunciados do tipo if-else, ou seja, se P é um enunciado verdadeiro, então P_1 a P_n serão executados em ordem, caso contrário serão executados $Q_1, ..., Q_n$:*

```
if P:
    P_1
    P_2
    ...
    P_n
else:
    Q_1
    ...
    Q_n
```

onde P_i e Q_i são enunciados (note a indentação). O programa faz os enunciados $P_1, ..., P_n$ se P é verdade, caso contrário faz $Q_1, ..., Q_n$.

5. *Pode fazer enunciados do tipo while:*

```
while P:
     P_1
     ...
     P_n
```

onde P_i são enunciados. O programa faz os enunciados $P_1, ...P_n$ enquanto P é verdade.

Se a máquina de Turing é uma composição de enunciados do tipo acima, então ela é um programa-while. E além disso, a máquina de Turing computa x sse existe um programa-while atribuindo a variáveis $a_1, ..., a_n$ aos resultados de `input()` e gerando o output `return(y)`, para alguma variável y ao fim da composição do programa.

3.1.3 Máquinas while-codificadoras

Uma observação importante é que dado uma máquina while, podemos esquecer os strings binários e pensar neles apenas como strings de um alfabeto mais sofisticado. Isto é o que é feito na prática de manipulação de strings nas linguagens de programação atual. Pode-se esconder a implementação detalhada da codificação ASCII, ou qualquer outra, que transforma strings em strings binários convenientes. Podemos rastrear historicamente este tipo de visão abstrata com respeito às possibilidades manipulatórias de strings à prova original de Incompletude dada por Gödel. De fato, as funções de manipulação de strings oferecidas por linguagem de programação atual como Python não são muito diferentes das que Gödel pensou. Vejamos algumas funções que nos auxiliarão (não demonstraremos)[3]:

PROPOSIÇÃO **5.** ***Máquina while-codificadora*** *Dado um número c, é possível construir máquinas while codificadoras lexicográficas de tamanho c que definem as seguintes funções de manipulação de string:*[4] .

```
str(x,c)          #(interpreta x como um string não binário
na ordem lexicográfica c)

bin(x,c)          #(interpreta x como string binário)

len(s)          # (retorna o comprimento do string s)

s[i]            # (retorna a i-ésima letra do string s)

s[i:j]          # (retorna um substring de s entre a i-ésima
letra e a j-ésima letra)

s+d             #(concatena o string s ao string d)

s.split(a)      # (divide o string s tendo como divisória o string a
e devolve uma sequência de substrings)
```

[3]Uma referência para nos convencermos que é possível definir estas funções, além da existência das linguagens de programação modernas, é: [Bek06].

[4]Para o leitor interessado, comparar com as seguintes funções dadas por Gödel em seu artigo histórico, *On Formally Undecidable propositions of Principia Mathematica and Related Systems I*, [Dav04] p.18-19. Citamos a seguir literalmente, incluindo a notação original:

- $l(x)$: é o comprimento da sequência de números correlacionado com x.
- $n\ Gl\ x$: é o n-ésimo termo da sequência de números correspondendo ao número x.
- $x * y$: corresponde a operação de justaposição de de duas sequências finitas de números x.
- $Su\ x\binom{n}{y}$: surge de x substituindo y pelo n-ésimo termo de x.

Por outro lado, as funções estabelecidas na proposição acima são literalmente a notação usada em Python. De certo modo, portanto, podemos dizer que Gödel foi o primeiro a criar um código de um programa no sentido contemporâneo da palavra.

```
s.replace(a,b) # (troca no string s, o string a por b)
```

3.2 Máquinas de Turing Universal

Uma máquina de Turing Universal é uma máquina de Turing que tem o poder de simular qualquer outra máquina de Turing (inserida devidamente codificada como input na máquina de Turing Universal). Mas o que é simular? Simular, em termo popular, é representar. Podemos fazer uma representação em uma tabela, mais precisamente uma tabela de configurações, repetindo passo a passo o resultado das instruções de um algoritmo. Ou seja, é exatamente o que fizemos no exemplo 2.1 quando escrevemos a tabela de configurações e instruções para a função sucessor. Simular portanto é uma forma de descrever todas as configurações e instruções de uma máquina de Turing para um dado input. Naquela ocasião, nós pegamos o input dado (11) e descrevemos todas as configurações geradas pela máquina de Turing específica, ordenando os passos. Ou seja, nós simulamos uma máquina de Turing específica. Portanto, nós fizemos o papel de máquina de Turing Univeral.

Mas esta simulação pode ser mecanizada. No exemplo que fizemos, havia uma repetição de determinados procedimentos, resultantes da própria definição de máquina de Turing. Ao fazer no papel qualquer simulação de máquina de Turing, já se pode ver como seria uma possível mecanização. E é neste contexto geral (que pode ser simulado com lápis e papel) que iremos ver como mecanizar o processo.

3.2.1 Exemplo de Simulação

Suponha que temos a tabela 2.1 preenchida até a configuração do passo 2 o restante em branco. Daremos um exemplo de como o algoritmo funcionaria a partir daí, preenchendo automaticamente os espaços em branco da tabela. Reproduziremos uma tabela parcial para cada instrução desse algoritmo:

nº de passos	instrução	configuração (o string significativo)
2		$\mathbf{11q_0}$
$\phi = \{q_0 0 R q_0,\ q_0 1 R q_0,\ q_0 B L q_1 ...\}$		
memória: $(\mathbf{q_0}, \mathbf{B})$		

Tabela 3.1: item 1

1. olhe para a configuração do passo em questão (linha 2) e veja qual símbolo está sendo lido pela cabeça da máquina (i.e. em $11q_0$, é o B) e olhe qual é o atual estado da configuração (q_0). Guarde na memória estas duas informações (q_0, B);

2. olhe para o conjunto das instruções ϕ e busque qual instrução começa com o estado que lemos na configuração guardada na memória do item anterior

nº de passos	instrução	configuração (o string significativo)
2	$\mathbf{q_0BLq_1}$	$11q_0$
$\phi = \{..., (\mathbf{q_0BLq_1})...)\}$		
memória: $(\mathbf{q_1}, \mathbf{L})$		

Tabela 3.2: item 2

(q_0) e depois o símbolo que lemos do item anterior (B) (que no caso é o estado q_0BLq_1), copie e cole a instrução na coluna instrução ao lado da configuração do passo em questão (linha 1), leia o restante da instrução e guarde na memória (q_1, L);

nº de passos	instrução	configuração (o string significativo)
2	q_0BLq_1	$11q_0$
3		$\mathbf{11q_0}$
$\phi = \{..., (q_0BLq_1)...)\}$		
memória: (q_1, L)		

Tabela 3.3: item 3

3. copie e cole a configuração do passo atual (linha 2) para o próximo passo (linha 3) e pule para prosseguir a leitura na próxima linha.

nº de passos	instrução	configuração (o string significativo)
2	q_0BLq_1	$11q_0$
3		$\mathbf{1q_11}$
$\phi = \{..., (q_0BLq_1)...)\}$		
memória: (q_1, L)		

Tabela 3.4: item 4

4. Nesta nova linha, executaremos a mudança de passo conforme a definição 7. Em primeiro lugar atualizamos o estado (de q_0 para q_1) e em suma, teremos dois casos, a depender do outro conteúdo da memória:

   (a) se for um elemento de $\{R, L\}$, então mova o símbolo da cabeça da máquina para a direita ou para esquerda, colocando B convenientemente (esse é o nosso caso, pois temos na memória L. Temos que mover o sublinhado para esquerda);

   (b) se o símbolo for um elemento de nosso alfabeto, então troque o que está na leitura da cabeça da máquina pelo que está na memória.

5. Se houver mais quádruplas para ler, então repita os itens de 1-4. Se não, pare o algoritmo e dê como resultado o output da última linha.

Vemos aqui qual é o maior problema da simulação: codificar o conjunto de instruções. Pois a quantidade de símbolos que representam estados (q_i) não é fixo e, portanto, temos como tarefa codificar todas estes símbolos possíveis para depois aplicar o procedimento que demos no exemplo anterior. Daí porque tivemos que falar anteriormente sobre codificações. Tendo uma máquina while-codificadora que faz todo o trabalho de codificação para nós, não é muito difícil ver que este procedimento pode ser de fato programável. Esta simulação nos dá uma máquina universal, que simula qualquer outra máquina de Turing, desde que alimentada com o input codificado correto.

3.2.2 Máquinas Universais

TEOREMA **5.** *(Turing 1936)* ***Teorema da universalidade:***
Seja $T = \{P, Q\}$ uma máquina de Turing, então existe uma máquina de Turing U, que denominaremos máquina Universal, e que é tal que:

$$U(m, x) = T(x),$$

onde x é input tanto de T quanto de U, e m é outro input para a máquina U e corresponde ao código do programa de T.

Demonstração. Em primeiro lugar, precisamos dar mais precisão sobre como codificaremos o conjunto de instruções e configurações. Considere g a função codificadora lexicográfica 17 definida no seguinte alfabeto:

$$g : Q \cup \{B, R, L\} \to \{0, 1\}^*$$

Ou seja, realizaremos uma codificação de comprimento fixo $c = \log(|Q| + 3)$. Podemos passar a informação de c, a informação de quantas instruções há no programa n junto às outras informações concernentes ao programa da seguinte maneira:

$$m = 1^n 01^c 0 g(p),$$

onde p é a concatenação das instruções de P. I.e. p é um string do tipo:

$$q_0 b_0 b_1 q_k ... q_r b_{i-1} b_i q_n.$$

Logo existe uma máquina de Turing U com o seguinte programa que descreveremos resumidamente. Para detalhes sobre a simulação, um programa mais detalhado feito para a linguagem Python é dado no apêndice. (comentários sobre cada passo estão após #)

```
m = input()

x= input()
```

```
# Recebemos estes inputs codificados da maneira acima,
# temos que retirar os números n e c para sabermos como descodificá-los:

def UN: Função que conta uns até achar um zero.

def Limpa: Função que retira o primeiro bloco do seguinte tipo: 1...10.

n = UN(m)

m = Limpa(m)

c = UN(m)

m = Limpa(m)

# Em mãos da informação de qual é
# o tamanho da codificação de tamanho fixo (c),
# descodificamos m

programa = str(m,c)

while(Existir outros estados para ler)
    Faça a simulação dada no exemplo da sessão anterior
    (ou o programa do apêndice).
    Armazenamos as informações do string significativo na variável string.

s = bin(string, c)

return(s)
```

□

COROLÁRIO **1.** *Para toda função p.c. ϕ existe uma função universal ψ tal que:*

$$\psi(m, x) = \phi(x)$$

A existência da máquina de Turing universal mostra, em seu princípio, a ideia de que o hardware não precisar crescer indefinidamente para absorver a complexidade da classe de todas as máquinas de Turing: esta complexidade pode ser transferida para o input. Não é necessário construir maquinários específicos para lidar com um problema específico. É possível construir uma máquina programável que, caso não tenhamos a funcionalidade desejada, basta programarmos a funcionalidade desejada (i.e. inserir um input). Em sentido geral, aqui está presente a ideia de que programas são de fato números, como vimos isto na introdução e veremos mais detidamente no próximo capítulo.

Capítulo 4

Máquinas como números e números como máquinas

> No capítulo anterior, fizemos um caminho de ideias que atualizou nossa concepção do que significa computar. Vimos também, embora não explicitamente, que é possível construir (provar a existênciade) máquinas de Turing cada vez mais complexas: máquina de Turing → máquina while → máquina codificadora → máquina universal.
>
> Veremos na primeira seção deste capítulo o teorema da enumerabilidade e parametrização, resultados mais fortes do que o teorema da universalidade;[1] também veremos nessa seção o mesmo método da diagonalização originado por Cantor, porém sendo utilizado para demonstrar que é impossível criar uma máquina de Turing que consegue determinar se outra máquina de Turing pára ou não. Neste sentido, esta primeira seção é uma espécie de embate entre os conceitos de enumerabilidade e diagonalização. Na segunda seção, trataremos sobre o teorema do ponto-fixo. E depois veremos como a classe de funções computáveis se localiza no miolo do confronto entre o conceito de enumerabilidade e diagonalização, e como isso tem como consequência a auto-referência.

Podemos pensar linguagens de programação como C, Python, Fortran, etc... como definições de determinados strings que, dado um computador físico, funcionam como funções computáveis universais. Obviamente essa afirmação precisaria de prova, porém estamos trabalhando em termos intuitivos, e nossa intuição nos diz que tudo que é feito por estas linguagens pode ser traduzido (compilado é um dos termos usados) tanto 1) para o nível da máquina física, que por sua vez pode ser visto como um modelo finito e limitado de uma máquina de Turing; quanto 2) traduzido de uma linguagem de programação para outra. Isto também

[1] No sentido de que implicam o teorema da universalidade.

pode ser entendido ao realocarmos a argumentação que demos na demonstração do teorema 5 da universalidade nesse contexto de maquinário finito.

Desse modo, podemos afirmar que, se definirmos corretamente o input e output de strings, podemos criar máquinas de Turing que funcionam exatamente como C, Python, Fortran. Ou seja podemos criar uma espécie de tradutor destas linguagens para o contexto de máquinas de Turing. E assim é possível mostrar que cada um desses modelos de computação equivalem a um só, à classe de funções computáveis. Logo, é possível falar de funções p.c. $\phi_C, \phi_{Python}, \phi_{Fortran}$.

Além desse interesse prático atual, é preciso salientar que a classe de funções computáveis por máquina de Turing não é equivalente somente a nossa prática contemporânea computacional, mas também é equivalente a diversos outros modelos computacionais que foram desenvolvidos ao longo da história. Duas perguntas surgem naturalmente desta reflexão e dA aparente ubiquidade das funções computáveis parciais ou totais:

> A classe de funções parciais computáveis constitui a classe de todas as funções que existem?

A resposta é não. E um dos objetivos desta seção é explorar exatamente a razão disto, cuja resposta está relacionada, entre outras coisas, com métodos de diagonalização originados com Cantor.

4.1 Diagonalização e Enumeração de Máquinas de Turing

4.1.1 Enumeração e Parâmetro

Como vimos na introdução, o teorema de Cantor 1 nos diz que o conjunto de todos subconjuntos de um conjunto enumerável infinito é não-enumerável. A ideia pode ser transposta para o contexto de alfabetos e strings. Neste contexto, podemos enunciá-lo da seguinte forma: o conjunto de todos strings infinitos sobre um alfabeto de cardinalidade maior ou igual a 2 não pode ser enumerável.

Assim, temos uma resposta negativa para o problema da enumeralidade destes conjuntos infinitos. Porém, se mudarmos o enunciado e nos perguntarmos se o conjunto de todos subconjuntos *finitos* de um conjunto enumerável infinito é enumerável, a resposta será sim.

Tomemos o conjunto enumerável $\mathbb{N}$, e tomemos todos os subconjuntos finitos com dois elementos. Para cada elemento deste subconjunto finito com dois elementos podemos associar uma sequência $\{a_1, a_2\} \mapsto \langle a_1, a_2 \rangle$. E utilizando o resultado 18 de que é possível mapear estas sequências bijetivamente nos naturais, temos: $\{a_1, a_2\} \mapsto n$, para algum $n \in \mathbb{N}$. Podemos argumentar de maneira mais geral pra obter:

Proposição **6.** *O conjunto de todos subconjuntos finitos de um conjunto enumerável infinito é enumerável.*

Demonstração. Sabemos discussão e definição de codificação de pareamento standard 18 que toda sequência de tamanho n pode ser bijetivamente mapeada nos naturais. Logo, o conjunto das sequências finitas sobre um conjunto enumerável é enumerável. Como também podemos mapear injetivamente todo subconjunto finito de tamanho n nas sequências de tamanho n, temos o resultado. □

TEOREMA **6.** ***Teorema da enumeração:*** *Seja $T_n = \{P, Q\}$, então existe uma máquina de Turing Universal T tal que:*

$$T(n, x) = T_n(x)$$

Demonstração. Seja i um contador que T aloca em algum espaço de memória. A máquina T vai gerando todos os códigos m de programas possíveis da forma como descrevemos no teorema da universalidade. Para cada código m, T verifica se este código é de fato um programa válido observando se ele respeita as cláusulas da definição de 5 de máquina de Turing . Para cada programa que encontrar, T soma 1 ao contador i. Ao encontrar o n-ésimo programa, T simula o programa tal como mostramos no teorema da universalidade.

Observações: Como o código é lexicográfico, não há o perigo de pularmos algum programa válido. Além disso, sabemos que encontraremos todas as máquinas, pois o conjunto de todos os programas tem o mesmo tamanho do conjunto de subconjuntos finitos de um conjunto enumerável, logo, pelo teorema anterior o conjunto dos programas é enumerável. □

COROLÁRIO **2.** *É possível estabelecer uma enumeração bijetiva parcial computável:*

$$\pi : \mathcal{FPC} \to \mathbb{N},$$

onde $\mathcal{FPC}$ é o conjunto das funções parciais computáveis. Ou seja:

- *Para todo e, ϕ_e é uma função p.c.*
- *Se ψ é p.c. existe um número e tal que: $\phi_e = \psi$.*
- *Existe uma função parcial computável correspondente a uma máquina universal tal que*

$$\phi(n, x) = \phi_n(x)$$

LEMA **1.** ***(Lema do Empilhamento)****É possível gerar de forma computável infinitos índices para uma mesma função p.c.*

Demonstração. Seja ϕ_n uma função parcial computável. Introduza novas instruções redundantes na M.T. associada, por exemplo $q_k a R q_k$, $q_{k+1} a L q_{k+1}$, onde a é um símbolo qualquer. Geraremos uma outra MT que computa a mesma função com índice diferente de n.

□

TEOREMA **7.** ***(Teorema S-m-n ou Teorema do Parâmetro)*** *(Kleene) Dados m e n, existe uma função computável $s_n^m(e, x_1, ..., x_n)$ tal que:*

$$\phi_{s_n^m(e,x_1,...,x_n)}(y_1, ..., y_m) = \phi_e(x_1, ..., x_n, y_1, ..., y_m,)$$

Demonstração. Façamos com que a função s corresponda a uma máquina que decodifica o índice e, emula a máquina correspondente a ϕ_e, adiciona os parâmetros desejados para cada x_i na configuração inicial e devolve o número codificado das novas instruções. Note que este procedimento tem que ser feito toda vez que os parâmetros mudarem. Isto não é um problema, pois o procedimento que faremos é independente de quais forem os inputs.

□

Como vimos na introdução (sessão ambivalência computacional), o teorema do parâmetro faz a operação inversa do teorema da enumeração. Estes dois teoremas completam o círculo fundamental da teoria da computabilidade e mostram que do ponto de vista da computabilidade funções parciais computáveis e números estão no mesmo nível. Com o mesmo nível ontológico, queremos dizer que do ponto de vista da teoria que estamos desenvolvendo, tanto faz dizermos que computadores são números naturais em uma determinada teoria aritmética ou são funções parciais computáveis. Ambas coisas se comportam da mesma forma, os resultados são espelhados, tanto a teoria da computabilidade e da aritmética se entrecruzam.

Com o teorema do parâmetro podemos interpretar dados do input como incorporados ou *hardcoded* no programa. Em última instância toda função computável pode ser pensada como uma função constante, um número, que ao ser passado a uma máquina, retorna outro número, ou, o que é o mesmo, outra máquina. Como [Odi99] diz: o teorema "incorpora a noção de subcomputação e uma versão efetiva de composição".

Aqui fica mais explicito do que antes, quando expomos o teorema da universalidade, a ideia de que programas são números e números são programas.[2] Desta perspectiva, a atividade do programador quando está escrevendo se aproxima da atividade de alguém que busca incorporar determinados inputs de seu teclado na memória do computador, e através de uma sucessão de traduções para a linguagem mais próxima da máquina, assim construir um subprograma, ou seja, algo que a máquina pode acessar quando desejar. Mas, ao mesmo tempo, este subprograma pode ser visto como um determinado input que faz com que o computador acesse os estados e configurações corretos para uma determinada tarefa. Números ganham dois sentidos diferentes, um enquanto entidade construída a partir da ideia de contagem e indução, outro como programa.

4.1.2 Diagonalização

Agora podemos enumerar e, consequentemente, podemos checar a existência de uma máquina de Turing a partir de um método. Porém, a pergunta que vem

[2] Ver [Odi99], p.132

a seguir é se esta enumeração de funções p.c. por máquina de Turing pode ser convertida em uma enumeração de máquinas de Turing de um tipo específico: as máquinas de Turing que param. Isto não é o caso.

O argumento de porque isto não é o caso vem diretamente do raciocínio empregado por Cantor em seu teorema, e seu núcleo conceitual está na possibilidade de achar uma função diagonal. Vamos explicitar aquilo que [Odi99] p.145 expõe como a diagonalização em sua essência. Exporemos como uma proposição cuja prova é imediata, nosso propósito porém é expor nomenclaturas e a generalização da ideia.

PROPOSIÇÃO **7.** *1. Seja S qualquer conjunto (com pelo menos dois elementos) $(a_{\omega,\omega})$ uma matriz infinita com elementos $a_{ij} \in S$;*

2. Seja a linha $l_k = \{a_{k,i}\}_{i\in\omega}$, para algum $k \in \mathbb{N}$;

3. Se $d : S \to S$ é uma função tal que $d(a) \neq a$ Isto é, d é uma função diferente da identidade.

Então, a diagonal transformada pela função d nunca será igual a alguma linha. Isto é, $\{d(a_{ii})\}_{i\in\omega} \neq l_k$ para todo $k \in \omega$.

$$\begin{bmatrix}
\mathbf{b_1} & a_{12} & \dots & & & a_{1n} & \dots \\
a_{21} & \mathbf{b_2} & \dots & & & a_{2n} & \dots \\
\vdots & \dots & \ddots & & & & \dots \\
a_{i1} & a_{i2} & \dots & \mathbf{b_i} & \dots & a_{in} & \dots \\
\vdots & \dots & & & \ddots & & \dots \\
a_{n1} & a_{n2} & & & & \mathbf{b_n} & \dots \\
\vdots & \dots & & & & & \ddots
\end{bmatrix}$$

De fato, basta perceber que há pelo menos um elemento da diagonal transformada diferente de cada linha, isto é $d(a_{ii}) = b_i \neq a_{ii}$.

□

Temos também que esclarecer um aspecto, há mais funções *não computáveis* do que funções parciais computáveis por máquinas de Turing. Isto segue do teorema de Cantor. Como vimos na introdução o conjunto das partes de $\mathbb{N}$ está em bijeção com o conjunto $2^{\mathbb{N}}$ (das funções $\mathbb{N} \to 2$). Mas $f \in 2^{\mathbb{N}}$ pode ser pensada como a função característica de um subconjunto de $\mathbb{N}$, onde os elementos que são levados em 1 são aqueles que estão neste subconjunto e os que são levados em 0 são os que não estão. Podemos pensar neste conjunto $2^{\mathbb{N}}$ como subconjunto de $\mathbb{N}^{\mathbb{N}}$, e com mais razão ainda este conjunto de funções será não enumerável. Mas o número de funções computáveis é enumerável pelo teorema da enumerabilidade. Logo, o conjunto das funções em geral (não necessariamente computáveis) é maior do que o conjunto das funções computáveis.

PROPOSIÇÃO **8.** *Há mais funções do que funções parciais computáveis.*

Um exemplo específico de função que não pode ser computada é uma função que computa se uma outra função computável ϕ_n qualquer pára ou não, o que ficou conhecido na literatura como o problema da parada (ou teorema da parada). A ideia deste teorema é mostrar que é impossível criar um programa que decide se outro programa pára ou não. A intuição da prática da programação pode ser pensada da seguinte maneira: alguém dá um programa para que seja testado. Nós adicionamos um input a este programa, ele pode parar ou não. Podemos criar um programa que testa se este programa pára ou não, basta criar um programa que espera o programa dado parar. Mas se este programa não parar, não é possível dizer nada sobre a parada do programa.

TEOREMA **8.** *(Turing, Problema da parada) Não há uma função total computável d tal que*

$$d(n,x) = \begin{cases} 1 & \textit{se } \phi_n(x) \downarrow \\ 0 & \textit{se } \phi_n(x) \uparrow \end{cases}$$

Demonstração. Provaremos por absurdo usando a diagonalização. Suponha que existe uma função nas premissas do enunciado. Defina a função *parcial* computável ϕ_i:

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

Definimos a função ϕ_i a partir da função total computável d e podemos afirmar que ela é a i-ésima máquina pelo teorema da enumeração. Sobre o detalhe da como criar a indefinibilidade em ϕ_i: podemos simplesmente criar um subprograma que faz a máquina de Turing correspondente a ϕ_i ir infinitamente para direita quando receber a informação que $d(x,x)$ é um.

Faça $x = i$. Então:

$$\phi_i(i) \downarrow \Leftrightarrow \phi_i(i) = 1 \Leftrightarrow d(i,i) = 0 \Leftrightarrow \phi_i(i) \uparrow$$

A primeira equivalência à esquerda se segue da definição que demos de ϕ_i, as outras decorrem da definição de d do enunciado.

Se segue dessa contradição que não há uma função p.c. que consegue computar,para toda outra função p.c., se outra função parcial está definida ou não. □

Notemos que nesta prova fornecemos uma matriz aos moldes do teorema de Cantor: $a_{ij} = \phi_i(j)$, e a função diagonal é a d.

Uma conclusão talvez um pouco estranha destes resultados, mas que reflete a peculiaridade das funções *parciais* computáveis em relação às *totais* computáveis, é a seguinte:

1. As funções *totais* computáveis são enumeráveis. Pois, estas são um subconjunto das parciais computáveis, e estas como vimos são enumeráveis pelo teorema da enumerabilidade.

2. Pelo problema da parada, as funções *totais* computáveis não podem ser enumeradas por uma máquina de Turing. Pois não podemos ter uma máquina que sabe se outra máquina pára ou não.

3. O teorema da enumerabilidade não vale se restringimos nossa noção de função a apenas as que param (i.e. funções totais computáveis). Ou dito de outro modo, embora haja uma enumeração das funções computáveis, essa enumeração não pode ser computada (i.e. não é uma enumeração efetiva).

Esta consideração nos mostra mais uma vez a importância da distinção entre funções parciais e totais, pois há resultados dentro da teoria da computabilidade que não valem para funções totais, mas que valem quando retiramos a condição de que estas funções sejam totais e possam ser também parciais. Entre elas, como dissemos, o teorema da enumerabilidade. Odifreddi na página 146 [Odi99] se refere a ele como versão do teorema de Cantor para funções recursivas.

Proposição **9.** *(Kleene, Turing) Não há uma função total computável f que enumera todas as funções totais computáveis g. Isto é não é possível afirmar que:*

$$f(n, x) = g_n(x),$$

para f e g computáveis.

As funções parciais computáveis não estão sujeitas a este mesmo argumento porque elas, como Odifreddi [Odi99] diz na página 152, "parecem ter defesas internas contra a diagonalização". Ou seja, enumerar algo através de uma função *total* computável é diferente de enumerar algo através de uma função p.c. Isto nos leva à definição:

Definição **19. Conjunto computacionalmente enumerável (c.e.) (ou recursivamente enumerável r.e.); conjunto computável (ou recursivo).** Um conjunto $\mathcal{W}_e$ é *computacionalmente enumerável* (c.e.) se é o domínio de uma função parcial computável ϕ_e. Um conjunto $\mathcal{W}_e$ é *computável* se possui uma função característica computável.

Segue-se imediatamente do problema da parada que:

Corolário **3.** *Existe um conjunto c.e. não computável.*

Este corolário é tido como a versão computacional do teorema da incompletude. Porém, não fazemos referência a nenhum teoria, no teorema da incompletude por outro lado, temos que fazer referência a algum tipo de teoria.

Definição **20. Conjunto parada:** Defina $K_0 = \{\langle x, y\rangle : \phi_x(y) \downarrow\}$ como o conjunto parada.

Provemos mais algumas propriedades sobre conjuntos computacionalmente enumeráveis e computáveis:

PROPOSIÇÃO **10.** ***Propriedades dos conjuntos c.e.:***

1. *Um conjunto é c.e. sse é imagem de uma função parcial computável.*
2. *Um conjunto é c.e. sse é o conjunto vazio ou imagem de uma função computável.*
3. ***(Teorema de Post)*** *Um conjunto é computável sse tanto seu complemento quanto ele mesmo são c.e.*
4. *Um conjunto infinito é computável sse é computacionalmente enumerável em ordem crescente.*
5. *Todo conjunto infinito c.e. possui um subconjunto infinito computável.*

Demonstração. Em ordem:

1. $A = \mathcal{W}_e$, logo para algum e, podemos definir:

   $$\phi(x) = x \Leftrightarrow \phi_e(x) \downarrow$$

   E o domínio de ϕ_e é igual à imagem de ϕ. Logo A é a imagem de ϕ. A volta pode ser feita igualmente.

2. *(intuitivo)* Pelo anterior, A é imagem de uma função p.c. e ϕ_e e podemos definir uma $f(x)$ computando aos poucos todos os possíveis valores de ϕ_e e colocando em f à medida que encontramos a imagem de A. (Cuidado, podemos cair no problema da parada se não definirmos precisamente o que significa 'computar aos poucos'. Para uma demonstração mais cuidadosa ver [Odi99] p.138. A ideia da prova é fazer uma operação de *dovetailing*, ou seja de entrelaçar as computações estabelecendo um limite para que o programa não fique rodando infinitamente em busca de um output). A volta se dá porque se A for o conjunto vazio é o domínio da função $\phi \uparrow$ para qualquer input.

3. *(Ida)* Se A é computável, então seu complemento também será computável. Como toda função computável é parcial computável e temos que A e A^c é domínio da função χ_A, temos por definição que ambos são c.e.

   (Volta, intuitivo) Suponha que A e A^c são c.e. Pelo item 2, façam A a imagem de f e A^c a imagem de uma g. Podemos gerar a lista:

   $$\{f(0), g(0), f(1), g(1), ...\}$$

   Examinando a lista, podemos decidir se $x \in A$. Basta verificar se para algum $i \in \mathbb{N}$, $f(i) = x$ ou $g(i) = x$ (caso no qual $x \in A^c$).

   Notemos que assumimos que x vai aparecer na lista porque:

   $$\{f(0), g(0), f(1), g(1), ...\} = A \cup A^c = \mathbb{N},$$

caso contrário A e A^c são finitos e podemos vasculhar na força bruta se x está em A ou A^c. Além disso assumimos que nem A nem A^c são triviais (o conjunto vazio ou os naturais) caso contrário eles já seriam computáveis.

4. A ida é imediata (toda computável é c.e.). Quanto a volta, se A é enumerável em ordem crescente: $a_1, a_2, ...$, então podemos achar x testando se $x = a_i$ ou se $x > a_i$.

5. Pelo item 2 A é imagem de uma f computável. Logo, podemos usar f para definir uma função g utilizando um contador i tal que $g(0) = f(0)$ e $g(i+1) = f(x)$ sempre que $f(x)$ for maior que $g(i)$. A imagem de g é a um conjunto c.e. $B \subseteq A$ gerado em ordem crescente e portanto é computável pelo item anterior.

□

4.2 Teorema do Ponto-Fixo

4.2.1 Discussão preliminar

Vejamos algumas aplicações já relacionadas com o teorema do ponto fixo seguindo Rebecca Weber [Web12] p.84. Ela explica que o teorema do ponto fixo é o que pode ser salvo de um ponto-fixo no nível das funções. Expliquemos. Se alguém não nos dissesse que o ponto fixo das funções recursivas ocorre no nível dos *índices* e não no nível das próprias funções, poderíamos pensar que se trata de um ponto fixo para a função computável f. Para toda f é possível encontrar um n tal que:

$$f(n) = n.$$

Obviamente esta afirmação é falsa em geral, é só pensar em uma função que não intercepta o eixo x. Porém, persistamos um pouco mais e vejamos o que há de errado na seguinte argumentação. Seja $\delta(x) = \phi_x(x)$, uma função total que computa sempre a x-ésima função parcial em x. Pelo teorema da enumeração, para algum e, $f \circ \delta = \phi_e$. Mas então:

$$\delta(e) = \phi_e(e) = f(\delta(e).$$

Logo $\delta(e)$ é o ponto fixo de f? Qual foi o erro? Em assumir que $f(\delta(e)) \downarrow$. E ela terá que divergir nesse ponto se quisermos garantir a falsidade de que toda função possui um ponto fixo. Logo, esta reflexão nos mostra que o ponto fixo em questão deve estar no nível dos índices:

$$\phi_{f(n)} = \phi_n$$

Uma outra observação inicial antes de provarmos o teorema. Imaginemos que temos uma função parcial $\phi_u(u)$. Como esta função é parcial, então ela pode carregar consigo uma indefinibilidade essencial. Se essa função funciona

como índice para outra função, então podemos interpretá-la como uma função total e deixar a carga do indefinido para a função indexada (a função indexada é a 'de cima', o índice é o 'de baixo'). Vejamos como fazer isso:

$$\phi_e(u,x) = \begin{cases} \phi_{\phi_u(u)}(x) & \text{, se } \phi_u(u)\downarrow \\ \uparrow & \text{, se } \phi_u(u)\uparrow \end{cases}$$

Poderíamos escrever sinteticamente a equação a cima como: $\phi_e(u,x) = \phi_{\phi_u(u)}(x)$, já dando a entender que o lado esquerdo está definido somente quando o índice do lado direito está definido.

Pela parametrização podemos achar

$$\phi_{s(e,u)}(x) = \phi_e(u,x).$$

Podemos pensar $s(e,u)$ como uma função de uma variável, pois podemos fixar cada e. Logo, podemos fazer: $d(u) = s(e,u)$. E escrever resumidamente:

$$\phi_{d(u)}(x) = \phi_{\phi_u(u)}(x)$$

Logo, conseguimos efetivamente uma função total d que faz o mesmo trabalho de ϕ_u, pois sempre que $\phi_u \uparrow$, a função $\phi_{\phi_u} \uparrow$, e também $\phi_{d(u)} \uparrow$. Por outro lado se ϕ_u converge, como d já é total, isto não causa nenhum problema. Portanto, criamos um truque para pensarmos programas-índices como funções totais.

Utilizaremos isso para o teorema do ponto fixo. Uma última, observação. Segundo [Odi99], p. 153.

$$\phi_{\phi_a(u)}$$

pode ser imaginado como uma transformação do programa de número u pela função parcial ϕ_a. Já se fizermos $a = u$:

$$\phi_{\phi_u(u)}$$

pode ser pensada como uma transformação do programa u por si mesmo. E se considerarmos:

$$\phi_{f(\phi_u(u))}$$

pode ser pensada como a transformação do programa de número $\phi_u(u)$ de acordo com a função f. E logo, poderia ser pensada como o programa que transforma a si mesmo de acordo com a função f. Ora um programa que faz isso, se tem o índice n só pode também ter como número $f(n)$. Isto é a situação imagética que provaremos formalmente a seguir com duas provas diferentes, a primeira é a versão imediata, e a segunda a quase-imediata.[3]

[3] Apresentaremos duas provas porque este teorema é de importância fundamental e ao mesmo tempo difícil de ser apreendida intuitivamente.

4.2.2 Teorema do ponto fixo

TEOREMA **9.** ***(Teorema do Ponto-Fixo) (Kleene)*** *Para toda função computável (total) f existe um ponto fixo n tal que:*

$$\phi_n = \phi_{f(n)}$$

Primeira prova: Pelo teorema do parâmetro (e pelo truque que exploramos acima para a função $f \circ \phi_x$) temos:

$$\phi_{f\circ\phi_x(x)} = \phi_{d(x)}$$

Mas d tem índice v em alguma enumeração: $d(x) = \phi_v(x)$. Logo:

$$\phi_{f\circ\phi_x(x)} = \phi_{\phi_v(x)}$$

Fazendo $x = v$:

$$\phi_{f(\phi_v(v))} = \phi_{\phi_v(v)}$$

Logo $n = \phi_v(v)$ é nosso ponto fixo.

□

Segunda prova: Pelo teorema do parâmetro:

$$\phi_{d(u)}(x) = \begin{cases} \phi_{\phi_u(u)}(x) & \text{, se } \phi_u(u)\downarrow \\ \uparrow & \text{, se } \phi_u(u)\uparrow . \end{cases} \tag{4.1}$$

Pela enumeração, seja ϕ_v tal que

$$\phi_v = f \circ d. \tag{4.2}$$

Afirmamos que $n = d(v)$ é o ponto-fixo da função f. De fato, como ϕ_v é total, converge. Então, pela equação 4.1, $\phi_{d(v)} = \phi_{\phi_v}$. Logo:

$$\begin{aligned} \phi_n &= \phi_{d(v)} && \text{(def. n)} \\ &= \phi_{\phi_v(v)} && \text{(eq.4.1)} \\ &= \phi_{f\circ d(v)} && \text{(eq.4.2)} \\ &= \phi_{f(n)} && \text{(def. n)} \end{aligned}$$

□

Como salientaram Soare p.29, [Soa16], e Odifreddi p.154, [Odi99], este teorema pode ser visto como uma "diagonalização que falha". Vamos nos referir à nossa definição de diagonalização da seção anterior e definir uma diagonalização que falha como a contrapositiva do processo de diagonalização. Ou seja:

(Diagonalização que falha.) Seja $(a_{\omega,\omega})$ uma matriz infinita com elementos de um conjunto S e seja d uma função qualquer, se a diagonal $D =$

$\{d(a_{ii})\}_{i\in\omega}$ é igual a alguma a linha $l_i = \{a_{i,i}\}_{i\in\omega}$ da matriz, então existe pelo menos um elemento a_{ii} da matriz tal que $d(a_{ii}) = a_{ii}$.

$$\begin{bmatrix}
\mathbf{d(a_{i1})} & a_{12} & ... & & & a_{1n} & ... \\
a_{21} & \mathbf{d(a_{i2})} & ... & & & a_{2n} & ... \\
\vdots & ... & \ddots & & & & ... \\
\mathbf{a_{i1}} & \mathbf{a_{i2}} & ... & \boldsymbol{a_{ii}} & ... & \mathbf{a_{in}} & ... \\
\vdots & ... & & & \ddots & & ... \\
a_{n1} & a_{n2} & & & & \mathbf{d(a_{in})} & ... \\
\vdots & ... & & & & & \ddots
\end{bmatrix}$$

Ou seja, a_{ii} é o ponto fixo.[4]

Acontecerá a mesma coisa com as funções p.c., embora não do mesmo jeito, pois faremos duas diagonalizações. Seguiremos a segunda demonstração que demos acima (a primeira não deixa claro essa questão):

Podemos fazer uma matriz infinita $a_{i,j} = \phi_{\phi_i(j)}$:

$$\begin{bmatrix}
\phi_{\phi_1(1)} & \phi_{\phi_1(2)} & \cdots & & & \phi_{\phi_1(v)} & \cdots \\
\phi_{\phi_2(1)} & \phi_{\phi_2(2)} & \cdots & & & \phi_{\phi_2(v)} & \cdots \\
\vdots & ... & \ddots & & & & ... \\
\phi_{\phi_e(1)} & \phi_{\phi_e(2)} & \cdots & \phi_{\phi_e(e)} & \cdots & \phi_{\phi_e(v)} & \cdots \\
\vdots & ... & & & \ddots & & ... \\
\phi_{\phi_v(1)} & \phi_{\phi_v(2)} & & & & \phi_{\phi_v(v)} & \cdots \\
\vdots & ... & & & & & \ddots
\end{bmatrix}$$

Pelo teorema do parâmetro, a diagonal será igual a alguma das linhas. Seja esta linha a correspondente a e. Logo:

$$\begin{bmatrix}
\phi_{\phi_1(1)} & \phi_{\phi_1(2)} & \cdots & & & \phi_{\phi_1(v)} & \cdots \\
\phi_{\phi_2(1)} & \phi_{\phi_2(2)} & \cdots & & & \phi_{\phi_2(v)} & \cdots \\
\vdots & ... & \ddots & & & & ... \\
\phi_{d(1)} & \phi_{d(2)} & \cdots & \phi_{d(e)} & \cdots & \phi_{d(v)} & \cdots \\
\vdots & ... & & & \ddots & & ... \\
\phi_{\phi_v(1)} & \phi_{\phi_v(2)} & & & & \phi_{\phi_v(v)} & \cdots \\
\vdots & ... & & & & & \ddots
\end{bmatrix}$$

Obtemos com esta equiparação linha-diagonal, a equação 4.1: $\phi_{d(u)} = \phi_{\phi_u(u)}(x)$, para cada u da diagonal (primeira diagonalização).

Façamos uma translação da linha-diagonal pela função f, ou seja façamos a equação 4.2 $\phi_v = f \circ d$ (segunda diagonalização):

[4] Notemos que este é exatamente o conteúdo do teorema do ponto fixo tal como exibida na introdução.

$$\begin{bmatrix}
\phi_{\phi_1(1)} & \phi_{\phi_1(2)} & \cdots & & & \phi_{\phi_1(v)} & \cdots \\
\phi_{\phi_2(1)} & \phi_{\phi_2(2)} & \cdots & & & \phi_{\phi_2(v)} & \cdots \\
\vdots & \ldots & \ddots & & & & \ldots \\
\phi_{d(1)} & \phi_{d(2)} & \cdots & \phi_{d(e)} & \cdots & \phi_{d(v)} & \cdots \\
\vdots & \ldots & & & \ddots & & \ldots \\
\phi_{f\circ\phi_v(1)} & \phi_{f\circ\phi_v(2)} & & & & \phi_{f\circ\phi_v(v)} & \cdots \\
\vdots & \ldots & & & & & \ddots
\end{bmatrix}$$

Logo, $n = d(v)$ é o ponto fixo, pois já sabíamos pela equação 4.1: $\phi_{d(v)} = \phi_{\phi_v(v)}$.

$$\begin{bmatrix}
\phi_{\phi_1(1)} & \phi_{\phi_1(2)} & \cdots & & & \phi_{\phi_1(v)} & \cdots \\
\phi_{\phi_2(1)} & \phi_{\phi_2(2)} & \cdots & & & \phi_{\phi_2(v)} & \cdots \\
\vdots & \ldots & \ddots & & & & \ldots \\
\phi_{d(1)} & \phi_{d(2)} & \cdots & \phi_{d(e)} & \cdots & \phi_{\mathbf{d(v)}} & \cdots \\
\vdots & \ldots & & & \ddots & & \ldots \\
\phi_{f\circ\phi_v(1)} & \phi_{f\circ\phi_v(2)} & & & & \phi_{\mathbf{f(d(v))}} & \cdots \\
\vdots & \ldots & & & & & \ddots
\end{bmatrix}$$

Segundo Odifreddi, [Odi99], p. 153, Kleene foi o primeiro quem provou o teorema do ponto fixo com o propósito de tentar diagonalizar a classe das funções parciais computáveis. Queria com isto mostrar que há funções que poderiam ser algoritmicamente computáveis de alguma forma, porém não parcialmente computáveis. Esta imunização à diagonalização soma-se a evidências para a tese de Church-Turing. Exploremos mais isso.

A ideia provavelmente foi a seguinte: o processo de diagonalização parece sempre nos fornecer funções que não estavam na classe original de funções que supomos serem as únicas que existiam. Isso se mostra mesmo na prova inicial de Cantor, se não soubéssemos o resultado (é basicamente o que todos pensam antes de ver a prova), imaginaríamos que o conjunto potência de um conjunto enumerável é enumerável. A nossa intuição deste fato se dá pela simples prática matemática. Várias operações com conjuntos infinitos enumerável nos dá como resultado outro conjunto enumerável: intersecção, união, produto cartesiano, etc... Depois de vermos todas estas provas nós somos levados falsamente a esperar que a operação de tomar a parte de um conjunto enumerável produz outro conjunto enumerável. A quebra de expectativa se dá justamente quando introduzimos o método de diagonalização.

Notemos que a operação de tomar o conjunto das partes é o mesmo de pensar todas as funções característica de um determinado conjunto. Já falamos sobre isto, mas é óbvio, já que há uma correspondência de um para um entre funções características e subconjuntos. Logo:

$$2^X = \{f : f : X \to \{0, 1\} = 2\} = \wp(X)$$

Como X^X é maior do que 2^X, o processo de diagonalização fornece uma quebra de expectativas quando vamos pensar no conjunto das funções X^X, que possui cardinalidade maior do que X em geral.

Depois de ver que o processo de diagonalização quebra a expectativa de algo que parece intuitivo (dado uma prática matemática anterior com operações como união,etc..), podemos pensar que a diagonalização de alguma forma sempre será o candidato para quebrarmos a expectativa, de criarmos conjuntos "estranhos". E quebrar a aparentemente inocente tese de Church-Turing, e assim criar uma função intuitivamente computável que não estava na nossa descrição inicial de funções parciais computáveis. Ou seja, tentamos diagonalizar essa classe de funções para tentar achar uma nova função que é computável, mas não é uma função parcial computável pelos métodos já utilizados antes.

Mas é aí que a quebra de expectativa entra em cena outra vez, ou melhor dizendo, a quebra da quebra de expectativa. A classe das funções computáveis, ao contrário da classe das funções, não pode ser diagonalizada, justamente porque apesar de construirmos uma nova função parcial e computável, esta função não será uma função nova, diferente de todas as anteriores, ela será justamente alguma função já listada anteriormente, em uma enumeração. Ou seja, justamente uma função parcial $\phi_{f(n)}$ que é computável e computa a mesma coisa de alguma ϕ_n já enumerada anteriormente.

4.3 Auto-referência e índices

4.3.1 Quines

O teorema do ponto-fixo pode ser usado como instrumento para gerar programas autorreferenciais. Um exemplo deste tipo de programa é aquele que dá como resultado o seu próprio código. Estes programas são chamados de *quines*.

Definição **21.** *Quines* são programas do tipo: $\phi_e(x) = e$.

Proposição **11.** *Existem quines.*

Demonstração. Seja ψ uma função computável tal que: $\psi(x, y) = x$.(não é necessário ser parcial computável, porque é apenas a função projeção).

Pelo teorema do parâmetro 7, existe uma função computável f tal que:

$$\phi_{f(x)}(y) = \psi(x, y) = x.$$

Como f é computável, pelo teorema do ponto fixo, existe um ponto fixo e, ou seja, $\phi_{f(e)} = \phi_e$.

Logo:

$$\phi_e(y) = \psi(e, y) = e.$$

□

Proposição **12.** *Existem quines do seguinte tipo:*

$$\phi_e(x) = c(e, g(x)),$$

onde c é uma função codificadora.

Demonstração. Seja ψ computável tal que: $\psi(x, y) = c(x, g(y))$

Pelo teorema do parâmetro existe uma função f tal que:

$$\phi_{f(x)}(y) = \psi(x, y) = c(x, g(y))$$

Como f é computável, usamos o teorema do ponto-fixo:

$$\phi_e(y) = \psi(e, y) = c(e, g(y))$$

□

A proposição acima não apenas imprime seu próprio código, mas também simula a função g. Isto nos permite dizer, de forma não formal, que máquinas conseguem não apenas se auto-reproduzir, mas mais do que isso, reproduzir a si mesmas e outras máquinas possivelmente mais sofisticadas.[5] Além disso, como salienta [Odi99], p. 165:

> Auto-referência nunca é direta: ela vem de uma confusão controlada entre dois níveis de significados para inteiros: que podem ser vistos tanto como números como nomes para fórmulas.

A diferença entre estes dois níveis está na base da diferença entre uso e menção e tem uma longa tradição na filosofia. Quine (o filósofo) salienta esta distinção de forma intuitiva ao dizer: Boston possui por volta de 800.000 pessoas (uso), por outro lado, "Boston", a palavra, possui seis letras (menção).[6] Da mesma forma aqui, confundimos índices (programa sendo usado) e números dos índices (programa sendo mencionado). Sabemos que esta confusão não somente é possível, mas ocorrerá necessariamente na teoria da computabilidade, e por isto temos exemplos de programas auto-referenciais.

Com esta reflexão, usaremos o teorema do ponto-fixo para desbaratar a auto-referência de sentenças e programas nos próximos capítulos. Com isto conseguiremos o teorema 14 do ponto-fixo para aritmética, essencial para descortinar as sentenças auto-referentes do teorema da incompletude de Gödel. E também no teorema 18 construiremos uma versão do ponto fixo para o teorema de Chaitin.

A ideia geral é que as proposições acima nos permitem definir intuitivamente programas escritos do seguinte tipo:

$\phi_n =$ este programa é de código n e podemos imprimir seu código ou mesmo usá-lo para produzir outras funções $f(n)$.

[5] [Odi99] discute o exemplo paradigmático do *Conway's game of life*, um autômato celular que consegue simular dentro de si máquinas universais.

[6] Segundo o artigo de Raatkainen [Raa98] isto está em Quine, *Mathematical Logic*, W.W. Norton, New York, 1940.

4.3.2 Teorema de Rice

Outras aplicações do ponto-fixo consistem em nos ajudar a provar a não computabilidade de determinados conjuntos. Podemos usar pontos fixos para nos ajudar a demonstrar o teorema de Rice, uma generalização do problema da parada.

Definição **22. Conjunto índice:** Seja A um conjunto tal que $A \subseteq \mathbb{N}$. Chamamos de A um *conjunto índice* se temos a propriedade de que para todo $x \in A$ se $\phi_x = \phi_y$, então $y \in A$.

Teorema **10.** ***Teorema de Rice:*** *Se A é um conjunto índice diferente de $\emptyset$ e $\mathbb{N}$, então A não é computável.*

Demonstração. Prova por contradição. Suponhamos que A é computável. Ou seja a sua função característica χ_A é computável. Como A não é vazio e $A \neq \mathbb{N}$, pegue $a \in A$ e $b \notin A$. E considere:

$$f(x) = \begin{cases} a & \text{, se } \chi_A(x) = 0 \\ b & \text{, se } \chi_A(x) = 1 \end{cases}$$

Aplicando o teorema do ponto fixo, existe um n tal que:

$$\phi_{f(n)} = \phi_n$$

Suponha que $n \notin A$, logo $\chi_A(n) = 0$. Então, $\phi_a = \phi_{f(n)} = \phi_n$. Pela definição de conjunto índice: $n \in A$. Contradição.

Suponha que $n \in A$, logo $\chi_A(n) = 1$, e então $\phi_b = \phi_n$. Pela definição de conjunto índice $b \in A$. Contradição.

□

Ou seja, não é possível computar propriedades não triviais de conjuntos de funções parciais computáveis. Por exemplo, decidir se ϕ pára ou não, decidir se ϕ pára com um determinado argumento, decidir se ϕ retorna um determinado valor, decidir se ϕ é sobrejetora, etc...

4.3.3 Sistemas de índices aceitáveis

Vamos agora explorar melhor as possíveis enumerações de funções parciais computáveis. Lembrando que nós estipulamos uma enumeração no começo deste texto, a codificação lexicográfica 17. Outras codificações geram consequentemente outras enumerações de máquinas de Turing. Também, se mudássemos nosso modelo computacional, usássemos *Lambda Calculus*, por exemplo, a enumeração também mudaria. Ou melhor, qualquer modificação mínima nas definições anteriores resultariam em enumerações completamente distintas. A pergunta óbvia é: como posso garantir que estes resultados não dependam de peculiaridades das definições iniciais?

Definição **23. Sistema de índices aceitável:** Definimos π *um sistema de índices* se π é uma indexificação sobrejetora do tipo $\pi : \mathbb{N} \to \mathcal{FPC}$, onde $\mathcal{FPC}$ é o conjunto de funções p.c. Isto é, a seguinte indexificação pertence a uma determinada π:

$$\phi_0, ..., \phi_n, ...$$

Um sistema de índices π é *aceitável* com relação ao sistema π_0 se existem f e g funções computável totais que formam um *isomorfismo computacional* entre os dois sistemas de índices, isto é, tais que:

$$\phi_{f(e)} = \psi_e,\ \psi_{g(e)} = \phi_e$$

Quer dizer, se dado um sistema de índices inicial π_0 , π é um sistema de índices aceitável se é possível ir e voltar de π_0 e π a partir de f e g.[7]

Teorema **11.** ***Teorema de Rogers*** *Os seguintes itens são a ida e a volta de um mesmo teorema, que afirma: os resultados obtidos pela escolha de uma determinada enumeração independem do sistema de índices escolhido sse satisfazem o teorema do parâmetro e da enumeração.*

i *Se um sistema de índices π satisfaz o teorema 6 da enumeração e o teorema 7 do parâmetro, então π é aceitável.*

ii *Se π_0 é um sistema de índices para:*

$$\phi_0, ..., \phi_n, ...,$$

onde vale o teorema da enumeração e do parâmetro, e se π é um sistema de índices aceitável para

$$\psi_0, ..., \psi_n, ...,$$

obtido de π_0. Então, vale uma versão do teorema do parâmetro e da enumeração apropriado a π.

Demonstração. Provaremos (i), depois (ii) e, neste item, provaremos primeiro o teorema da enumeração e depois o teorema do parâmetro.

i Sejam ϕ e ψ funções universais, e e x números. Temos:

$$\phi_e(x) = \phi(e, x) = \psi_{f(e)}(x),$$

pelos teoremas da enumeração (primeira igualdade) e parâmetro (segunda igualdade). Da mesma forma obtenha g.

[7] Em uma notação funcional isto quer dizer que $\pi_0 f = \pi$ e $\pi g = \pi_0$.

ii Assuma que a indexificação π para as ψ_i é aceitável.

(Enumeração) Temos:

$$\psi_e(x) = \phi_{f(e)}(x) = \phi(f(e), x),$$

pois π é aceitável (primeira igualdade) e pelo teorema da enumeração para π_0 (segunda igualdade). Mas notemos que $\phi_{f(e)}$ é computável parcial pela segunda igualdade e teorema da enumeração. E como π é um sistema de índices, e assim sobrejetora, temos que existe um índice m para $\phi_{f(e)}$ no sistema π. Mas como $\phi_{f(e)}$ é uma função arbitrária, mostramos que para qualquer função parcial existe uma enumeração π. Tome uma função universal ψ dentro desta enumeração como base para ter:

$$\psi_e(x) = \psi(e, x).$$

(Parâmetro) As seguintes igualdades valem:

$$\psi_e(x, y) = \phi_n(e, x, y) = \phi_{s(n,x)}(y) = \psi_{g(s(e,x))}(y)$$

Justificaremos as igualdades em ordem da esquerda para direita:

(a) pois provamos na parte da enumeração que ψ é enumerável, logo ψ_e deve coincidir com algum ϕ_n.

(b) pelo teorema do parâmetro para π_0.

(c) pois π é aceitável.

Mas isto quer dizer que para $s' = g \circ s$:

$$\psi_e(x, y) = \psi_{s'(e,x))}(y)$$

□

Como usamos apenas o teorema do parâmetro e da enumeração para demonstrar o teorema do ponto fixo, temos:

Corolário **4.** *Um sistema de índices aceitável satisfaz o teorema do ponto-fixo.*

A moral da história: não importa qual tipo de enumeração se use para provar os resultados básicos da computabilidade, desde que seja adequada. Teremos os resultados da teoria da mesma forma. Rogers resume: diferentes numerações de Gödel funcionam para a computabilidade como diferentes sistemas de coordenadas descrevem o mesmo objeto na geometria analítica.

É por isto que alguns autores partem já da caracterização de funções parciais computáveis como aquelas obedecendo basicamente estes dois teoremas fundamentais: parametrização e enumeração. O restante, ponto-fixo, etc... é consequência. O contrário não vale, se assumirmos o ponto fixo, não podemos

mostrar a parametrização e a enumeração.[8] É por isso que se considera que estes dois resultados formam uma base para o restante da teoria da computabilidade.[9]

[8] A não ser se assumirmos ponto fixo e composicionalidade.

[9] Resultados também aproximam estes dois resultados às categorias cartesianas fechadas. Ver [Odi99] p. 218.

Capítulo 5

Incompletude de Gödel

Neste capítulo utilizaremos os métodos desenvolvidos nos outros capítulos para provar o teorema da incompletude de Gödel. Na primeira seção, mostraremos, embora não explicitamente, como máquinas podem simular a noção de prova da aritmética, e como as funções computáveis podem ser definidas dentro da aritmética. Na segunda seção, veremos que é possível representar as funções computáveis que computam a prova de um teorema (se realmente existir uma prova). Para fazer isto, precisaremos do lema da separação, e daremos como resultado preliminar o teorema da incompletude de Gödel-Rosser. Na terceira seção, veremos como é possível destilar a auto-referência presente na prova de incompletude utilizando a aplicação do teorema do ponto-fixo do capítulo anterior para este contexto aritmético. Terminaremos provando uma sequência de resultados de incompletude.

Intuitivamente uma prova da incompletude da aritmética mostra que não há uma teoria tal qual a geometria euclidiana no contexto da aritmética. Ou seja, uma teoria em que todas fórmulas possíveis podem ser provadas ou refutadas a partir de um conjunto de axiomas. Na geometria euclidiana, se tirarmos um axioma, por exemplo o postulado das paralelas, não teremos mais uma teoria completa. Acrescentado este axioma, qualquer adição de axioma é redundante. O teorema da incompletude nos diz que, por mais que tentemos acrescentar axiomas a um conjunto primeiro de axiomas para completarmos aquilo que entendemos intuitivamente ser a teoria da aritmética, este projeto sempre resultará em fracasso. Sempre haverá uma uma fórmula que é indecidível, ou seja, que não pode ser provada nem refutada.

Há vários caminhos para mostrar a incompletude da aritmética. Vamos citar alguns:

1. O caminho original de Gödel, em que se exibe exatamente e construtivamente a fórmula que não é demonstrável nem refutável. Gödel teve que

assumir algumas hipóteses a mais (ω-consistência), justamente para manter a intuição da sentença de Gödel que faz o trabalho que ele queria, a sentença que intuitivamente diz de si mesma que não pode ser provada nem refutada. Rosser conseguiu eliminar tal hipótese a mais com o custo de também perder a intuição original.

2. O caminho em que não se exibe a sentença indecidível, um caminho não construtível. Provamos essencialmente que a teoria da aritmética é não decidível. Uma teoria é *decidível* se há alguma função parcial computável que consegue estipular em um número finito de passos se uma determinada fórmula faz parte ou não do conjunto de teoremas da teoria. Shoenfield [Sho67] realiza este caminho.

3. O caminho em que provamos que o conceito de representabilidade aritmética é um modelo formal de computação equivalente ao modelo das máquinas de Turing. Assim, o problema da parada representará dentro da aritmética uma determinada fórmula com as características que queremos. Ela não pode ser provada nem refutada, caso contrário seria possível provar o problema da parada.

Sob quaisquer aspectos, uma prova da incompletude de Gödel deve cumprir os seguintes requisitos, colocaremos as escolhas feitas por nós em parênteses:

- Ter algum sistema formal (a teoria aritmética **R** em lógica de primeira ordem);
- Ter um modelo computacional universal (máquinas de Turing);
- Uma codificação (representação lexicográfica).

Estas escolhas podem ser diferentes das feitas por nós, por exemplo poderíamos trabalhar com lambda calculus, codificações de pareamento standard e teorias dos conjuntos. Em especial poderíamos trabalhar, como faz [Smu92], com a própria aritmética como modelo computacional (talvez esse tipo de prova seja a mais autêntica no sentido de evidenciar a intuição de que estamos construindo algo que fala de si mesmo). Porém, estas escolhas não podem ser completamente arbitrárias. Os requisitos são óbvios quando olhamos o nosso objetivo final: é necessário um sistema formal que consegue nomear suas próprias fórmulas. E com nomear estamos nos referindo a nossa própria experiência mundana de por aspas em torno de um texto e apontar para ele como se fosse um objeto. Ou, também, a nossa experiência em coordenar sentenças através de pronomes: que, isto, aquilo, etc... Em suma, é necessário que possamos transitar livremente entre o uso e a menção de fórmulas.[1] E isto será feito pela escolha de alguma codificação adequada.

Com esta reflexão é possível ver que estas escolhas devem obedecer os seguintes princípios:

[1] Quine foi talvez o primeiro a explicitar este requisito da incompletude.

- É possível simular as provas do sistema formal no modelo computacional adotado (para nós será imediato pela codificação e pela tese de Church versão instrumental);
- É possível codificar todos os componentes do modelo computacional, transformando estes em nomes para o sistema formal (em nosso caso, isto também é óbvio pela nossa longa experiência computacional com números binários);
- É possível provar, na linguagem do sistema formal, uma fórmula com quantificadores limitados que decide se um dado nome (número) é ou não uma computação (isto é menos óbvio, mas ainda assim, perfeitamente intuitivo se pensarmos que a definição de computação 2.1.2 é algo completamente descritivo e plausível de ser formalizado).

Colocando tudo junto vemos que o sistema formal conseguirá falar sobre computações, que por sua vez podem falar de coisas do sistema formal. Em especial, poderemos falar sobre a própria provabilidade do sistema formal dentro do sistema formal, e a partir daí temos a famosa frase indecidível: "esta sentença não é demonstrável". Ou em termos lógicos:

$$G \leftrightarrow \neg P(\ulcorner G \urcorner)$$

Porém para evidenciar esta versão intuitiva faremos o seguinte caminho:

1. Definiremos rapidamente nossos requisitos formais;
2. Mostraremos que é possível encontrar uma sentença concreta indecidível para sentença de Gödel-Rosser (menos intuitiva que esta acima, mas mais promissora em termos de concretude);
3. Mostraremos o ponto-fixo com a explicação desta intuição com conceitos que vimos antes.

5.1 Preliminares Aritméticos

5.1.1 Sintaxe e Hierarquia Aritmética

Definição **24. Linguagem aritmética:** Definimos a linguagem aritmética como a seguinte linguagem: $L = \{s, +, \cdot, 0\}$. Adotamos a nomenclatura para números:

$$\overline{\mathbf{n}} := \overbrace{ss...s}^{n\text{vezes}} 0$$

Definição **25. Hierarquia Aritmética:** Dizemos que um quantificador $Q \in \{\exists, \forall\}$ está *limitado* em uma fórmula F se F pode ser escrita das seguintes formas com G livre de quantificadores:

$$(\forall \mathbf{x} < \overline{\mathbf{n}})\mathbf{G}(\mathbf{x}) = \forall \mathbf{x}(\mathbf{x} < \overline{\mathbf{n}} \rightarrow \mathbf{G}(\mathbf{x})).$$

$$(\exists \mathbf{x} < \overline{\mathbf{n}})\mathbf{G}(\mathbf{x}) := \exists \mathbf{x}(\mathbf{x} < \overline{\mathbf{n}} \wedge \mathbf{G}(\mathbf{x})).$$

Definimos uma fórmula como sendo Σ_0 se há uma forma equivalente tal que pode ser definida usando os símbolos: $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, Q\mathbf{x} < \overline{\mathbf{n}}$.

Definimos uma fórmula F como sendo Σ_1 se existe uma Σ_0-fórmula G tal que F pode ser escrita da forma: $\exists \mathbf{x_1}, ..., \exists \mathbf{x_n} G(\mathbf{x_1}, ..., \mathbf{x_n})$.

Analogamente, F é Π_1 se existe uma Σ_0-fórmula G tal que F pode ser escrita: $\forall \mathbf{x_1}, ..., \forall \mathbf{x_n} G(\mathbf{x_1}, ..., \mathbf{x_n})$.

Definimos indutivamente uma fórmula F como sendo Σ_n se existe uma Π_{n-1}-fórmula G tal que F pode ser escrita: $\exists \mathbf{x_1}, ..., \exists \mathbf{x_n} G(\mathbf{x_1}, ..., \mathbf{x_n})$. Similarmente para Π_n-fórmulas.

5.1.2 Semântica e Definibilidade

Definição **26. Modelo standard da aritmética:** Escreveremos $\mathcal{N}$ para a estrutura da *aritmética standard* ou o standard model da aritmética: é a estrutura para linguagem aritmética $L = \langle <, s, +, \cdot, 0\rangle$ com domínio enumerável $\mathbb{N} = \{0, 1, 2, ...\}$ onde os indivíduos, funções e predicados são interpretados do jeito usual.[2] Definimos a *teoria standard da aritmética* dos naturais sob a notação $Teo(\mathcal{N})$ (ver definição 54).

Definição **27.** Dizemos que um predicado P é *definível*[3] em uma L-estrutura $\mathcal{A}$ se para todos $a_1, .., a_n$ elementos do domínio A, existe uma L-fórmula $\mathbf{F}$ tal que:

$$P(a_1, ..., a_n) \Leftrightarrow \mathcal{A} \models \mathbf{F}(a_1, ..., a_n).$$

Em especial, dizemos que P é *aritmeticamente definível* se para todos $a_1, .., a_n$ elementos do domínio A, existe uma fórmula com símbolos $\{s, +, \cdot, 0\}$ tal que:

$$P(a_1, ..., a_n) \Leftrightarrow \mathcal{N} \models \mathbf{F}(a_1, ..., a_n).$$

Σ_n-fórmulas possuem a propriedade de serem fechadas por determinadas operações lógicas que não introduzem outros tipos de quantificadores que não os existenciais. A possibilidade de fazer isto está em parte justificada na possibilidade de efetuar equivalências destas fórmulas a partir do método prenex e colocá-las em uma forma normal. Iremos enunciar este resultado como uma proposição:

[2] I.e. 0 é associado ao indivíduo 0, $s0$ ao 1, $ss0$ ao 2, etc... s é associado à função 'sucessor', $+$ associada à função 'mais' , $\cdot$ é associada a função 'vezes' e o predicado $<$ é interpretado como o predicado de 'menor que'. Nós fazemos referência à noção intuitiva da metalinguagem para explicar quem são estas funções e este predicado.

[3] [Smu92] usa a nomenclatura *exprimível*.

PROPOSIÇÃO **13.** *As classes de fórmulas Σ_n e Π_n são fechadas pelos os conectivos $\vee$ e $\wedge$. A classe Σ_n é fechada pela quantificação existencial e Π_n é fechada por quantificações universais. Além disso, a negação de uma Σ_n-fórmula é uma Π_n-fórmula.*[4]

Um resultado clássico que une a teoria da computabilidade aos resultados semânticos aritméticos é o fato que podemos definir na linguagem da aritmética uma fórmula que traduz computações de máquinas de Turing — e além disso podemos fazer tudo isto nos restringindo a Σ_0-fórmulas da aritmética.

A prova disso envolve basicamente um procedimento semelhante à produção de uma máquina universal. E de fato, se considerarmos a definibilidade aritmética como sendo um modelo computacional, o que se está de fato fazendo é construir dentro da linguagem aritmética um simulador de máquinas de Turing. Como [Bek06] diz, podemos ver a aritmética como uma linguagem computacional declarativa, que nos dá fórmulas mas não estipula como elas devem ser avaliadas, em contraposição às linguagens imperativas como as das máquinas de Turing, C, $FORTRAN$, etc... De fato, fórmulas do tipo Σ_0 podem ser vistas como operadores de busca mínima, e são semelhantes neste sentido a funções recursivas de busca mínima.

Apesar de não ser complicado de se provar este fato, a construção é razoavelmente longa. Pode-se por exemplo consultar [Bek06] para um tratamento detalhado para máquinas de Turing e uma codificação semelhante à codificação lexicográfica que demos. Resumiremos isto enunciando a seguinte proposição:

PROPOSIÇÃO **14.** *Qualquer máquina de Turing pode ser definível por uma Σ_0-fórmula aritmética.*

Apenas para termos alguma intuição de como isto é feito, o procedimento é o seguinte:

- Descobrimos como definir aritmeticamente o tamanho de um número em base binária.[5] Definimos uma codificação lexicográfica binária $\ulcorner\cdot\urcorner : A^* \to \mathbb{N}$ e codificamos todas letras do alfabeto utilizado nos programas de uma máquina de Turing. Logo, basta definir outras funções como: concatenação de códigos, tamanho de um string, busca de substrings dentro de um string, sequências de strings.
- Como as definições que demos de computação 2.1.2 e de máquina de Turing 5 podem ser codificadas pelo método acima, basta transferir extensivamente as definições para aritmética (ou seja, a maioria das definições

[4] Prova desta proposição pode ser achada por exemplo em [Smu92], p.50.

[5] Podemos utilizar por exemplo um método dado por [Smu92] creditado a Myhill:

$$Pow_2(x) :\leftrightarrow (\forall z \leq x)((z|x \wedge z \neq 1) \to 2|z)$$

A partir daí definimos

$$y = 2^{|x|} :\leftrightarrow (Pow_2(y) \wedge y < x \wedge y > 1) \wedge (\forall z \leq y)\neg(Pow_2(z) \wedge z < x \wedge z > 1),$$
$$|x| = y :\leftrightarrow (y = 0 \wedge x = 0) \vee (2^{|y|} \leq 2^{|x|} \wedge 2^{|y+1|} > 2^{|x|}).$$

irão envolver diversas disjunções que tratarão dos diversos casos que podem ocorrer para passarmos de um passo para outro, se a máquina ler o código de uma instrução do tipo tal faça tais coisas com o código da configuração).

- Assim, definimos um predicado $Comp(m, n)$ se a máquina de Turing de código m pára e computa o input de código n. Logo:

$$T_m(n) \downarrow \Leftrightarrow \mathcal{N} \models Comp(m, n)$$

Note que obviamente não sabemos de antemão se $Comp$ irá parar, caso contrário violaríamos o problema da parada. Isto nos dá uma descrição dos conjuntos computacionalmente enumerável em termos de definibilidade aritmética.

TEOREMA **12.** ***Caracterização de predicados c.e.:***

- *Um predicado P é c.e. sse é definível por uma Σ_1-fórmula aritmética.*
- *Um predicado P é computável sse é definível por uma Σ_0-fórmula aritmética.*

Demonstração. (Intuitiva)
(ii)É uma consequência das considerações acima.
(i) De fato de um lado pela nossa descrição intuitiva de $Comp$ que demos acima:

$$P(y) \Leftrightarrow \mathcal{N} \models \exists x Comp(x, y)$$

E a volta se dá notando que para cada número natural m, $Comp(m, y)$ é Σ_0. E para cada natural n, podemos calcular $Comp(m, n)$ a partir de sua tabela de verdade, já que eliminaremos todos os quantificadores existenciais ou universais limitados de $Comp$ substituindo por n. Como isto é intuitivamente factível, então há uma função computável que faz este procedimento para todo n. Logo, $P(y)$ é imagem desta função computável, e portanto c.e pela proposição 10.
□

5.1.3 Axiomas e Computabilidade

Trabalharemos com uma axiomática para lógica de primeira ordem com um conjunto de axiomas extensos.

Esquemas de Axiomas para lógica clássica de primeira ordem:

L1 *Axioma da tautologia*: Todas as tautologias.

L2 *Axioma da substituição:* $\mathbf{F_x[a]} \rightarrow \exists \mathbf{xF}$.

L3 *Axioma da identidade:* $\mathbf{x = x}$.

L4 *Axioma da igualdade:* $(\mathbf{x_1} = \mathbf{y_1} \wedge ... \wedge \mathbf{x_n} = \mathbf{y_n}) \rightarrow \mathbf{f}(\mathbf{x_1}, ..., \mathbf{x_n}) = \mathbf{f}(\mathbf{y_1}, ..., \mathbf{y_n})$.

L5 *Axioma de Leibniz:* $(\mathbf{x_1} = \mathbf{y_1} \wedge ... \wedge \mathbf{x_n} = \mathbf{y_n}) \rightarrow \mathbf{P}(\mathbf{x_1}, ..., \mathbf{x_n}) \leftrightarrow \mathbf{P}(\mathbf{y_1}, ..., \mathbf{y_n})$.

Regras de inferência para lógica de primeira ordem:

r1 *Regra da substituição:* Se $\mathbf{x}$ não é livre[6] em $\mathbf{G}$, então deduza:

$$\frac{\mathbf{F} \rightarrow \mathbf{G}}{\exists \mathbf{x} \mathbf{F} \rightarrow \mathbf{G}}$$

r2 *Modus Ponens:* Deduza:

$$\frac{\mathbf{F} \rightarrow \mathbf{G} \text{ e } \mathbf{F}}{\mathbf{G}}$$

Definição **28.** Uma *teoria* é um conjunto de fórmulas fechada pelas regras de inferência de lógica de primeira ordem, contendo todas fórmulas subentendidas pelo esquema de axiomas para lógica clássica de primeira ordem (chamado de *axiomas lógicos*) e contendo um conjunto de fórmulas Γ subentendidos por outros esquemas de axiomas a serem estipulados (chamados de axiomas não lógicos). Os elementos de uma teoria são chamados de *teoremas*.

O esquema de axiomas da tautologia contém uma quantidade infinita de instâncias de axiomas de infinitos tipos. Então $P \vee \neg P$ é uma instância desse axioma. Assim como: $\neg\neg P \leftrightarrow P$. Já os outros axiomas possuem uma quantidade infinita de axiomas de um tipo definido. Por exemplo $x_1 = x_1$ está dentro deste esquema.

A consideração do esquema de axiomas da tautologia não nos atrapalha computacionalmente em comparação a outros esquemas axiomáticos como o hilbertiano. Isto porque para qualquer sistema axiomático de primeira ordem podemos derivar o teorema da tautologia: toda tautologia é um teorema. E além disso sabemos que o procedimento para descobrir que algo é uma tautologia é computável. Em termos simples: faça uma tabela de verdade. E é computacionalmente factível produzir um algoritmo que dada uma fórmula, computa se sua tabela de verdade é de uma tautologia ou não. Logo, isto significa que o conjunto das tautologias é computável. Em outras palavras, a parte proposicional da lógica de primeira ordem é decidível, apesar de sua parte não proposicional não ser.

Vamos agora enunciar alguns fatos sobre a relação entre este sistema axiomático e a nossa teoria da computabilidade.

Proposição **15.** *Se o conjunto de axiomas não-lógicos de uma teoria é computável, o predicado $Pr'(\ulcorner X \urcorner, y)$, que verifica se y é um código de prova da fórmula X, é computável.*

[6] Isto é, a ocorrência de x em G está ligada por um quantificador, se houver.

Demonstração. (Intuitiva) A prova deste fato envolve a construção sistemática de predicados dentro da linguagem da aritmética com o objetivo de verificar se algum código realmente é o código de uma prova. Para isto tem-se que verificar que tal código foi construído obedecendo as regras de inferência a partir dos axiomas. Tal tipo de construção pode ser verificada por exemplo em [Sho67][p.123-126]. Vamos dar apenas uma intuição de porque é possível verificar estes axiomas. Como vimos, o conjunto dos axiomas lógicos é computável pelo argumento que demos acima (tabelas de verdade); e o conjunto dos axiomas não lógicos é computável por hipótese. Além disso, podemos verificar se é verdade ou não que na sequência de fórmulas codificada por y temos as hipóteses da regra de Modus Pones: $F \to X$ e X. Se sim, então y é uma prova para $\ulcorner X \urcorner$. Também, se X é conclusão da regra de substituição e temos $\ulcorner F \to G \urcorner$ em algum lugar da sequência codificada de y, podemos construir um algoritmo que verifica se x não está livre em G. Então, se isto é o caso, y é um código de prova para $\ulcorner X \urcorner = \ulcorner \exists x F \to G \urcorner$. Logo, temos uma função computável característica que decide se $(\ulcorner X \urcorner, y) \in Pr'$ ou não. □

Corolário **5.** *O predicado $Rf'(\ulcorner X \urcorner, y)$, que verifica se a fórmula $\neg X$ é provada pelo código de prova y, é computável. (i.e. que verifica se X é refutado pelo código de prova y).*

Corolário **6.** *Pr' e Rf' são aritmeticamente definíveis.*

Demonstração. Pelo teorema 12 existe uma fórmula que os definem, pois são computáveis. □

Definição **29.** Sejam Pr e Rf as fórmulas aritméticas que definem respectivamente Pr' e Rf' cuja existência é possível pelo corolário anterior.

Os conjuntos P e R dos números de Gödel das fórmulas prováveis e refutáveis são definidos na aritmética pelas seguintes Σ_1-fórmulas:

$$P(\ulcorner X \urcorner) \Leftrightarrow \mathcal{N} \models \exists y Pr(\ulcorner X \urcorner, y),$$
$$R(\ulcorner X \urcorner) \Leftrightarrow \mathcal{N} \models \exists y Pr(\ulcorner \neg X \urcorner, y).$$

Dizemos que P é o *predicado de provabilidade* para uma teoria T. E R é o *predicado de refutabilidade.*

Temos que P e R são Σ_1 por definição e pelo teorema 12 são c.e.

Corolário **7.** *P e R são c.e.*

Agora restringiremos estes conjuntos P e R a apenas fórmulas de Gödel autorreferentes, que aplicam o seu próprio número de Gödel x a si mesmo. Enunciamos tais sentenças da seguinte forma: $F_x(x)$.

Corolário **8.** *Os conjuntos P^* e R^*, dos números de Gödel das fórmulas do tipo específico autorreferente $F_x(x)$ que são prováveis ou refutáveis, [7]. podem ser definidos na aritmética pelas seguintes Σ_1-fórmulas:*

[7] Note a diferença de P e R, pois $P^* \subseteq P$ e $R^* \subseteq R$.

$$P^*(x) \Leftrightarrow: \mathcal{N} \models \exists y Prov(x, y)$$
$$R^*(x) \Leftrightarrow: \mathcal{N} \models \exists y Ref(x, y)$$

onde $Prov$ e Ref são fórmulas aritméticas afirmando que uma máquina de Turing verifica que y é um código de uma respectiva prova ou refutação da fórmula $F_x(x)$, onde x é o número de Gödel da fórmula $F_x(z)$.

Demonstração. A possibilidade de fazer esta definição é similar a argumentação da existência de P e R. □

DEFINIÇÃO **30.** Para fixar a notação defina $P^*, Prov, Ref$ e R^* como na proposição acima (note a diferença $Prov \neq Pr$ e $P \neq P^*$![8]).

5.2 A sentença indecidível de Gödel-Rosser

5.2.1 O sistema aritmético R

Apresentaremos uma extensão conservativa da aritmética de Robinson[9] utilizada por [BJB12] p.266 e [Sho67] p.22 que chamaremos **Q** :

Q1 $s(x) \neq 0$;

Q2 $s(x) = s(y) \rightarrow x = y$;

Q3 $x + 0 = x$;

Q4 $x + s(y) = s(x + y)$;

Q5 $x \cdot 0 = 0$;

Q6 $x \cdot s(y) = (x \cdot y) + y$;

Q7 $x \not< 0$;

Q8 $x < s(y) \leftrightarrow (x = y \vee x < y)$;

Q9 $(x < y) \vee (x = y) \vee (y < x)$;

[8] De fato, o predicado $Prov$ aparece no lema 34, e já possui dentro de si uma diagonalização, pois é referente à máquina de Turing que acha $\ulcorner X_x(x) \urcorner$. Isto induz uma prova da incompletude de Gödel-Rosser sem o teorema do ponto fixo. Por outro lado com Pr procuramos $\ulcorner X \urcorner$, e a diagonalização será feita posteriormente, após provarmos o ponto-fixo para lógica.

[9] Utilizamos aqui o mesmo símbolo usual para a Aritmética de Robinson **Q**, devido à simplicidade notacional. Porém , a Aritmética de Robinson não contém os axiomas Q7-Q9 e contém o seguinte axioma Q0 a mais: $x = 0 \vee \exists y(s(y) = x)$. **N**. Esta variante que utilizaremos pode ser chamada **Q**+ e é extensão conservativa de **Q**, mas o contrário não vale. Ambas são subsistemas de **PA**.

Se acrescentarmos o axioma de indução (PA10) para o esquema de fórmulas **F**:

$$\mathbf{F}_x[0] \wedge \forall(x)(\mathbf{F} \rightarrow \mathbf{F}_x[s(x)]) \rightarrow \mathbf{F}$$

teremos **PA**.

Utilizamos este sistema para derivar um subsistema que trabalharemos: **R**. O sistema **R** é subsistema de **Q** (i.e. um sistema tal que Q é sua extensão). Que **R** é subsistema pode ser demonstrado usando indução (externa) a partir dos axiomas de **Q**[10]. Usaremos a notação $\overline{n} = s \circ s \circ ... \circ s(0)$, n vezes composto. Estes são números dentro do sistema formal (não confundir com strings). Assim, a parte da esquerda do primeiro axioma a seguir é equivalente a dizer que $\vdash_T \overline{m} + \overline{n} = k$. [11] Além disso usamos o negrito para indicar que são esquemas de axiomas, ou seja na realidade cada um destes axiomas representa infinitos axiomas de fato e a variável em negrito é uma metavariável que indica este fato. E atenção: *também aqui adotaremos uma notação flexível, apenas nas definições tomaremos o cuidado de fazer esta separação de números na metateoria (i.e. n) e na linguagem objeto (i.e. $\overline{\mathbf{n}}$).*

O esquema axiomático R:

$R1$ $\overline{\mathbf{m}} + \overline{\mathbf{n}} = \overline{\mathbf{k}}$, onde $\mathcal{N} \models m + n = k$;

$R2$ $\overline{\mathbf{m}} \cdot \overline{\mathbf{n}} = \overline{\mathbf{k}}$, onde $\mathcal{N} \models m \cdot n = k$;

$R3$ $\overline{\mathbf{m}} \neq \overline{\mathbf{n}}$, onde $\mathcal{N} \models m \neq n$;

$R4$ $\mathbf{x} < \overline{\mathbf{n+1}} \leftrightarrow (\mathbf{x} = 0 \vee ... \vee \mathbf{x} = \overline{\mathbf{n}})$;

$R5$ $(\mathbf{x} < \overline{\mathbf{n}}) \vee (\mathbf{x} = \overline{\mathbf{n}}) \vee (\overline{\mathbf{n}} < \mathbf{x})$

As provas que se seguem acompanham o livro do Smullyan [Smu92]. Em muitos momentos justificaremos um teorema ou consequência *por lógica*. Isto quer dizer que a afirmação decorre ou é uma fórmula válida simplesmente por algum axioma ou resultado fundamental da lógica (por exemplo $\forall x A(x) \rightarrow A(a)$) , uma tautologia, ou uma consequência tautológica de algum axioma ou enunciado anterior. Não explicitaremos cada passo.

5.2.2 Σ_1-completude

Definição **31. Σ_0-completa:** Dizemos que a teoria T é Σ_0-completa se para toda Σ_0-sentença **F**:

$$\mathcal{N} \models \mathbf{F} \Rightarrow \vdash_T \mathbf{F}$$

[10] Para isto, ver [Smu92] p.70. Para alguma intuição do processo: $R1$ pode ser derivado de $Q4$ por indução em m assumindo a hipótese de indução e vendo que $\vdash_Q n + m + 1 = k + 1$ (o caso base é $Q3$).

[11] Ou seja, estes axiomas tem como objetivo reunir informações básicas para provar uma completude restrita a determinada parte da aritmética, como veremos a seguir, a Σ_0-completude.

Equivalentemente para Π_1-completa, Σ_1,-completa, etc ...

PROPOSIÇÃO **16.** **R** *é* Σ_0*-completo.*

Demonstração. Vamos dividir em duas partes gerais a prova: (i) para sentenças atômicas (i.e. apenas as fórmulas: $n < m$, $n = m$, $n + m = k$, $n \cdot m = k$), (ii) para sentenças não-atômicas.

i- Sentenças atômicas: Nesta parte provaremos mais do que o necessário (para nos ajudar na parte (ii)): (ia) toda Σ_0-fórmula atômica verdadeira é demonstrável, e (ib) toda a Σ_0-fórmula atômica falsa é refutável.

ia- Toda Σ_0*-sentença atômica verdadeira é demonstrável:*

- $\vdash_R n = n$, por lógica.
- $\vdash_R n < m$, para algum m maior que n, por $R4$ e notar que cada disjunto em $n = 0 \vee ... \vee n = m - 1$ é demonstrável pelo ítem anterior, logo a disjunção toda também é demonstrável.
- $\vdash_R n + m = k$, para algum k, por $R1$.
- $\vdash_R m \cdot n = l$, para algum l, por $R2$.

ib- Toda a Σ_0*-sentença atômica falsa é refutável:*

- $\vdash_R n \neq n$, por $R3$.
- $\vdash_R n \not< m$, por $R4$ e notar que $n \neq 0 \vee ... \vee n \neq m - 1$ é demonstrável pelo item anterior.
- $\vdash_R n + m = p$, para $p \neq k$, por R_4.
  $\vdash_R p \neq k$, por $R3$.
  $\vdash_R n + m \neq k$, por transitividade.
- $\vdash_R m \cdot n \neq k$, como o item anterior.

(ii)- Sentenças em geral:
Pela definição de Σ_0-sentença, temos três possibilidades: 1) atômica, 2)do tipo: $\neg X$ ou $X \vee Y$, 3) $(\exists x < n)F(x)$.

Provaremos por indução no grau de complexidade da fórmula que toda Σ_0-sentença verdadeira é demonstrável, e toda falsa é refutável. A parte que desejamos (verdadeiro implica demonstrável) sairá como caso particular.

Notemos que o caso base em que 1) S é atômica sai do item anterior; e 2) sai imediatamente por lógica.[12] Para o caso 3), suponha que S seja verdadeira e do tipo $(\exists x < n)F(x)$. Logo, existe pelo menos um $F(m)$ com $m < n$ tal que $F(m)$ é verdadeira e, pela hipótese de indução, demonstrável. Logo, por

[12]Rascunho: suponha que $\neg X$ esteja na condição da hipótese de indução e seja verdadeira, logo X é falsa, e por hipótese é refutável. Logo $\neg X$ é demonstrável; e demais casos semelhantemente.

introdução de existencial e por $m < n$ ser demonstrável temos que $(\exists x < n)F(x)$ é demonstrável.

Suponha que S seja falsa e do tipo requerido. Então, todas as sentenças:

$$F(0), ..., F(n-1)$$

são falsas e então refutáveis pela hipótese de indução. Logo:

$$(x = 0 \vee .. \vee x = n-1) \rightarrow F(x)$$

é refutável e $(\exists x < n)F(x)$ também.

□

Corolário **9.** **R** *ou qualquer extensão sua é* Σ_1*-completa.*

Demonstração. Para uma fórmula Σ_0 o resultado sai imediatamente pelo lema anterior. Suponha $R(a_1, ..., a_n)$ uma Σ_1-fórmula, então existe S um predicado Σ_0 tal que:

$$R(a_1, ..., a_n) \leftrightarrow \exists y S(a_1, ..., a_n, y)$$

Suponha que $R(a_1, ..., a_n)$ seja verdadeira. Então, $S(a_1, ..., a_n, a)$, para algum a também será verdadeira por definição. Logo, a fórmula $S(a_1, ..., a_n, a)$ é verdadeira e, pela hipótese de Σ_0-completude, demonstrável. Assim, podemos introduzir o existencial trocando a por $\exists y$.[13] Logo $\exists y S(a_1, ..., a_n, y)$ é demonstrável.

Podemos assumir como hipótese de indução que qualquer Σ_1-fórmula R' de um determinado grau de complexidade de quantificadores existenciais será demonstrável, Logo R tal que $R(x_1, ..., x_n) \leftrightarrow \exists y R'(x_1, ..., x_n, y)$ também será pela mesma argumentação anterior.

Obviamente qualquer extensão de **R** conservará esse resultado.

□

Definição **32. Enumerabilidade de predicados e conjuntos**

Dizemos que uma fórmula F enumera um predicado (ou conjunto) R se para todos $a_1, .., a_n, b \in \mathbb{N}$:

- $R(a_1, ..., a_n) \Rightarrow$ existe um inteiro positivo b tal que $\vdash_T F(a_1, ..., a_n, b)$;
- $\neg R(a_1, ..., a_n) \Rightarrow$ para todo inteiro positivo b $\vdash_T \neg F(a_1, ..., a_n, b)$.

De acordo com [Smu92] p.64:

Lema **2.** *Se* T *é* Σ_1*-completo, então todas fórmulas* Σ_1 *são enumeráveis em* T*.*

[13] Uma justificativa para esta operação aparece no teorema da substituição do Shoenfield, [Sho67], p.32.

Demonstração. Seja R Σ_1 nas mesmas condições do corolário da proposição 16. Então existe uma F tal que $\exists y F(a_1, ..., a_n, y)$ é demonstrável.

- Imediato pelo corolário, já que F é Σ_1 e será demonstrável, logo o será em especial para algum b.
- Suponha que $\neg R$ vale, então $\neg F(a_1, ..., a_n, b)$ será verdade para todo b. Portanto, como S é Σ_0, $\neg F(a_1, ...a_n, b)$ será demonstrável.

Logo, pela definição de relação enumerável temos que $R(x_1, .., x_n)$ é enumerável. □

5.2.3 Separação e Representabilidade

DEFINIÇÃO **33. Separabilidade de predicados e conjuntos**

Dizemos que uma fórmula **F** separa a relação R de S em T se para todos $a_1, ..., a_n \in \mathbb{N}$:

- $R(a_1, ..., a_n) \Rightarrow \vdash_T F(a_1, ..., a_n)$;
- $S(a_1, ..., a_n) \Rightarrow \vdash_T \neg F(a_1, ..., a_n)$.

Por [Smu92] p. 82.

LEMA **3.** ***Lema da separação:*** *seja T qualquer extensão de R, se os conjuntos A^* e B^* são conjuntos disjuntos enumeráveis pelas fórmulas respectivamente $A(x, y)$ e $B(x, y)$, em T, então A^* e B^* são separáveis pela fórmula de Gödel-Rosser:*[14]

$$GR(x) := \forall y(A(x, y) \rightarrow (\exists z \leq y)B(x, z)),$$

Demonstração. Provaremos primeiro que $n \in B^* \Rightarrow GR(x)$ é demonstrável; e depois que $n \in A^* \Rightarrow \neg GR(x)$ é demonstrável:

- Suponha $n \in B^*$.
  1. $\vdash_T B(n, k)$, para algum k, pela hipótese de enumerabilidade de B.
  2. $\vdash_T \neg A(n, m)$, para todos m, em especial para todos $m \leq k$, pela hipótese de que A e B são disjuntos e enumeráveis.
  3. $\vdash_T y \leq k \rightarrow \neg A(n, y)$, por $R4$.
  4. $\vdash_T A(n, y) \rightarrow k \leq y$, por $R5$ e lógica (contrapostiva).
  5. $\vdash_T A(n, y) \rightarrow k \leq y \wedge B(n, k)$, por 1.
  6. $\vdash_T \forall y(A(n, y) \rightarrow (\exists x \leq y)B(n, x))$, por lógica (introduzimos o quantificador existencial e depois o universal).
- Suponha $n \in A^*$.

[14] Onde: $x \leq y \leftrightarrow: x < y \vee x = y$.

1. $\vdash_T A(m,k)$, para algum k, pela enumerabilidade de A.
2. $\vdash_T \neg B(n,m)$, para $m \leq k$, pois A e B são disjuntos.
3. $\vdash_T (\forall z \leq k)B(n,y)$, por lógica.
4. $\vdash_T \forall y(A(n,y) \rightarrow (\exists z \leq y)B(n,z))$, por lógica.

□

DEFINIÇÃO **34. Representabilidade de conjuntos e predicados:**
Uma fórmula **F** *representa fracamente* [15] uma relação R[16] na teoria T se para todos $a_1, ..., a_n$ números inteiros positivos:

$$R(a_1, ..., a_n) \Leftrightarrow \vdash_T \mathbf{F}(\overline{\mathbf{a_1}}, ..., \overline{\mathbf{a_n}}).$$

Uma fórmula **F** *representa* na teoria T[17] uma relação R se para todos $a_1, ..., a_n$ números inteiros positivos:

- $R(a_1, ..., a_n) \Rightarrow \vdash_T \mathbf{F}(\overline{\mathbf{a_1}}, ..., \overline{\mathbf{a_n}})$;
- $\neg R(a_1, ..., a_n) \Rightarrow \vdash_T \neg\mathbf{F}(\overline{\mathbf{a_1}}, ..., \overline{\mathbf{a_n}})$.

Observemos que representabilidade é o caso especial de separabilidade quando $R = R$ e $S = \neg R$.

LEMA **4.** *F separa R de $\neg R$ sse F representa R*

DEFINIÇÃO **35. Representabilidade de funções:**
Dadas uma fórmula F com $x_1, ..., x_n$ variáveis distintas na teoria T e uma função total $f : \mathbb{N}^n \rightarrow \mathbb{N}$ tal que $f(a_1, ..., a_n) = b$, dizemos que, para todos $a_1, .., a_n$:

- *F representa fracamente f* se

$$f(a_1, ..., a_n) = b \Leftrightarrow \vdash_T F_{x_1,...,x_n,y}[a_1, ..., a_n, b]$$

- *F representa f* se:
  - $f(a_1, ..., a_n) = b \Rightarrow \vdash_T F_{x_1,...,x_n}[a_1, ..., a_n, b]$
  - $f(a_1, ..., a_n) \neq b \Rightarrow \vdash_T \neg F_{x_1,...,x_n}[a_1, ..., a_n, b]$

[15] Usamos a notação de acordo com [Odi99]. Já [Smu92] e [BJB12]usam a terminologia *representa* para este conceito. Outros usam *numera*.
[16] Podemos interpretar um conjunto como uma relação unária.
[17] Usamos a notação de acordo com [Sho67] e [Odi99] . Já [Smu92] e [BJB12] usam a terminologia *define*. Outros usam a notação *numera dualmente*.

- *F representa fortemente*[18] f se F representa f e além disso:

$$\vdash_T \forall(y)(F_{x_1,...,x_n,y}[a_1,...,a_n,y] \leftrightarrow y = b).$$

Uma função é *representável* (ou definível ou fortemente definível) em T se existe alguma fórmula que o representa em T.

PROPOSIÇÃO **17.** *As seguintes afirmações valem:*

1. *Todo predicado que é representável em T é representável fracamente em T, o contrário não necessariamente vale (a não ser que T seja completa).*
2. *Toda função que é fortemente representável em T é representável, e toda função representável sob a hipótese de T consistente é fracamente representável.*
3. *Se R é representável fracamente em T, então sua função característica κ_R é representável fracamente em T.*
4. *Se R é representável em T, então sua função característica κ_R é representável em T (mas não vale a recíproca).*
5. *R é representável em T sse κ_R é fortemente representável em T.*

Demonstração. Não provaremos os resultados de 1-4.

5) *Ida*: Suponha que R é representável em T pela sentença $F(\vec{a})$.[19] Mostraremos que κ_R é fortemente representável por $G(\vec{a}, y)$ definido da seguinte maneira:

$$(F(\vec{a}) \wedge y = 1) \vee (\neg F(\vec{a}) \wedge y = 0)$$

De fato, se $\kappa(\vec{a}) = 1$ temos que $R(\vec{a})$ vale e então, pela representabilidade, $\vdash_T F(\vec{a})$. E portanto, por lógica,

$$\vdash_T G(\vec{a}, y) \leftrightarrow y = 1.$$

Uma prova similar se segue para $\kappa_R(\vec{a}) = 0$.

Volta: Suponha que κ_R é fortemente representável por $F(\vec{a}, y)$. A seguinte fórmula $G(\vec{a})$ representa $R(\vec{a})$:

$$\vdash_T F_{\vec{x},y}[\vec{a}, 1].$$

Onde substituímos todas as ocorrências de y por 1. Isto nos dá o que queremos: $\vdash_T G(\vec{a})$ porque $\vdash_T F(\vec{a}, y) \leftrightarrow y = 1$.

□

TEOREMA **13.** *Os dois teoremas valem para uma extensão consistente T de* **R***:*

[18] [BJB12] usa a nomenclatura *definível* para esta cláusula (sem assumir as cláusulas anteriores).

[19] Usamos $\vec{a}$ para denotar a uma n-upla do tipo $(a_1, ..., a_n)$.

- ***Teorema da representabilidade.*** *Se um predicado é computável, então é representável em T. Se uma função é computável, então é representável fortemente em T.*

- ***Ehrenfeucht-Feferman.*** *Se um predicado é c.e., então é fracamente representável em T.*

Demonstração. Não realizaremos a prova do teorema de Ehrenfeucht-Feferman.[20]

Para o teorema da representabilidade. Provaremos primeiro a afirmação para um predicado R, depois a afirmação para a função f.

- Se R é computável, então, pelo teorema de Post versão simples (ver proposição 10), tanto R quanto R^c são c.e., logo Σ_1, pelo teorema 12. Logo, pelos lemas 2 e 3, são mutualmente separáveis por uma fórmula F. Mas, por definição (ou o lema seguinte à definição 34), F separa R de R^c em T sse F representa R em T. Portanto, temos que R é representável em T.

- Suponha que f é computável. Primeiro mostraremos que $f(x) = y$ é computável sse $G(x, y)$ é computável, onde G é o gráfico de f. De fato, podemos ver que f pode ser usada para definir a função característica de G, e isto implica a ida, i.e. que G é computável. Conversamente, G é computável sse sua função característica o é, e usamos isto para construir f.

  Como G é computável, pelo item anterior G é representável. Logo, f é fortemente representável pela proposição 17.

□

5.2.4 Ponto-fixo lógico

Podemos usar o teorema do ponto-fixo da computabilidade para produzir fórmulas auto-referenciais. Utilizando esta ideia podemos gerar o equivalente ao um ponto fixo na lógica:

Definição **36.** Dizemos que a sentença X é o *ponto-fixo* da fórmula $F(x)$ sse

$$\vdash_T X \leftrightarrow F(\overline{\ulcorner X \urcorner})$$

Teorema **14.** ***Teorema do ponto-fixo para lógica:*** *Se T é uma teoria em que toda função computável é fortemente representável, então toda fórmula $F(x)$ admite um ponto fixo.* [21]

[20] Em sua prova é necessário desenvolver um argumento de diagonalização similar ao feito para provar o teorema de Gödel-Rosser, ver [Smu92], p. 87

[21] Podemos enfraquecer esta hipótese, basta ter uma teoria que possui uma função fortemente representável que calcula o número de Gödel de uma expressão qualquer. Smullyan na página 103 de [Smu92] enfraquece ainda mais e requer apenas a condição de aceitabilidade.

Demonstração. Dado uma F seja f uma função computável tal que:

$$\phi_{f(n)}(x) = \ulcorner F(\phi_n(x)) \urcorner,$$

Notemos que $\phi_{f(n)}$ é uma função computável, pois não calcula de fato ϕ_n, apenas calcula o número de Gödel da expressão a direita, e isto pode ser feito de forma computável.[22]

Logo, pelo teorema 9 do ponto-fixo da computabilidade:

$$\phi_n(n) = \phi_{f(n)}(n) = \ulcorner F(\phi_n(n)) \urcorner$$

Pela hipótese de que $\phi_n(x)$ é computável, temos que é representável fortemente por uma $G(n, y)$ Mostraremos que $X := \exists y(F(y) \wedge G(n, y))$ é o ponto fixo de F.

De fato, como ϕ_n é fortemente representável:

$$\vdash_T \forall y(G(n, y) \leftrightarrow y = \ulcorner X \urcorner)$$

Mas, por lógica, podemos trabalhar esta expressão, adicionando a fórmula $F(y)$ dos dois lados da implicação e substituir o quantificador universal pelo existencial. Logo, chegamos a:

$$\vdash_T \exists y(G(n, y) \wedge F(y)) \leftrightarrow \exists y(y = \ulcorner X \urcorner \wedge F(y))$$

E isto implica logicamente:

$$\vdash_T \exists y(G(n, y) \wedge F(y)) \leftrightarrow F(\ulcorner X \urcorner)$$

Portanto, pela definição de X:

$$\vdash_T X \leftrightarrow F(\ulcorner X \urcorner)$$

□

Este teorema consegue gerar uma sentença X que diz de si mesma que ela tem a propriedade F. Ou, ao menos, podemos interpretá-lo intuitivamente assim.

5.2.5 Incompletudes

PROPOSIÇÃO **18.** ***Propriedades do predicado de provabilidade*** *As seguintes afirmações são verdadeiras para T uma teoria extensão consistente axiomatizável de* $\mathbf{R}$*:*

1. $\vdash_T X \Rightarrow \vdash_T P(\ulcorner X \urcorner)$.
2. *Se T for ω-consistente*[23]*, então* $\vdash_T X \Leftrightarrow \vdash_T P(\ulcorner X \urcorner)$.

[22] Note que ϕ_n é na realidade a expressão da máquina de Turing de índice n construído na linguagem da aritmética

[23] I.e. sempre que $\vdash_T \exists x F(x)$, então existe n tal que $\nvdash \neg F(n)$.

3. $\vdash_T P(\ulcorner X \to Y \urcorner) \to (P(\ulcorner X \urcorner) \to P(\ulcorner Y \urcorner))$.

4. $\vdash_T P(\ulcorner X \urcorner) \to P(\ulcorner P(\ulcorner X \urcorner) \urcorner)$.

5. $\vdash_T X \to Y \Rightarrow \vdash_T P(\ulcorner X \urcorner) \to P(\ulcorner Y \urcorner))$.

6. $\vdash_T X \to (Y \to Z) \Rightarrow \vdash_T P(\ulcorner X \urcorner) \to (P(\ulcorner Y \urcorner) \to P(\ulcorner Z \urcorner))$.

7. $\vdash_T X \to (P(\ulcorner X \urcorner) \to Y) \Rightarrow \vdash_T P(\ulcorner X \urcorner) \to P(\ulcorner Y \urcorner)$.

Demonstração. Provaremos as afirmações em ordem:

1. Suponha $\vdash_T X$. Então $\mathcal{N} \models P(\ulcorner X \urcorner)$. Logo, pelo corolário da Σ_1 completude do lema 16 $\vdash_T P(\ulcorner X \urcorner)$.

2. De fato, usando o mesmo truque acima da Σ_1-completude, para todos x, y:

$$Pr(x, y) \Rightarrow \vdash_T Pr(x, y)$$
$$\neg Pr(x, y) \Rightarrow \vdash_T \neg Pr(x, y)$$

   Mostraremos que P é fracamente representável por $\exists y Pr(x, y)$.

   Por um lado assuma $P(x)$, temos rapidamente, pela Σ_1-completude, para algum n: $P(n) \Rightarrow \vdash_T P(n)$.

   Por outro lado, assuma $\vdash_T \exists y Pr(x, y)$. Logo por ω-consistência, para algum n, $\nvdash_T \neg Pr(x, n)$. Então, para algum n, $\neg Pr(x, n)$ não pode valer, caso contrário teríamos $\vdash_T \neg Pr(x, n)$. Logo, $Pr(x, n)$ é o caso, para algum n. E a partir daí, $P(x)$ vale também. O que conclui a afirmação.

3. A afirmação é verdadeira nos modelo standard pois é uma afirmação do modus ponens para o conteúdo de P. A partir de sua verdade podemos fazer uma argumentação parecida com a anterior, como ela é Σ_1, temos que é demonstrável.

4. Assumindo $\vdash_T P(\ulcorner X \urcorner)$ chegamos em $\vdash_T P(\ulcorner P(\ulcorner X \urcorner) \urcorner)$ por 1. Logo, temos o resultado por lógica. Também assumindo o contrário, temos o resultado por lógica.

5. Temos o seguinte raciocínio a partir dos itens anteriores:

   $\vdash_T X \to Y$

   $\vdash_T P(\ulcorner X \to Y \urcorner)$, por 1.

   $\vdash_T P(\ulcorner X \urcorner) \to P(\ulcorner Y \urcorner)$, por 3.

6. Óbvio aplicando o item anterior e o item 3.

7. $\vdash_T X \to (P(\ulcorner X \urcorner) \to Y)$

   $\vdash_T P(\ulcorner X \urcorner) \to (P(\ulcorner P(\ulcorner X \urcorner) \urcorner) \to P(\ulcorner Y \urcorner)$, por 6.

$\vdash_T P(\ulcorner X \urcorner) \to P(\ulcorner Y \urcorner)$, por 4 e lógica.

□

Com isto temos uma bateria de alguns resultados de limitação de teorias:

TEOREMA **15.** *Seja T uma extensão consistente axiomatizável de* $\mathbf{R}$*. Então*

1. ***Teorema de Tarski*** *(i) o predicado $T(x)$ que afirma $\mathcal{N} \models X \leftrightarrow T(\overline{\ulcorner X \urcorner})$, para todo X, não existe em T. (ii) Além disso, não existe um $T(x)$ tal que $\vdash_T X \leftrightarrow T(\overline{\ulcorner X \urcorner})$, para todo X.*

2. ***Teorema de Church:*** *O conjunto das sentenças decidíveis de T não é computável (i.e. T é indecidível). (E além disso, temos uma versão abstrata da incompletude).*

3. ***Teorema de Gödel:*** *Suponha que T é ω-consistente. Então, todo ponto fixo da fórmula $\neg P(x)$ é indecidível em T.*

4. ***Teorema de Gödel-Rosser versão do ponto-fixo:*** *Qualquer ponto fixo de $GR'(x) \leftrightarrow: \forall y(Rf(x,y) \to (\exists z \leq y)Pr(x,z))$ é indecidível em T.*

5. ***Segundo Teorema da Incompletude:*** *$\nvdash_T consis$, onde $consis$ é a fórmula definida como $\neg P(\bot)$.*[24]

6. ***Teoema de Löb:*** *$\vdash_T P(\ulcorner Y \urcorner) \to Y \Rightarrow \vdash_T Y$.*

Demonstração. Provaremos em ordem:

1. (i) Suponha o contrário. Logo, existe um ponto-fixo para $\neg T$:

   $$\vdash_T X \leftrightarrow \neg T(\overline{\ulcorner X \urcorner}).$$

   Logo, $\vdash_T \neg X \leftrightarrow T(\overline{\ulcorner X \urcorner})$. Pela hipótese e pelo fato de que $T \subseteq Teo(\mathcal{N})$, $\mathcal{N} \models X \leftrightarrow \neg X$, contradição.

   (ii) Suponha o contrário. Usamos o mesmo ponto-fixo anterior para obter uma contradição.

2. Suponha o contrário, então P^* e R^* são computáveis. Seja M o número de Gödel de uma sentença que afirma que a máquina de Turing T pára com o input m. Logo, $M \in P^*$ ou $M \in R^*$ (ou exclusivo, pois T é consistente). Mas então resolvemos o problema da parada 8. Contradição.

   (Além disso, conseguimos a incompletude desta prova porque incomputabilidade das sentenças decidíveis implica incompletude. A argumentação intuitiva deste fato é pela contrapositiva: se tivéssemos uma teoria completa, então bastaríamos deixar uma máquina de Turing testando a provabilidade e refutabilidade de uma lista de todas fórmulas possíveis. E isso implica que o conjunto das formulas decidíveis é computável.)

[24] Podemos deixar: $\bot \leftrightarrow: 0 \neq 0$.

3. Existe um ponto-fixo G para a fórmula $\neg P(x)$. Logo:

$$\vdash_T G \leftrightarrow \neg P(\ulcorner G \urcorner)$$

Assim, $\nvdash_T G$. Pois caso contrário $\vdash_T G$ e $\vdash_T \neg P(\ulcorner G \urcorner)$, por modus ponens. Mas também de $\vdash_T G$, pela propriedade do predicado de provabilidade 18, $\vdash_T P(\ulcorner G \urcorner)$. Contradição.

E $\nvdash_T \neg G$. Pois caso contrário $\vdash_T \neg G$ e $\vdash_T P(\ulcorner G \urcorner)$, por modus ponens. Pelas propriedades do predicado de provabilidade item 2, sob a condição de ω-consistência, $\vdash_T G$. Contradição.

4. Provaremos um *lema para este item*: Se $F(x)$ separa o conjunto [25] A de B em uma teoria consistente T, então $F(x)$ representa fracamente um superconjunto $\hat{A} \supseteq A$ disjunto de B.

Suponha que $\hat{A}$ seja algum predicado representável fracamente por $F(x)$. Pela assunção de separabilidade de A e definição de representabilidade fraca de $\hat{A}$, temos que, para todo $n \in \mathbb{N}$:

$$n \in A \Rightarrow \vdash_T F(n) \Leftrightarrow n \in \hat{A}$$

Logo, $A \subseteq \hat{A}$. Por outro lado, se $F(x)$ separa A de B, para todo $n \in \mathbb{N}$:

$$n \in B \Rightarrow \vdash_T \neg F(n)$$

Suponha que:

$$n \in A \cap B \Rightarrow \vdash_T F(n) \text{ e } \vdash_T \neg F(n)$$

Mas como T é consistente a implicação acima não pode ocorrer. Logo, $A \cap B = \emptyset$ e, portanto, como $A \subseteq \hat{A}$:

$$\hat{A} \cap B = \emptyset.$$

Isto termina a demonstração do lema

Defina:

$$GR'(x) := \forall y(Prov(x, y) \rightarrow (\exists z \leq y) Ref(\ulcorner x \urcorner, z))$$

Existe um ponto-fixo G para a fórmula GR'. Logo:

$$\vdash_T G \leftrightarrow GR'(\ulcorner G \urcorner)$$

Pelo lema da separação 3 e lema acima GR' separa o conjunto R de P e ademais representa fracamente $\hat{R'} \supseteq R$ disjunto de P.

[25] Ou predicado, a prova é a mesma.

- $\nvdash_T G$. Pois se $\vdash_T G$, então $\vdash_T GR'(\ulcorner G \urcorner)$ implica $\hat{R}'(\ulcorner G \urcorner)$, pela representabilidade fraca. E $\vdash_T G$ implica $P(\ulcorner G \urcorner)$, pela definição de P. Contradição, pois P e $\hat{R}'$ são disjuntos.
- $\nvdash_T \neg G$. Pois se $\vdash_T \neg G$, então $\vdash_T \neg GR(\ulcorner G \urcorner)$. Ao mesmo tempo:

$$\vdash_T \neg G \Rightarrow R(\ulcorner G \urcorner) \Rightarrow \hat{R}'(\ulcorner G \urcorner) \Leftrightarrow \vdash_T GR(\ulcorner G \urcorner),$$

  a bimplicação pela representabilidade fraca. Contradição.

5. Existe um ponto-fixo para $\neg P(x)$:

$$\vdash_T G \leftrightarrow \neg P(\ulcorner G \urcorner).$$

   Logo, pela primeira parte da prova do teorema da incompletude:

$$\nvdash_T G.$$

   Por outro lado a equação do ponto-fixo é equivalente a:

$$\vdash_T G \leftrightarrow (P(\ulcorner G \urcorner) \rightarrow \bot)$$

   E pela propriedade 6 do operador de provabilidade isso é equivalente a: $P(\ulcorner G \urcorner) \rightarrow P(\bot)$. Que por sua vez pode ser escrita usando a definição de *consis* e a contrapositiva:

$$\vdash_T consis \rightarrow \neg P(\ulcorner G \urcorner)$$

   Portanto, se *consis* fosse demonstrável, também o seria $\neg P(\ulcorner G \urcorner)$. E pela equação do ponto fixo novamente teríamos: $\vdash_T G$. Contradição com a consistência de T pois tínhamos visto que $\nvdash_T G$.

6. Existe um ponto-fixo para $P(x) \rightarrow Y$:

$$\vdash_T X \leftrightarrow (P(\ulcorner X \urcorner) \rightarrow Y)$$

   Logo, pela propriedade 6 do predicado de provabilidade:$\vdash_T P(\ulcorner X \urcorner) \rightarrow P(\ulcorner Y \urcorner)$. E pela hipótese:

$$\vdash_T P(\ulcorner X \urcorner) \rightarrow Y$$

   Novamente pelo ponto-fixo tiramos que $\vdash_T X$ e $\vdash_T P(\ulcorner X \urcorner)$ pela propriedade 1 do predicado de provabilidade. Logo $\vdash_T Y$ da equação acima.

□

Capítulo 6

Incompletude de Chaitin

> Neste capítulo discutiremos algumas questões sobre complexidade de Kolmogorov. Esta definição surgirá naturalmente de questões relativas à computabilidade e pode ser usada para provar uma versão do teorema da incompletude que não utiliza necessariamente as técnicas do ponto-fixo que utilizamos no capítulo passado, nem auto-referência. Na primeira seção definiremos complexidade. Na segunda seção provaremos a incompletude, primeiramente sem usar o teorema do ponto fixo, e em seguida o utilizando. Na terceira seção discutiremos algumas interpretações sobre este resultado.

Façamos um exercício mental de examinar os seguintes strings e classificá-los por ordem de complexidade (entre parênteses f_1 diz a quantidade de uns):

a 0000000000.0000000000.0000000000.0000000000.0000000000 ($f_1 = 0$)

b 0101010101.0101010101.0101010101.0101010101.0101010101 ($f_1 = 25$)

c 1011010001.1011010001.0000110000.1000101101.1000101101 ($f_1 = 22$)

d 0100000000.0000011111.0000000000.0000101001.1000000001 ($f_1 = 10$)

e 0110111001.0111011110.0010011010.1011110011.0111101111 ($f_1 = 36$)

f 0010010010.1111010011.1111011100.0000010111.1101001010 ($f_1 = 26$)

Apenas olhando para estes strings é possível ver que eles possuem estruturas internas diferentes. De fato, eles foram gerados de formas bem diferentes:

a é um string que foi gerado com a ideia de apenas ter zeros;

b foi gerado através do algoritmo de intercalar zeros e uns;

c é um string cujas primeiras 10 letras foram criadas por um gerador de números aleatórios, em seguida repetimos mais uma vez estas letras, adicionamos quatro 0s e um 1, e depois repetimos o string de trás para frente;

d é a meteorologia dos últimas 50 dias, 1 significa que houve chuva, 0 sol;

e é a versão binária do número de Champernowne, que é formado pela concatenação de todos os números naturais representados em base binária;

f foi gerada através do lançamento de cara e coroa.

Notemos que no sentido intuitivo de complexidade[1], sem saber como estas sentenças foram geradas, diríamos que o string a e o string b são os menos complexos. Com algum tempo e disposição é possível descobrir o padrão subjacente em c. Quanto ao string d é possível ver algum padrão de repetição, mas, ainda assim, parece mais complexo que a e b. Os strings d e e, parecem aleatórios. A única forma de dizê-los para uma outra pessoa, que precisa anotá-los com precisão, é repetindo todo o string, do começo ao fim.

Saber a frequência da ocorrência de uns (f_1) e zeros (f_0) não nos ajuda muito na descoberta dos padrões. Pois o string b e o string c possuem praticamente os mesmos números de uns, mas diríamos que b é menos complexo que c.

Se tivéssemos um computador com determinadas funções pré-programadas, poderíamos fazer uma associação entre complexidade da sentença e o tamanho dos comandos que teríamos que digitar para gerar os strings em questão. Por exemplo, com os comandos seguintes programados: $n\times$ para indicar para repetir o string n vezes, a função "reflection" para indicar que é para refletir o string, e assim por diante, poderíamos medir ($|\cdot|$) o tamanho do código que usamos para gerar estes strings:

a 0000000000.0000000000.0000000000.0000000000.0000000000

- $|50 \times (0)| = 6$

b 0101010101.0101010101.0101010101.0101010101.0101010101

- $|25 \times (01)| = 7$

c 1011010001.1011010001.0000110000.1000101101.1000101101

- $|\text{reflection}(2 \times (1011010010)(00001))| = 32$

d 0100000000.0000011111.0000000000.0000101001.1000000001

- $|(01)13 \times (0)5 \times (1)14 \times (0)(10)2 \times (1)8 \times (0)(1)| = 38$

e 0110111001.0111011110.0010011010.1011110011.0111101111

- $|\text{Champernowne.binary.expansion}| = 41$

f 0010010010.1111010011.1111011100.0000010111.1101001010

- $|(0010010010.1111010011.1111011100.0000010111.1101001010)| = 52$

[1] No sentido de acharmos um string aparentemente mais fácil ou difícil de descrever.

Vemos de forma mais ou menos intuitiva para onde este exemplo nos conduz: a *complexidade de um string* é o tamanho do menor programa de uma linguagem utilizada para gerar este string.

Mas, a primeira pergunta que surge é: o comprimento do programa não depende da linguagem que se está usando? Façamos uma analogia. Esperamos de determinados especialistas, por exemplo um médico ou um engenheiro, que eles dominem vocabulário básico do assunto em que eles trabalham, pois assim eles podem ser mais eficientes no manejo de algum problema, ou mesmo podem conversar entre si de forma mais precisa. O mesmo se pode dizer de determinadas áreas de linguagens formais. Determinadas linguagens de programação são mais eficientes que outras para lidar com determinados problemas. Por exemplo, Assembly é uma linguagem destinada ao trato de problemas que dependem da linguagem arquitetural do computador, Python por outro lado não coloca este assunto como uma de suas prioridades, porém será muito mais eficiente que Assembly em outros assuntos, por exemplo para tratar de strings.

Esta objeção é válida. Foi apenas através do desenvolvimento da teoria da computabilidade que tais considerações relativas à linguagem foram eliminadas. Isto porque, como vimos, podemos construir uma função computável que emula qualquer outra máquina de Turing específica: a máquina de Turing Universal. Através dela é possível argumentar que é possível construir máquinas de Turing Universais que têm uma característica que nos permite eliminar esse problema, chamada de aditividade ótima. Os exemplos de linguagem de programação a que estamos acostumados, Assembly, Python e mesmo as máquinas de Turing, todas podem ter programas universais com esta propriedade. Esta propriedade, dito de modo discursivo, é: podemos dividir as tarefas de programas maiores em subprogramas. A essência da ideia é que há uma convergência na tarefa do programador. Todas estas linguagens apenas diferem na forma como descrevem ao computador uma função computável que o hardware acessará, e a diferença destas descrições é dá ordem de uma constante que depende apenas de peculiaridades da linguagem. O trabalho bruto mínimo realizado pelo computador por outro lado é o que realmente irá definir o trabalho computacional, e este não se deixará afetar pelo modo como é descrito. Esta ideia é um dos teoremas deste campo de estudo, chamado de teorema da invariância, que diz que as medidas de complexidade definidas sobre estas máquinas universais guardam a otimização aditiva. E será um dos assuntos a serem abordados neste capítulo.

Em termos práticos, podemos associar o código de um programa que gera um determinado string ao resultado da compressão deste string. Este é um resultado da teoria: podemos associar complexidade a métodos estatísticos de compressão (não trataremos sobre isso). Isto ajuda a entender o que está em jogo. Esta compressão pode ser realizada por exemplo por ferramentas de compressão zip.[2]

Utilizando a biblioteca de compressão zlib[3] podemos comparar duas compressões:

[2]Os métodos mais utilizado atualmente são os métodos de compressão estilo Huffmann.

[3]Biblioteca de compressão desenvolvida por Jean-loup Gailly e Mark Adler: <https://zlib.net/>. Utilizamos a versão disponível da biblioteca para Python.

- **String x**: 0101010101010101010101010101010101010101010101010101010101
  0101010101010101010101010101010101010101010101010101010101010101010101
  0101010101010101010101010101010101010101010101010101010101010101010101
  0101010101010101010101010101010101010101010101010101010101010101010101
  0101010101010101010101010101010101010101010101010101010101010101010101
  0101010101010101010101010101010101010101010101010101010101010101010101
  0101010101010101010101010101010101010101010101010101010101010101010101
  0101010101010101010101010101

- Comprimento do string : 400

- String comprimido: 0111100011011010000010110000110100100101000101
  0100000000000000001010011110011000000100001001101l

- Tamanho do string comprimido: 96

- Razão tamanho string comprimido/original : 0.24

Comparemos à compressão do string y gerado aleatoriamente:

- **String y**: 1011111001011011110010100010001101001111011010110011001
  0011101101111001111111110100000100000000111100000010000001110010
  1000101110010110010110111001001011100100110110011001011001010101
  0100110011111110010101101101101100101111011111101111100100110101
  0
  1001111110101110011001001010001000110100001100110010010011100010
  0010110010101010100100101011011111000100100010010000011110010000
  0000010001000001011101111

- Comprimento do string: 400

- String comprimido: 0111100011011010000000010011001000000000110011
  0111111111101111100101101111001010001000110100111101101011001100
  1001110110111100111111111010000010000000011110000001000000111001O
  1000101110010110010110111001001011100100110110011001011001010101
  0100110011111110010101101101101100101111011111101111100100110101
  1001111110101110011001001010001000110100001100110010010011100010
  0010110010101010100100101011011111000100100010010000011110010000
  0000010001000001011110111101110111011100100011000100100l0

- Tamanho do string comprimido: 488

- Razão tamanho string comprimido/original : 1.22

O tamanho da compressão do string x é menor que o tamanho da compressão de y, embora x e y ambos tenham o mesmo tamanho. Intuitivamente isto mede a aleatoriedade dos strings dados. O string y é mais aleatório (de fato, foi gerado por um pseudo-gerador de números aleatórios), já o x não. O string x possui mais regularidades, por esta razão pôde ser melhor comprimido, já o y teve uma compressão maior que o string original. Quando a compressão é maior que o string original, poderíamos por definição, ao invés de forçar a compressão,

estabelecer o próprio string como sua própria descrição. Este seria a menor forma de descrever o string. Logo, diríamos que a compressão de y é o próprio y, caso no qual a complexidade do string y ficaria por volta de 400.

O intuito deste capítulo é fazer a ligação entre a noção informal de complexidade surgida desta considerações intuitivas e as definições que fornecemos de funções computáveis dos capítulos anteriores, fornecendo assim uma nova consideração sobre fenômenos incomputáveis. A definição formal se baseia na possibilidade de basearmos a formalização de complexidade do ponto de vista universal. Porém, isto também trará consigo um ponto negativo. Como veremos, a função complexidade não é computável, embora isto não seja exatamente um problema para estabelecer aproximações para a complexidades de strings.

6.1 Definição de complexidade

6.1.1 Complexidade relativa

Para começarmos a falar de complexidade, vamos definir o que significa complexidade com relação a uma função parcial qualquer, **não** necessariamente computável. A ideia é que a complexidade com relação a esta função parcial será o menor tamanho de input que a função recebe para calcular um determinado número, se este input existir.

Definição **37. Complexidade relativa a uma função parcial** Dados p, x números naturais, definimos a *complexidade relativa a função parcial* ϕ, $C_\phi : \mathbb{N} \to \mathbb{N}$, como

$$C_\phi(x) = \min\{|p| : \phi(p) = x\}$$

onde min é a função que dado um conjunto calcula o menor elemento deste conjunto. Se não encontrarmos nenhum, $C_\phi(x) \uparrow$.

Em primeiro lugar, antes de nos perguntarmos sobre como é possível que esta definição possa realmente ser usada como medida universal, vamos deixar claro um raciocínio que é muito usado quando falamos de complexidade relativa a uma função parcial. Suponha que temos uma função parcial ϕ e por algum método conseguimos descobrir que $\phi(p) = x$. É claro que este talvez não seja o menor p que nos descreve x. Mas por outro lado, qualquer outro candidato q tal que $\phi(q) = x$, tem que ter tamanho menor ou igual a p. Isto é $C_\phi(x) \leq |p|$. Isto é muito usado em provas relacionadas a complexidade, pois, como veremos, muitos raciocínios se resumirão em definir um programa e, depois, usar propriedades universais do teorema da invariância que discutiremos a seguir. Vamos colocar isto em destaque na próxima proposição:

Proposição **19.** *Suponha que para algum p e x, $\phi(p) = x$. Logo, $C_\phi(x) \leq |p|$.*

Em segundo lugar, para discutirmos o teorema da invariância, observemos que C_ϕ é uma funções nos naturais. Mas demos, no capítulo sobre codificações,

definição 12 uma bijeção entre os naturais e os strings binários. E além disso, se observarmos que tamanho é preservado a menos de uma constante entre estes dois conjuntos, podemos pensar em x e p como se fossem strings binários, como já observamos. De igual maneira podemos pensar em $C_\phi : \mathcal{B} \to \mathcal{B}$.

Claro, ϕ poderia ser parcial computável. E é justamente quando pensamos assim que propriedades universais interessantes de invariância começam a surgir. Pensando em funções parciais computáveis é possível ter uma ideia intuitiva do que está acontecendo. Podemos pensar genericamente nos inputs como subprogramas pelo teorema do parâmetro 7. O que estamos fazendo é rodar todos os subprogramas de um programa até achar o menor que calcula um determinado número.

A função p.c. ϕ poderia ser inclusive universal. Obviamente, é o que queremos. Não seria nada interessante nos basear na complexidade de programas que nos retornam apenas 1, não conseguiríamos computar a complexidade de nenhum número maior que 1. Também não seria interessante um programa que não tem uma variedade interessante de subprogramas, por isso as funções universais aparecem como naturais. Porém, mesmo máquinas universais precisam ser escolhidas com cuidado. Queremos as máquinas universais que possuem a seguinte propriedade.

Definição **38. Função ad.ot:** Uma função ϕ p.c. é aditivamente ótima se para toda ψ existe uma $c_{\phi,\psi}$ tal que $C_\phi(x) \leq C_\psi(x) + c_{\phi,\psi}$.

Notemos que $c_{\phi,\psi}$ é uma constante que depende destas duas funções. Provaremos que é possível escolher funções universais que possuem estas propriedade.

Teorema **16.** ***Teorema da invariância:*** *Existe uma função universal parcial computável aditivamente ótima. Isto é existe uma função universal ϕ tal que para toda ϕ' temos:*

$$C_\phi(x) \leq C_{\phi'}(x) + c_{\phi,\phi'}$$

Demonstração. Seja c uma função codificadora computável injetora tal que $c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ com inversa também computável. A escolha dela é arbitrária, a função de pareamento standard é suficiente (ver definição 18). Logo, existe uma função p.c. ϕ que recebe um número do tipo $c(p, q)$ e roda um subprograma para decodificar p de q, e após dá como resultado:

$$\phi(c(p, q)) = \phi_p(q).$$

Isto é possível pelo teorema da enumeração 6. Podemos também escolher ϕ tal que o número de ϕ nesta enumeração seja ϕ_0. Assim, $\phi(c(0, q)) = \phi_0(q) = \phi(q)$. Note que ϕ faz o mesmo trabalho de uma função universal, porém é unária.

Mas, $|c(p, q)| > |q|$, pois p pode crescer indefinidamente, logo existe uma constante dependendo de p tal que $|c(p, q)| \leq |q| + c_p$. Com a finalidade de produzir x, as funções ϕ e ϕ_p acharão o menor programa q, a menos de c_p. Na realidade ϕ pode ser mais eficiente que ϕ_p, pois, por conta de sua universalidade

e a definição $\phi = \phi_0$, ela pode achar um outro programa que faz o trabalho melhor que ϕ_p. Logo:

$$C_\phi(x) \leq C_{\phi_p}(x) + c_p$$

Como p é suficiente para darmos uma descrição de ϕ_p, então: $C_\phi(x) \leq C_{\phi_p}(x) + c_{\phi,\phi_p}$. Logo ϕ é aditivamente ótima e também universal.

□

Porém para termos alguma ideia sobre o tamanho das constantes escolheremos uma função codificação que nos ajude. Seja uma $c : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$ definida da seguinte forma, para p e q strings binários:

$$c(p,q) = \overbrace{11...1}^{|p| \text{ vezes}}\ 0pq$$

Esta função é computável com inversa computável. Pois podemos criar um programa com um contador i que conta o tamanho de x e coloca os 1s correspondentes na frente do string; e inversamente conta o número de 1s e separa x de y.[4] Além disso é injetora, pois possui inversa.

Também $|c(p,q)| \leq |q| + 2|p| + 1$, pois por inspeção, para determinar o tamanho de $c(p,q)$ contamos o números de 1s e zeros na frente de pq , mais o tamanho de p, e mais o tamanho de q. Logo, podemos fazer $c_p = 2|p| + 1$. E portanto, pela proposição sob esta codificação, e mantendo em mente o isomorfismo entre strings binários e naturais pela função tamanho:

$$C_\phi(x) \leq C_{\phi_p}(x) + 2|p| + 1$$

Corolário **10.** *A constante c_p na proposição acima pode ser limitada por, pelo menos, $O(1) = 2|p| + 1$.*[5]

Esta constante pode ser chamada de *custo de simulação* de um programa por outro. Se tivermos duas funções uni.ad.ot., ψ e τ, pelo teorema da invariância temos que a diferença entre estes dois programas tem que ser limitado por um custo de simulação que depende tanto de ψ quanto de τ. Logo, fazendo $\psi = \phi$ e $\tau = \phi'$ temos $C_\psi(x) \leq C_\tau(x) + c_\tau$. E por outro lado substituindo $\tau = \phi$ e $\psi = \phi'$, temos $C_\tau(x) \leq C_\psi(x) + c_\psi$. Logo temos o seguinte resultado:

Corolário **11.** *Sejam ψ e τ duas funções parciais aditivamente ótimas, então:*

$$|C_\psi(x) - C_\tau(x)| \leq c_{\psi,\tau},$$

onde $c_{\psi,\tau} = O(1)$.

Este resultado expressa uma propriedade essencial para a teoria da computação. Linguagens de programação como Assembly e C (ou quaisquer outras, mesmo programas na linguagem de máquinas de Turing) possuem programas

[4] Ver prova do teorema da universalidade.
[5] Para mais informação sobre a função O ver apêndice, seção 8.1.

que são aditivamente ótimos. Podemos chamá-los de intérpretes. Não é novidade que Assembly possua um programa interprete que consiga simular qualquer programa da linguagem C. Se não fosse assdm, um programa em C não conseguiria dialogar com o hardware específico de cada máquina e fazer o computador funcionar. Apesar de ser um processo complicado, o custo de simulação entre estas duas linguagens é da ordem de uma constante, pois em última análise, depende apenas de utilizar códigos para separar subprogramas de programas maiores. O sentido contrário também vale, podemos construir um programa interprete de Assembly em C.

Que todas estas interpretações variem da ordem de uma constante e não dependam do input que estamos trabalhando é essencial para nós, humanos. Nós não temos a capacidade algorítmica nem de memória para conversar com cada hardware específico. E na realidade, queremos programas com sintaxes (ou mesmo gráficos e toda comodidade computacional que estamos familiarizados) intuitivas e abstratas o bastante para não nos preocuparmos com trabalhos desnecessários de dialogar com as máquinas em suas peculiaridades. O que este teorema diz é que estes gastos de simulação não afetarão de forma decisiva o trabalho bruto a ser realizado pela computador, pois serão da ordem de uma constante a depender das linguagens de programação. A não ser que, é claro, se trate de um computador muito lento e linguagens muito pesadas, ou problemas que precisem ser completamente otimizados. Logo, é possível concluir que estamos livres para utilizar linguagens de programação cada vez mais abstratas.

6.1.2 Complexidade absoluta

Tendo em mente este resultado, agora podemos definir precisamente o que é complexidade do ponto de vista destes programas universais ad.ot. Ganhando assim esta invariância com respeito a qualquer complexidade relativa. E, mais genericamente, ganhando universalidade com respeito a qualquer linguagem que tenha o poder de descrever programas universais: funções recursivas parciais, lambda calculus, máquinas de Turing, etc...

Definição **39. Complexidade absoluta de um string:** (ou complexidade de informação algorítmica) (diremos apenas complexidade[6]) Dados p e x números naturais, fixamos $\upsilon = \upsilon_0$ como uma função p.c. universal aditivamente ótima como padrão chamada função *descrição* para o restante do texto (e Uni, sua máquina de Turing correspondente); e também fixamos uma codificação $\langle\cdot\rangle$ pelo menos ou mais eficiente que a que descrevemos no corolário acima, i.e. $|\langle p, q\rangle| \leq |q| + 2|p| + 1$, e que é usada pela descrição. Definimos a *complexidade*, $C : \mathbb{N} \to \mathbb{N}$, de x :

$$C(x) = \min\{|p| : \upsilon(\langle p, 0\rangle)) = x\},$$

[6]Há outros tipos de complexidades chamadas complexidade temporal, ou complexidade computacional, que têm mais a ver com a quantidade de passos necessários para um programa computar algo. Como não estamos interessado primariamente nestas, podemos usar a notação complexidade sem perigo de ambiguidade, reservando o nome complexidade computacional se quisermos falar deste outro tipo de complexidade.

dispensaremos os $\langle\cdot\rangle$.[7]

Colocamos zero como input de $\upsilon(p,0) = \upsilon_p(0)$, pois queremos salientar que a máquina de Turing equivalente recebe o string λ. Ou seja, υ_p tem que calcular x sem ajuda de nenhum tipo de programa como input auxiliar. E por esta razão é chamada de complexidade *absoluta*, em contraposição à complexidade condicional, que depende de outro input para calcular a complexidade de um número.

Definição **40. Complexidade condicional:** Nas mesmas condições da definição acima, para todo y natural defina: $C(x|y) = \min\{|p| : \upsilon(\langle p, y\rangle)) = x\}$.

Em outras palavras: $C(x) = C(x|0)$. Também definimos outro conceito auxiliar, o menor programa que calcula um string:

Definição **41. Programa minimal:** Definimos o *menor programa* que calcula um número x, nas mesmas condições acima: $x^* = \min\{p : \upsilon(\langle p, 0\rangle)) = x\}$.

Um ponto interessante a se observar é que temos o seguinte resultado das definições:

$$C(x) = |x^*|$$

Além disso, como x^* é de fato um programa para x:

$$\upsilon(x^*, 0) = x$$

Proposição **20.** *Para cada x , a medida de complexidade C é limitada superiormente por:*

$$C(x) \leq |x| + c$$

Demonstração. Se o tamanho do menor programa para x exceder $|x|$, simplesmente tome x como o menor programa. Isto é faça x^* o programa: " imprima x ". Haverá alguns custos c para imprimir x, a depender da linguagem trabalhada. Logo, temos $C(x) = |x^*| \leq |x| + c$.

□

Corolário **12.** *Ademais, temos pelo menos:*

$$C(x) \leq \log(x) + c$$

Demonstração. Lembramos que para o programa x^*, x nada mais é que sua representação lexicográfica binária como um string binário elemento de $\mathcal{B}$: i.e. $bin(x)$. Mas pela proposição 2 $|bin(x)| = \log x + O(1)$ e, como c também possui a ordem de uma constante, usamos a proposição acima para obter: $C(x) \leq \log(x) + c$. (Um detalhe, o programa poderia receber x sob outra representação

[7]Notemos que dispensamos os subscritos para denotar a diferença entre complexidade relativa e absoluta.

que não a binária, por exemplo como uma sequência de uns, mas então não seria um programa mais eficiente que este. Aqui vemos os frutos de termos tratado de forma binária a representação que demos no capítulo de codificações). □

Podemos usar esta ideia como pedra de toque para outros resultados.

PROPOSIÇÃO **21.** *A função parcial C é ilimitada.*

Demonstração. De fato, suponha o contrário. Logo, para todo x existe um y tal que $C(x) \leq C(y) = n$. Existem apenas finitos programas de tamanho n ou menor. E, logo, existem um número finito de programas mínimos de tamanho n ou menor. Porém existem infinitos números x. Logo existe um x para o qual não existe nenhum programa mínimo x^*. Contradição, pois como vimos, em último caso, podemos fazer x^*: " imprima x ".

□

É possível definir melhor os limites inferiores para a complexidade, e ver que uma espécie de função piso acompanha bem o valor da função. Ou seja, a função complexidade é bem aproximável pela forma da função $\log_2(x)$. Apesar de todas estas facilidades analíticas da função complexidade, o ponto crucial e desafiador é que ela não é computável. A prova deste resultado é semelhante ao resultado de incompletude que veremos na próxima seção, está baseado no paradoxo de Berry:

> O_1 menor_2 número_3 que_4 não_5 pode_6 ser_7 definido_8 com_9 menos_{10} de_{11} vinte_{12} palavras_{13}

Mas acabamos de definir este número com treze palavras.

PROPOSIÇÃO **22.** *C não é computável.*

Demonstração. Suponha por absurdo que seja computável, e seja $T_C(x)$ a máquina que a computa a função C no string x. Logo, podemos usar T_C como subprograma de um outra máquina T_{berry} que não recebe nenhum input e faz o seguinte:

```
i=1
while(i < 0)
    if T_C(i) > |T_berry| + c
        return i
    i = i +1
```

O programa testa em ordem lexicográfica todos os strings[8] até achar a condição de que $T_C(i) > |T_{berry}| + c$, onde c é a constante a ser definida posteriormente, $|T_{berry}|$ é o tamanho do próprio programa, que pode ser calculado

[8]Note que o while nunca vai parar até atingir a condição estipulado no If, pois nunca vai cumprir a condição de que $i < 0$.

pela proposição 11 da existência de quines.[9] Quando achar esta condição, o programa retorna o string i. O programa deve encontrar esta condição em algum momento, pois sabemos que T_C é uma função ilimitada pela proposição anterior. Logo, o programa pára sempre que T_C pára, e retorna i.

Mas isso é uma contradição, pois a máquina T_{berry} tem tamanho $|T_{berry}|$ e retorna um string i com complexidade maior que sua própria complexidade.

Podemos exibir explicitamente a contradição da seguinte forma. Temos pela definição do programa:

$$C_{T_{berry}}(i) + c = |T_{berry}| + c < T_C(i) = C(i).$$

Pelo teorema 16 da invariância temos:

$$C(i) \leq C_{T_{berry}}(i) + d.$$

Logo:

$$C_{T_{berry}}(i) + c \leq C_{T_{berry}}(i) + d$$

Assim basta escolher $c > C_{T_{berry}}(i) + d$ para obter:

$$C_{T_{berry}}(i) > C_{T_{berry}}(i)$$

Contradição.

□

6.2 Incompletude de Chaitin

6.2.1 Strings aleatórios

Podemos definir um string como aleatório ou incompressível se o menor programa que o gera é aproximadamente de seu próprio tamanho. Em outras palavras, se o programa que o gera é incompressível, no sentido de que não podemos retirar suas irregularidades através de algum método de compressão (como compressões feitos por programas que analisam a regularidade estatística e buscam retirar redundâncias).

Definição **42. Strings aleatórios ou incompressíveis:** O string x é *aleatório ou incompressível* se $C(x) \geq |x|$. Um string é aleatório de tamanho k se $C(x) \geq k$ e $|x| = k$.

Mas tais strings aleatórios existem?

Um argumento combinatório pode nos responder sim a esta pergunta. Se pensarmos em strings de zeros e uns, temos 2^n possibilidades de strings de tamanho n (pelo princípio multiplicativo). Porém se pensarmos em programas

[9] Tal constante também pode ser estimada sem usar a proposição de existência de quines, para evitar o teorema do ponto-fixo, basta estimar este número pelos subprogramas que T_{berry} utiliza.

como strings, temos $\Sigma_{i=1}^{n-1} 2^i = 2^n - 1$ programas de tamanhos menores que n (assumindo que todo programa pode ser representado com um string de $\{0,1\}$, somamos todos estes programas e igualdade se segue da soma de P.G.). Suponha que todos descrevam pelo menos um dos 2^n strings maiores. Logo, pelo menos um string de tamanho n não é descrito por nenhum programa de menor tamanho. Logo, deve existir pelo menos um string x de tamanho n tal que $C(x) \geq n$ (e possivelmente outros).[10] Obtemos:

PROPOSIÇÃO **23.** *Existe um string aleatório de tamanho n.*

Com isto é possível ver uma linha raciocínio que leva à incompletude: um programa de tamanho k não pode provar que existe um número c aleatório muito maior do que k. Caso contrário poderíamos usar o próprio programa k para dar uma descrição do número c com apenas k bits. Contradição.

Como a aritmética é um programa, podemos usar esse raciocínio, o que faremos detalhadamente a seguir.

6.2.2 Incompletude de Chaitin

TEOREMA **17.** ***Incompletude de Chaitin:*** *Seja T uma extensão correta axiomatizável de $\mathbf{R}$. Então existe uma constante c (que depende de T) tal que T não prova a fórmula:*

$$C(x) > c$$

para todo x.

Demonstração. Faremos uma prova por contradição.

Em primeiro lugar, lembremos que pela proposição 15 o predicado Pr' é computável para qualquer teoria T. Logo, existe um programa de tamanho k que o computa. Em outra palavras k é a descrição dos teoremas de uma teoria, o que quer dizer que é uma descrição da própria teoria ($C(T) \leq k$).

Seja s_c o seguinte string:

$s_c :=$ " o menor string na ordem lexicográfica de tamanho c tal que T prova $C(s_c) \geq c$."

Ou seja, s_c é o menor string que é demonstrável em T ser aleatório de tamanho c. Notemos que, para todo c, existe algum menor s_c tal que $C(s_c) \geq c$ é verdade, pois a função $C(x)$ é ilimitada pela proposição 21 e, pela proposição anterior, deve existir um string aleatório de tamanho c. Logo, s_c está bem construída.

Construímos o seguinte programa $Prog$:

$Prog(\langle \ulcorner T \urcorner, c \rangle) =$ "achar nos teoremas de T a menor prova y tal que $Pr'(\ulcorner C(s_c) \geq c \urcorner, y)$ e devolver como output s_c."

[10] Note que não temos problema também com a proposição 20 devido à constante.

Para mostrar que $Prog$ é parcial computável fazemos o seguinte procedimento. $Prog$ pode encontrar s_c utilizando como subprogramas Pr' e conferindo se existe ou não uma prova de que $C(s_c) \geq c$ é teorema. Isto é, para todo código de prova y, testamos lexicograficamente para todo z de comprimento c se $Pr'(\ulcorner C(z) \geq c \urcorner, y)$ ou $Pr'(\ulcorner \neg C(z) \geq c \urcorner, y)$; e pegamos como s_c o menor z tal que $Pr'(\ulcorner C(s_c) \geq c \urcorner, y)$ (note que devido a ordem lexicográfica s_c é único).

Como, $Prog(\langle \ulcorner T \urcorner, c \rangle) = s_c$, logo pela pela proposição 19:

$$C_{Prog}(s_c) \leq |\langle \ulcorner T \urcorner, c \rangle|$$

E pelo teorema 16:

$$C(s_c) \leq C_{Prog}(s_c) + d \leq |\langle \ulcorner T \urcorner, c \rangle| + d$$

Utilizando que $\langle \cdot \rangle$ é a codificação mais eficiente, temos que ela pode ser feita com tamanho pelo menos $2k + |c| + 1 = 2k + log_2(c) + 1$ (ver prova do corolário do teorema da invariância e lembremos que k é o tamanho da descrição de T). Logo:

$$C(s_c) \leq 2k + log_2(c) + 1 + d.$$

Faça $c > 4(k + d)$.

Logo, substituindo c acima e por operações do logaritmo:

$$C(s_c) \leq 2k + log_2(k + d) + 3 + d \tag{6.1}$$

Mas assumimos que para todo x, em especial para $x = s_c$, $\vdash_T C(s_c) > c$. Portanto, pela correção de T:

$$C(s_c) \geq c > 4(k + d). \tag{6.2}$$

Que é muito maior do que o limite inferior da equação 1. (Note que $k + d$ da equação 2 é maior que $log_2(k + d)$) da equação 1, e que em geral $3(k + d)$ da equação 2 é maior que $2k + d + 3$ da equação 1). Contradição. Logo, para este $c > 4(k + d)$ escolhido que depende da teoria T, não é possível provar em T que $C(x) > c$ para qualquer x. É possível melhorar os limites inferiores de c. □

Corolário **13.** *T acima é incompleto.*

Demonstração. De fato, vimos que a função C é ilimitada e que portanto $C(x) > c$ é verdadeira, mas não demonstrável em T. Por outro lado, pela correção, por ser verdadeira, esta sentença não pode ser refutável. □

Seguimos a prova de Chaitin no artigo *Information Theoretic Computational Complexity*, teorema 2, que pode ser achado na coletânea de aritgos [Cha90] p.68; o livro texto [LV19] p.179 também dá uma prova semelhante. Acreditamos que esta prova é interessante porque ela faz um paralelo simples com a noção de aleatoriedade.

Em termos intuitivos, há algo do paradoxo da aleatoriedade nesta prova. Em termos não formais, um número é aleatório se ele é atípico. O paradoxo

é: não podemos dar um algoritmo que defina todos os números aleatórios, caso contrário os números aleatórios se tornariam típicos, e portanto deixaram de ser aleatórios e, por outro lado, os números típicos se tornariam aleatórios. Em resumo: moedas são muito mais eficientes do que computadores para gerarem aleatoriedade.

Porém, se aleatoriedade não é definível, caímos em um paradoxo quando definimos aleatoriedade como um string x tal que $C(x) > |x|$? Não, pois notemos que nossa definição de aleatoriedade possui uma característica fundamental: C não é computável (corolário 22). Ou seja, a universalidade da definição só foi atingida a partir de uma limitação como contraparte. Não é possível estabelecer com precisão completa quem são estes x aleatórios.

Lembremos que este mesmo tipo de limitação da classe de funções parciais computáveis foi o que permitiu a plausibilidade da tese de Church. Outras definições de aleatoriedades foram propostas, por Martin-Löf e Solomonov, e foram verificadas equivalentes assim como as definições de funções parciais computáveis. Esta convergência de definições levou à pergunta se não se tratava de um núcleo de definições igualmente estáveis. A tese de que todas as definições de aleatoriedade intuitivas são equivalentes é chamada de tese de Chaitin-Solomonov-Martin-Löf.

Tese de Chaitin-Solomonov-Martin-Löf: Tudo aquilo que é intuitivamente aleatório é equivalente às definições de aleatoriedade propostas por Chaitin (caso geral da que expomos), Martin-Löf ou Solomonov.

6.2.3 Prova alternativa

É possível também mostrar a incompletude de Chaitin utilizando o teorema do ponto-fixo, como salientado por [Raa98]. Porém, antes de iniciar a prova, exploraremos a definição alternativa de tratamento do assunto feita por [Odi99] p. 151 e [Raa98] em contraposição à desenvolvida por exemplo por Chaitin, [LV19], aqui e em outros textos. A principal diferença é que nessa definição alternativa de complexidade eles trabalham diretamente com o programa minimal, enquanto aqui trabalhamos com o tamanho do menor programa minimal. A diferença em termos quantitativos é da ordem exponencial, porém para efeitos dos resultados de incompletude não há grandes perdas. Argumentaremos a seguir detalhadamente o porquê.

No artigo, [Raa98] define complexidade como (mudamos a notação):

Definição **43.** Defina:

$$M_\phi(x) = \mu p(\phi_p(0) = x).$$

O operador μ é definida no contexto de funções recursivas parciais e se comporta assim: se a sentença $...x...$ é verdadeira para algum x, então $\mu x(...x...)$ é o menor x para o qual $...x...$ é verdadeiro.

A $\langle\cdot\rangle$ é a nossa função codificadora oficial que pode ser pensada como trabalhando diretamente com os strings. Note que nesta bijeção o 0 vai no string

vazio λ. Assim, se nós temos uma υ que funciona descrevendo strings utilizando essa codificação, como falamos na definição 39 de complexidade, temos:

$$\upsilon_p(\langle 0\rangle) = \upsilon(\langle p, 0\rangle) = \upsilon(\langle p\rangle)$$

A primeira igualdade é pelo teorema da enumeração, a segunda porque neste código $0 = \lambda$.

Logo, temos:

$$M_\phi(x) = \mu p(\phi_p(0) = x) = \mu p(\phi(p) = x) = \min\{p : \phi(p) = x\}$$

Bem, a última sentença é a definição do programa minimal x^* com respeito a ϕ. Logo, nós temos:

$$C(x) = |M_\phi(x)|$$

E nós sabemos que a bijeção lexicográfica funciona como em 17. Logo:

Proposição **24.**

$$C(x) = |M_\phi(x)| = \log M_\phi(x)$$

Assim, $M(x)$ se comporta exponencialmente com relação a $C(x)$.

Raatikainen diz na página 574 do artigo citado que $M_\phi(x)$ coincide com o jeito nosso de definir $C(x)$. Provavelmente ele quer dizer apenas com relação aos resultados de incompletude. Porque a função log é monotônica, fórmulas usando $\leq$ do tipo $(...\{x\}... \leq ...)$ podem ser transformadas em $(...C(x)... \leq ...)$ substituindo os números do tipo y por $\log(y)$. Neste sentido, nós poderíamos argumentar que: se $M_\phi(x) > c$ não é demonstrável na teoria T, então $C(x) > \log(c)$ não será demonstrável.

Portanto, pode ser argumentado que os resultados com $M_\phi(x)$ podem ser traduzidos para a complexidade $C(x)$, se nós os traduzirmos corretamente.[11]

Usaremos esta definição a seguir junto do teorema do ponto-fixo.

Teorema **18.** ***Incompletude de Chaitin versão do ponto-fixo:*** *Seja T uma extensão correta axiomatizável de $\mathbf{R}$. Então existe uma constante c (que depende de T) tal que T não prova a fórmula:*

$$C_\phi(x) > c$$

para todo x e ϕ uma função parcial computável universal.

Demonstração. Faremos a prova por contradição.

Existe uma função f computável tal que:

[11] Notemos que as enumerações admissíveis não irão ter um lugar aqui. Se nós tivermos uma enumeração de índices $\phi_0, ..., \phi_n, ...$, não poderíamos argumentar que há um sistema admissível de índices $\psi_0, ..., \psi_n$ tal que os índices $0, ..., n...$ são o tamanho dos programas. Porque nós temos mais de um programa do mesmo tamanho que nos dá o mesmo output, qualquer função computável f que vai do sistema de índices ψ para o sistema ϕ não seria sobrejetora.

$\phi_{f(z)}(0) =$ ache o menor y tal que $Pr(\ulcorner C_\phi(x) > z \urcorner, y)$
para algum x e dê como output x.

Pelo teorema 9 do ponto-fixo temos que existe um c tal que:

$$\phi_{f(c)} = \psi_c$$

Logo,

$\phi_c(0) =$ ache o menor y tal que $Pr(\ulcorner C_\phi(x) > c \urcorner, y)$
para algum x e dê como output x.

Pela hipótese $C_\phi(x) > c$ é demonstrável, logo $\phi_c(0)$ pára. E, pela definição alternativa de complexidade relativa a ϕ dada na proposição anterior, temos:

$$C_\phi(x) \leq M_\phi(x) \leq c.$$

Também pela hipótese de T provar $C\phi(x) > c$ e pela correção:

$$C_\phi(x) > c$$

Contradição.

□

E não precisamos usar o teorema da invariância neste teorema. Na realidade, não usamos nenhuma propriedade de complexidade que não a proposição 19, muito menos propriedades da complexidade aditivamente ótima. Este teorema poderia ser provado logo após a definição de complexidade (alternativa).

Mas, há alguma contrapartida para tamanha simplicidade? Sabemos que o ponto-fixo é um instrumento para trabalharmos com sentenças auto-referenciais. Uns dos contra-argumentos em tomar a incompletude de Gödel como pedra de toque da onipresença de fenômenos de indecidibilidade na matemática é justamente sua artificialidade. Muitas questões matemáticas aparentemente não possuem o caráter auto-referencial tal como exibido pelos teoremas da incompletude. E uma conclusão que se pode chegar é que em tais questões não há chance para fenômenos de indecidibilidade. Chaitin argumenta que sua versão do teorema da incompletude mostraria o caráter indecidível da matemática de forma mais explícita, pois sua fórmula é menos auto-referencial do que a de Gödel, justamente por evitar as diagonalizações encontradas no teorema do ponto-fixo. Mas várias perguntas surgem ao nos depararmos com esta afirmação. Calude e Jürgenstein tratam deste assunto em [CJ05].

6.3 A constante característica c é a medida de força de uma teoria?

6.3.1 Prova de trivialização da constante

Definição **44.** A *constante característica* de T (c_T) é a constante construída na prova do teorema de Chaitin e que depende da teoria T.

A constante característica c_T que aparece no teorema de Chaitin na seção precedente é a medida de força de T? A resposta curta é não. Porém vamos analisar esta questão melhor nesta seção.

A primeira coisa a se sinalizar sobre esta questão é que não se pode confundir de nenhuma maneira os teoremas em geral de uma teoria e os teoremas da forma $C(x) > c_T$. Se X é um teorema e $C(\ulcorner X \urcorner) > c_T$, isto não significa que $\nvdash_T X$. Vários contraexemplos nos lembram disso. O mais simples talvez seja o fato de que o teorema $x = x$ possui diversas instâncias na aritmética:

$$1 = 1, 2 = 2, ..., 10^{1000} = 10^{1000}, ...$$

Ou seja, se X é a sentença que varia sobre estas instâncias, X é teorema e é claro que $C(\ulcorner X \urcorner)$ é ilimitada, cresce logaritmicamente, e em especial para algum X, $C(\ulcorner X \urcorner) > c_T$, para qualquer c_T.

Logo, o que é limitado pela constante c_T não são os teoremas mesmos, mas os teoremas da forma $C(x) > c_T$.

Porém, esta consideração não elimina imediatamente a pergunta de se c_T pode de alguma forma ser pensada como medida de força em uma teoria. Raattikainen no artigo [Raa98] e Lambalgen no artigo [Lam89] analisam criticamente esta questão. Faremos um resumo a seguir.

O primeiro problema é pensar o que exatamente significa força de uma teoria. Por um lado gostaríamos de dizer que **ZFC** é mais forte que **PA**, pois prova mais coisas que **PA**. Logo, podemos pragmaticamente equiparar a definição de *força* e definição de extensão de uma teoria (embora talvez seja possível argumentar que a noção de força seja mais abrangente). Ou seja, uma teoria T_1 é mais *forte* que T_2 se T_1 é uma extensão de T_2.

Há problemas ainda com esta definição, pois pode haver conflito de linguagem, como o que acontece entre **PA** e **ZFC**. Mas podemos resolver isto observando que podemos definir uma linguagem padrão e inserir seus símbolos como extensões por definições. Adotamos este ponto de vista generalista e podemos afirmar por exemplo que **ZFC** é mais forte que **PA**, que por sua vez é mais forte que **R**.

O grande problema com relação às constantes vêm da seguinte proposição que aparece em [Raa98]. Utilizaremos o teorema 11 de Rogers.

Proposição **25.** *Seja T uma extensão axiomatizável correta de* **R**. *Então é possível construir uma função parcial computável universal ϕ tal que:*

$$\nvdash_T C_\phi(x) > 0$$

Demonstração. Provaremos por contradição.

Seja ψ uma função uni.p.c. que nos dá a seguinte enumeração:

$$\psi_0, ..., \psi_n, ...$$

Chamamos este sistema de índices $\pi_0 : \mathbf{N} \to \mathcal{FPC}$. Defina o seguinte sistema de índices troca-n para a enumeração de funções p.c. ϕ, que troca a enumeração dada em π_0:

$$\pi^n(x) = \begin{cases} 0 & \text{se } x = n \\ x+1 & \text{se } x < n \\ x & \text{se } x > n \end{cases}$$

Pelo teorema de Roger 11 os teoremas básicos de computabilidade valem para este sistema também. Defina uma noção de complexidade relativa ao sistema de índices π baseada nesta nova enumeração das f.p.c. ϕ: C_ϕ.

Fazendo o mesmo truque utilizando o teorema 9 do ponto-fixo que fizemos na demonstração do teorema 18 de Chaitin versão ponto-fixo, podemos definir uma p.c.u. ψ tal que:

$\psi_e(0) =$ Faça com que *este* programa tenha código (inicial) e.
E faça com que ele ache o menor y
tal que $Pr(\ulcorner C_\phi(x) > 0 \urcorner, y)$ para algum x
e dê como output x.

Pela hipótese $C_\phi(x) > 0$ é demonstrável, então $\psi_e(0)$ pára e nos retorna x. Logo $M_\phi(x)$ é determinada pelo índice de ψ_e no sistema de índices π^e. Mas este índice é dado por $\pi^e(e) = 0$, por definição. Logo, pela definição de programa mínimo e complexidade temos que:

$$C_\phi(x) \leq M_\phi(x) = 0$$

Por outro lado, pela correção e hipótese temos:

$$C_\phi(x) > 0$$

Contradição. □

Assim, no âmbito da constante característica c do teorema da incompletude de Chaitin, temos a seguinte consequência:

$$c_{\mathbf{ZFC}} = c_{\mathbf{PA}} = c_{\mathbf{R}} = 0.$$

Ou seja, a constante característica de qualquer outra teoria suficientemente forte seria igual a zero. Isto trivializaria qualquer tentativa de medir a força de uma teoria a partir de medidas de complexidades. Pois obviamente **ZFC** prova muito mais coisas que **PA** e para uma determinada medida de complexidade tanto **PA**

quanto **ZFC** não podem provar que teoremas possuem complexidade maior que zero.

Similarmente, [Raa98] mostra também que a constante característica das teorias pode crescer indefinidamente. E isto mais o argumento da proposição anterior mostram que qualquer tentativa de medir a força de uma teoria utilizando o teorema de Chaitin e a medida de complexidade C está fadada ao fracasso.

6.3.2 O princípio heurístico

Porém, uma objeção é que estas considerações de Raatikainen apenas mostram que esta medida de complexidade C específica não pode ser usada para medir a força de teorias. Mas, não mostrariam que uma outra medida de complexidade possível não seria suficiente para a tarefa. Felizmente este é o caso, e vamos mostrar resumidamente a seguir o porquê.

Calude no artigo *Is complexity a source of incompleteness?* ([CJ05]) define uma nova medida de complexidade δ_g. Esta medida é uma versão de uma outra medida de complexidade chamada *δ-aleatoriedade* definida da seguinte maneira:

$$\delta(x) = C(x) - |x|.$$

Esta medida é proporcional a aleatoriedade do string x, pois se temos um string aleatório x, então pela definição $C(x) \geq |x|$ e $\delta(x) \geq 0$. Ou seja, quanto mais aleatório x, mais $\delta(x)$ se aproxima de um número positivo c, e quanto menos aleatório for x, $\delta(x)$ será um número negativo, logo menor será $\delta(x)$. Logo essa medida é limitada por um c. O que será útil, pois a complexidade dos teoremas não irá crescer indefinidamente.

Calude considera uma medida δ_g variante dessa medida que leva em consideração os números de Gödel atribuído aos símbolos de uma teoria. Com isto ele consegue mostrar que nenhum teorema x é tal que $\delta_g(x) > c$. Ou seja consegue dar argumentos para o princípio heurístico de Chaitin, explícito no artigo *Gödel's Theorems and Information*, [Cha90] p. 113:

> A prova original de Gödel construiu uma asserção paradoxal que é verdadeira mas não demonstrável dentro de uma formalização usual da teoria dos números. Em contraste eu gostaria de medir o poder de um conjunto de axiomas e regras de inferência. Eu gostaria de poder dizer que se alguém tem dez quilos de axiomas e vinte quilos de teorema, então o teorema não pode ser derivado destes axiomas.

Porém devemos nos deter no entusiasmo aqui, pois não podemos usar isto para comparar teorias entre si. Isto é devido ao fato de que δ_g está ligada ao números de Gödel específicos utilizados pelos símbolos de T. Para dar um exemplo de [Raa98], poderíamos usar extensões da teoria,[12] definindo uma teoria com muitas outras constantes por exemplo, e colocando diversos axiomas

[12] Ver [Sho67], p. 33.

redundantes em uma teoria T'. Como δ_g depende dos símbolos da linguagem da teoria T em questão, isto mostra que não podemos usar δ_g para medir a complexidade de T'. Além do mais δ_g seria incomparável a uma $\delta_{g'}$ construída com os símbolos de T', pois $\delta_{g'}$ nos daria uma constante característica muito maior que a nossa δ_g original, apesar de T' ser uma extensão conservativa (ter o mesmos teoremas) de T.

Portanto, chegamos novamente a resposta que demos no início da seção: não. Não é possível comparar as forças de prova de uma teoria com outra a partir de uma medida de complexidade.

Capítulo 7

Conclusão

7.1 Comparando incompletudes

Vamos explorar aqui três aspectos em que os teoremas nos moldes gödelianos e de Chaitin diferem: 1) com respeito ao ponto-fixo; 2) com respeito à redução de um problema ao outro; 3) com relação à construtibilidade.

Em primeiro lugar, vimos que o teorema de Chaitin utiliza do teorema da invariância (que pressupõe somente o teorema da enumeração) para driblar o teorema do ponto-fixo. Neste sentido, ele não é autorreferente. Porém, ao utilizar o teorema da enumeração, ele está utilizando implicitamente que toda função parcial computável pode ser simulada por uma máquina universal. Embora tal fato em isolado não implique que há um ponto fixo para esta máquina universal, uma noção de imitação é utilizada. A máquina universal pode simular qualquer outra, inclusive a si mesma. A possibilidade de que ela pode simular a si mesma não implica que ela vai realmente fazer isto no caso da prova do teorema de Chaitin. Tal possibilidade fica apenas em potência quando provamos do teorema de Chaitin, porém é usada de alguma forma no teorema da incompletude nos moldes gödelianos.

Vamos fazer uma breve exposição quanto ao segundo ponto.

Em [Odi99][p.261] e [LP98][p.180], temos uma breve descrição de como realizar tal comparação em termos computacionais. Podemos enunciar o teorema de Gödel da seguinte forma:

> O conjunto dos teoremas de uma teoria suficientemente forte é computacionalmente enumerável.

Por outro lado o teorema de Chaitin pode ser enunciado:

> O conjunto dos teoremas de uma teoria suficientemente forte é computacionalmente enumerável, seu complemento é infinito e tal complemento não possui nenhum subconjunto infinito computacionalmente enumerável.

Ou seja, o teorema de Chaitin prova algo a mais: além de provar a existência de um conjunto de sentenças verdadeiras e não prováveis, prova também que não há um método de decisão para determinar quais sentenças são de tal forma. Pelo teorema de Gödel apenas podemos concluir que o conjunto de fórmulas verdadeiras não prováveis é infinito. Ou seja, dado um método para achar quais são as sentenças de Gödel, sempre temos um método para determinar quais são as sentenças de Chatin. Porém, dado um método para as sentenças de Chaitin, não temos um método para as sentenças de Gödel. O que quer dizer que o problema envolvido na obtenção das sentenças de Gödel é mais difícil do que o problema de achar as sentenças de Chaitin, pois mesmo se tivéssemos miraculosamente as sentenças de Chaitin, não conseguiríamos usar isso para computacionalmente achar as de Gödel.

De modo geral, dizemos que um problema A é *redutível* a um problema B ($A \leq_m B$), se um algoritmo hipotético para calcular B pode ser usado eficientemente para calcular o problema A. Se A não é reduzível a um problema B, dizemos que A é tão ou mais difícil que B.

O conjunto subjacente K_0 proveniente do conjunto dos teoremas prováveis do teorema de Gödel é diferente do conjunto B, dos teoremas considerados no teorema de Chaitin.[1] Podemos reduzir qualquer conjunto computacionalmente enumerável ao conjunto K_0, no sentido de que sempre existe uma função computável que computa se um elemento está em em um conjunto K_0 sempre que este elemento está no conjunto c.e. X ($K_0 \leq_m X$). Assim, como B é um conjunto c.e., ele pode ser reduzido a K_0 ($K_0 \leq_m B$), porém o contrário não vale ($B \not\leq_m K_0$), o que quer dizer que o problema envolvido em decidir se algo está em K_0 é mais difícil do que o envolvido em B. Dizemos que tais conjuntos que são reduzíveis a K_0 são chamados *completos* para a classe de problemas c.e. Por exemplo ambos os conjuntos K_0 e o conjunto dos programas que param (problema da parada) são conjuntos completos, o que quer dizer que ambos problemas podem ser reduzidos um ao outro.

O conjunto B não é completo. Apesar de B ser reduzível a K_0, K_0 não pode ser reduzível a B.

Em terceiro lugar, como vimos, o teorema da incompletude pode ser derivado utilizando métodos construtíveis providenciados pelo teorema do ponto-fixo. Por outro lado, como a função complexidade C não é computável, não podemos construir a sentença de Chaitin.

7.2 Caminhos futuros

Algumas questões que podem ser exploradas em uma futura pesquisa:

- Uma pergunta óbvia com relação à seção anterior é: qual a relação específica entre tais noções? Há alguma relação entre a existência de uma função de ponto-fixo e a existência do conjunto B?

[1] Observar que K_0 coincide, sob este ponto de vista, com o conjunto parada da definição 20. Isto quer dizer, ambos estes conjuntos são reduzíveis entre si.

- Embora o teorema de Chaitin não use o teorema do ponto-fixo, usa o teorema da universalidade. Ali como salientamos está presente uma noção informal de simulação que não parece poder ser evitada. É possível explorar tal definição de forma mais abstrata? Em quais outros contextos matemáticos tal noção informal de simulação age?

- Vimos na introdução que o argumento do ponto-fixo pode ser generalizado para contextos como a teoria dos conjuntos. Porém, há uma generalização mais frutífera, pois mais geral, é o chamado teorema do ponto-fixo de Lawvere. Neste teorema é utilizado a linguagem abstrata das teorias da categorias para generalizar ainda mais a questão. É possível obter o teorema da incompletude de Chaitin neste contexto?

Capítulo 8

Apêndice A: Notação, Definições Matemáticas e Preliminares Lógicos

Este capítulo é referente às definições matemáticas e lógicas de apoio às definições específicas trabalhadas no texto.

8.1 Definições Matemáticas

Começaremos com o que acreditamos ser uma das distinções mais importantes, entre função parcial e função total:

1. Denotamos relações pelas letras $R, P, Q, ...$ Relações são subconjuntos de produtos cartesianos $X_1 \times ... \times X_n$, onde X_i são conjuntos. Ou seja são subconjuntos de $\{\langle x_1, ...x_n\rangle : x_1 \in X_1, x_2 \in X_2, ..., x_n \in X_n\}$, onde $\langle x_1, ...x_n\rangle$ são n-uplas (i.e. conjuntos com alguma ordem interna, também pode ser representada por $(x_1, .., x_n)$).

2. Denotamos as funções **parciais** pelas letras finais do alfabeto grego:

   $$\phi, \psi, \chi, \xi, ...$$

   Uma *função parcial* $\phi : X \to Y$ é uma relação n-ária R de tal forma que, para todo $\langle x_1, ..., x_{n-1}\rangle \in X$, existe *no máximo* um elemento y de Y tal que $\langle x_1, ..., x_{n-1}, y\rangle \in R$. X e Y são chamados respectivamente *domínio* e *contradomínio*. Dito de outro modo, uma função parcial pode estar definida apenas em um subconjunto $Z \subseteq X$ do domínio e para o restante $(X - Z)$ indefinida.

3. Denotamos as funções **totais** pelas letras intermediárias do alfabeto latino:

   $$f, g, h, d, ...$$

Uma *função total* $f : X \to Y$ é uma relação n-ária R tal forma que, para todo $\langle x_1, ..., x_{n-1}\rangle \in X$, existe *exatamente* um y elemento de Y tal que $\langle x_1, ..., x_{n-1}, y\rangle \in R$. Dito de outro modo, uma função total é uma função parcial definida necessariamente em todo o domínio. Assim, toda função total é parcial.

4. Letras minúsculas denotam elementos $x, y, z, w, v, ...$de conjuntos. Denotamos conjuntos gerais (ou de preferência numéricos) pelas letras maiúsculas: $X, Y, Z, W, V, ...$;

5. Denotamos também conjuntos (de preferência não numéricos) pelas letras maiúsculas: $A, B, C, ..$; letras minúsculas denotam os elementos $a, b, c...$

6. Denotamos uma indexação de um conjunto X pelo conjunto I (finito ou infinito) por $\left\{x_n\right\}_{n\in I}$. Indexações são funções totais $x : I \to X$ tais que $x_i = x(i)$. O conjunto indexado pode ser chamado *família*. Além disso, a união de uma família de subconjuntos de X pode ser escrita nesta notação como $\cup_{i\in I} X_i$, assim como outras notações derivadas.

7. Denotamos fórmulas de uma teoria T [1] pelas letras $F, G, H, ...,$.

Lógica e Teoria dos conjuntos. Usaremos $\neg, \vee, \wedge, \to, \leftrightarrow, \forall, \exists$ (não, ou, e, implica, sse[2], para todo, existe) para símbolos da lógica de primeira ordem. Importaremos da teoria dos conjuntos o símbolo $\in$ e outros símbolos convencionais derivados deste, como $A \subseteq B \Leftrightarrow \forall x(x \in A \to x \in B)$, ou seja, $\subseteq$ é a inclusão não-estrita. Os outros símbolos $\{a, b\}$ (par de elementos), $\cup$ (união), $\cap$ (intersecção), $-$ (diferença de conjuntos), A^c (complemento) etc... são definíveis como no livro *Naive Set Theory* de Halmos [Hal98].

Fecho de uma relação: Seja X um conjunto e R uma relação binária em X. Considere $Y \subseteq X$. O *fecho* de Y por R, denotado por $[Y]_R$ (ou $\overline{Y}$), é dado pela seguinte condição: $x \in [Y]_R$ sse $x \in Y$ ou existe $y \in Y$ tal que xRy. Por exemplo, o fecho no subconjunto $Y = \{a, b\}$ de X pela relação R, com $X = \{a, b, c, d\}$, e $R = \{\{a, b\}, \{b, c\}, \{a, c\}$, é $\overline{Y} = \{a, b, c\}$.

Operações Matemáticas. Definiremos $\log_x y = z$ quando o número z for o logaritmo de y na base x, ou seja, sse $x^z = y$. Para $x \in \mathbb{R}$, $\lfloor x\rfloor$, é a aproximação de x por baixo, $\lceil x\rceil$ é a aproximação de x por cima. Por log fixamos a função logarítmica de base 2, $\log_2$. Por exemplo, $\lfloor \log 10\rfloor = \lfloor \log_2 10\rfloor = \lfloor 3.3\rfloor = 3$.

Utilizamos $a \equiv b \mod c$ para significar que a é equivalente a b módulo c.

Notação de Bachmann: Utilizamos a nomenclatura $f(x) = O(g(x))$ para dizer que $f(x)$ é igual a $g(x)$ a menos de uma constante c multiplicativa. Dito de outro modo, $f(x) = O(g(x))$ sse existem c e x_0 tal que $f(x) \leq cg(x)$, para todo $x > x_0$.

[1] Mais sobre a lógica de primeira ordem adotada será explicitado no capítulo sobre Incompletude de Gödel.

[2] Abreviação de "se, e somente, se".

Ordens e equivalência: Sejam A um conjunto e R_A (subscrito dispensável) uma relação definida dentro deste conjunto. Dizemos que R_A é uma relação de **pré-ordem** em A sse as seguintes propriedades se seguem:

1. (reflexividade) para todo $x \in A$, xRx ;
2. (transitividade) para todos $x, y, z \in A$, se xRy e yRz, então xRz.

Dependendo da propriedade que adicionarmos a esta pré-ordem temos um conceito novo. Por um lado, se além destas propriedades temos a propriedade da antissimetria — isto é, se para todos $x, y \in A$, xRy e yRx, implica $x = y$ — então chamamos esta relação de **ordem parcial** (e denotamos por $\leq$). Por outro lado, se adicionarmos a propriedade da simetria — para todos $x, y \in A$, se xRy, então yRx — então temos uma **relação de equivalência** ($\sim$).

Ainda mais, caso uma ordem parcial possua a propriedade da totalidade (todo elemento é comparável) — isto é , se para todos $x, y \in A$ ou xRy ou yRx — então dizemos que a relação é uma **ordem total**.

Ordem total estrita: Dados A um conjunto e $\leq$ uma ordem total definida neste conjunto, então $<$ é uma relação de *ordem estrita* total induzida por $\leq$ definida através da seguinte equivalência: $x < y$ sse $x \leq y$ e $x \neq y$.

8.2 Definições Lógicas

Esta seção será particularmente importante como apoio ao capítulo sobre a incompletude de Gödel. Definiremos alguns preliminares encontrados em qualquer livro de lógica. Em especial tomaremos um misto da nomenclatura de [Sho67] com uma notação mais flexível.

8.2.1 Sintaxe

Definição **45. Símbolos lógicos da linguagem de primeira ordem:** O conjunto $L(F)$ dos *símbolos de uma linguagem* de primeira ordem são:

- Variáveis, ordenadas lexicograficamente como: $x_1, ..., x_n, ...$ Nomenclatura: $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{x'}, ...$ ou $\mathbf{x_1}, ...\mathbf{x_n}$. Dando a entender que podemos ordenar alfabeticamente esta nomenclatura da forma mais conveniente;
- Símbolos de funções n-árias. Nomenclatura: $\mathbf{f}, \mathbf{g}, \mathbf{h}, \mathbf{f'}, \mathbf{f_1}...$ (aridade dada pelo contexto), para funções; $\mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{e'}...$ (aridade 0) para constantes;
- Símbolos de predicados n-ários e $=$. Nomenclatura: $\mathbf{P}, \mathbf{R}, \mathbf{S}, \mathbf{P'}, \mathbf{P_1}...$ para predicados e (incluso aridade 0, símbolo ao qual damos o nome também de *proposição*).
- Os símbolos: $\neg, \vee, \exists$.

Assinatura ou linguagem: Uma linguagem ou assinatura

$$L = \{\mathbf{f_1}, ..., \mathbf{f_n}, \mathbf{P_1}, ..., \mathbf{P_n}, ...\}$$

é determinada por um conjunto de predicados e funções, i.e. *símbolos não lógicos* (com exceção de =, considerada como simbolo lógico e já subentendida em todas linguagens).

Atenção nomenclatura flexível: *Adotaremos uma postura flexível ao longo do texto com relação a nomenclatura:* ***não*** *diferenciaremos a notação para variáveis metalinguísticas (i.e. variáveis que variam em expressões da linguagem e é feita em negrito normalmente) e símbolos do sistema formal a não ser que seja muito necessário (em definições ou esquemas de axiomas, por exemplo, que realmente necessitam desta diferenciação).* Ou seja, se o leitor vir um negrito, estamos querendo chamar atenção que se trata de uma variável metalinguística. Por exemplo, na definição acima a nomenclatura está em negrito, e isto quer dizer que estes símbolos são variáveis que variam nas expressões de fato do sistema formal. Veja que não definimos de fato símbolos para funções ou predicados, que serão especificadas para cada teoria ou estrutura. Por exemplo para teoria dos conjuntos temos a seguinte linguagem com apenas um símbolo: $L = \{\in\}$.

DEFINIÇÃO **46. Termos:** Defina indutivamente *termo*:

- Toda variável é um termo;
- Se $\mathbf{x_1}, ..., \mathbf{x_n}$ são termos, então $f\mathbf{x_1}, ..., \mathbf{x_n}$ é termo.[3]

DEFINIÇÃO **47. Fórmulas**
Defina indutivamente *fórmula*:

- Se $\mathbf{x_1}$ e $\mathbf{x_2}$ são termos, então $= \mathbf{x_1 x_2}$ é fórmula;
- Se $\mathbf{x_1}, ..., \mathbf{x_n}$ são termos, então $\mathbf{P x_1}, ..., \mathbf{x_n}$ é fórmula;
- Se $\mathbf{F}$ é fórmula então $\neg\mathbf{F}$ é fórmula;
- Se $\mathbf{F}$ e $\mathbf{G}$ são fórmulas, então $\vee\mathbf{FG}$ é fórmula;

  (observação: para facilitar a visualização, não usaremos normalmente esta notação infixa, mas a usual $F \vee G$, porém no que concerne a linguagem de fato, temos em mente esta construção específica. Seguiremos esta construção para simplificação de algumas passagens, e facilitar a comparação com o livro de Shoenfield [Sho67] e outros da literatura)
- Se $\mathbf{F}$ é fórmula, então $\exists\mathbf{x}\mathbf{F}$ é fórmula.

O conjunto formado pelos dois primeiros itens são chamados *fórmulas atômicas.*

[3] Aqui já está incluído constantes, pois n pode ser igual a zero.

Definição **48.** Vamos assumir que o leitor está familiarizado com a noção de variável livre e ligada. Usaremos $\mathbf{F}(\mathbf{x_1}, ..., \mathbf{x_n})$ para denotar que $\mathbf{x_1}, ..., \mathbf{x_n}$ são variáveis distintas livres na fórmula $\mathbf{F}$. Usaremos $\mathbf{F}(\mathbf{a_1}, ..., \mathbf{a_n})$ para denotar a substituição das variáveis distintas $\mathbf{x_1}, ..., \mathbf{x_n}$ pela constantes $\mathbf{a_1}, ..., \mathbf{a_n}$ respectivamente.

Para especificar qual e quantas variáveis exatamente estamos substituindo, usaremos a notação: $\mathbf{F}_{\mathbf{x_1},...,\mathbf{x_n}}[\mathbf{a_1}, ..., \mathbf{a_n}]$. Dizemos que uma fórmula é uma *sentença* se ela não possui variáveis ou se todas variáveis estão ligadas. Assim se $\mathbf{F}(\mathbf{a_1}, ..., \mathbf{a_n})$ for uma sentença, $\mathbf{x_1}, ..., \mathbf{x_n}$ serão todas as variáveis que F possui. Omitiremos e colocaremos quantos parênteses (e variantes) forem necessário para melhorar a legibilidade da notação polonesa. E também usaremos os usuais símbolos de $\forall, \wedge, \leftrightarrow, \rightarrow$, o símbolo cortado, por exemplo $\neq$, para negação de um símbolo, com as definições usuais.

8.2.2 Semântica

Definição **49. Estrutura:** Uma L-estrutura $\mathcal{A}$ para uma linguagem L é constituída dos seguintes componentes:

- Um conjunto A chamado *universo*.
- Para cada constante $\mathbf{c}$, um elemento $c^{\mathcal{A}}$ do universo A.
- Para cada símbolo não lógico de função não zero-ária $\mathbf{f}$, uma função $f^{\mathcal{A}} : A^n \to A$.
- Para cada símbolo não lógico de predicado $\mathbf{P}$ um predicado $P^{\mathcal{A}} \subseteq A^n$.

Adotaremos uma semântica Robinsoniana[4]. Esta abordagem é talvez menos conhecida, mas é feita por Shoenfield [Sho67], e outros, com variações, Boolos [BJB12] por exemplo. Qualquer outra abordagem seria equivalente. O importante é que consigamos definir a noção de satisfação e validade, verdade, etc... A ideia é introjetaremos o universo da estrutura (possivelmente infinito) dentro da linguagem através do seguinte mecanismo:

Definição **50. Linguagem diagrama:**
Para cada L-estrutura $\mathcal{A}$ e linguagem L, introduzimos constantes $\mathbf{a}$ (i.e. símbolos de funções zero-árias) em L para cada elemento a do conjunto universo A de $\mathcal{A}$, com símbolos condizentes chamados *nomes*, produzindo assim uma nova linguagem $L(\mathcal{A})$. Isto é:

[4]Note que estas definições são diferentes da abordagem Tarskiana de livros como por exemplos clássicos da lógica como Mendelsohn *Mathematical Logic*. A grande diferença é que nesta semântica não temos uma interpretação de variáveis livres como funções, mas como nomes indeterminados. Escolhemos esta abordagem porque a noção de variáveis como nomes, parece ser mais clara no contexto dos resultados que iremos descrever. Ver uma discussão disto em Buton e Walsh *Philosophy of Model Theory*. Porém o resultado é em termos práticos o mesmo. É possível também retardar o acréscimo dos infinitos nomes na linguagem através de mecanismos descritos em [BJB12].

$$L(\mathcal{A}) = L \cup A',$$

onde $\mathbf{a} \in A'$ sempre que $a \in A$.

Denominamos como *linguagem diagrama* $L(\mathcal{A})$ a nova linguagem assim formada pelo acréscimo destas novas constantes, e usamos a nomenclatura i e j para os nomes de $L(\mathcal{A})$.

Definição **51. Interpretação de termos e fórmulas de uma linguagem:** Definiremos a função interpretação $\mathcal{A}(x)$ (não confundir com a estrutura) da linguagem L na estrutura $\mathcal{A}$:

$\mathcal{A}:$ fórmulas fechadas e termos livres de variáveis de $L(\mathcal{A}) \to \{0,1\} \cup A$

Definimos indutivamente interpretação para termos $\mathbf{t}$, elementos do domínio A:

- Se $\mathbf{t}$ é um nome, então $\mathcal{A}(\mathbf{t}) = t^{\mathbf{A}}$.
- Se $\mathbf{t}$ é $f\mathbf{t_1}...\mathbf{t_n}$, então $\mathcal{A}(\mathbf{t}) = f^{\mathcal{A}}(\mathcal{A}(t_1), ..., \mathcal{A}(t_n))$.

Definimos indutivamente interpretação para fórmulas fechadas $\mathbf{F}$ (em que todas suas variáveis estão ligadas), $\mathcal{A}(F) = 0$ (falso) ou 1 (verdadeiro) de acordo com:

- Se $\mathbf{F}$ é $\mathbf{t} = \mathbf{u}$, então $\mathcal{A}(\mathbf{F}) = 1 \Leftrightarrow \mathcal{A}(\mathbf{t}) = \mathcal{A}(\mathbf{u})$.
- Se $\mathbf{F}$ é $\mathbf{P}\mathbf{x_1}, ..., \mathbf{x_n}$, então $\mathcal{A}(\mathbf{F}) = 1 \Leftrightarrow \mathbf{P}^{\mathcal{A}}(\mathcal{A}(\mathbf{x_1}), ..., \mathcal{A}(\mathbf{x_n}))$.
- Se $\mathbf{F}$ é $\neg\mathbf{G}$, então $\mathcal{A}(\mathbf{F}) = 1 \Leftrightarrow \mathcal{A}(\mathbf{G}) = 0$.
- Se $\mathbf{F}$ é $\mathbf{G} \vee \mathbf{H}$, então $\mathcal{A}(\mathbf{F}) = 0 \Leftrightarrow \mathcal{A}(\mathbf{G}) = 0$ e $\mathcal{A}(\mathbf{H}) = 0$.
- Se $\mathbf{F}$ é $\exists x\mathbf{G}$, então $\mathcal{A}(\mathbf{F}) = 1 \Leftrightarrow$ existe um nome $\mathbf{a} \in L(A)$ tal que $\mathcal{A}(\mathbf{G_x}[\mathbf{a}]) = 1$.

Definição **52. Verdade em uma estrutura e Modelos:** Uma fórmula (possivelmente aberta) $\mathbf{F}$ é *verdadeira* (ou satisfeita) *em uma estrutura* $\mathcal{A}$ se para todos $\mathbf{a_1}, ..., \mathbf{a_n}$ elementos de $L(\mathcal{A})$, a sentença livre de variáveis decorrente das substituições $F(\mathbf{a_1}, ..., \mathbf{a_n})$ é interpretada em $\mathcal{A}$ como $\mathcal{A}(F(\mathbf{a_1}, ..., \mathbf{a_n})) = 1$. Usaremos, nesse caso, a nomenclatura:

$$\mathcal{A} \models \mathbf{F}$$

Ademais se Γ é um conjunto de fórmulas, dizemos que $\mathcal{A}$ é um *modelo para* Γ se para toda fórmula $\mathbf{F} \in \Gamma$, $\mathcal{A} \models \mathbf{F}$. Escrevemos:

$$\mathcal{A} \models \Gamma.$$

Definição **53. Validade e consequência lógica:** Dizemos que uma fórmula **F** é *válida* se para toda estrutura $\mathcal{A}$, temos $\mathcal{A} \models \mathbf{F}$. E usamos, nesse caso, a nomenclatura:

$$\models \mathbf{F}.$$

Dizemos que a fórmula **F** é *consequência lógica* de um conjunto de fórmulas Γ se em todo modelo $\mathcal{A}$ de Γ, $\mathcal{A} \models F$. Usamos a nomenclatura:

$$\Gamma \models \mathbf{F}.$$

Definição **54. Teoria de um modelo:** A *teoria* da estrutura $\mathcal{A}$ é o conjunto de suas fórmulas verdadeiras: $Teo(\mathcal{A}) := \{F \in L(\mathcal{A}) : \mathcal{A} \models F\}$.

Capítulo 9

Apêndice B: Código de uma implementação em Python de uma simulação de máquinas de Turing

```
import sys

print("Este programa simula uma máquina de Turing. \nInsira o
↪  estado final. Ex: q_4\n")

est_final =  str(input()).strip()


print('Insira o número de instruções (ex.10):')

n_instrucoes = int(input())
#pontos
print("Sintaxe do programa: O imput deve ser como no ex:
↪  \n$q_0$0R$q_0$, $q_0$1R$q_0$, $q_0$BL$q_1$, $q_1$B0$q_4$,
↪  $q_1$01$q_4$, $q_1$10$q_2$, $q_2$0L$q_3$, $q_3$10$q_2$,
↪  $q_3$01$q_4$, $q_3$B0$q_4$ \n\nInsira o programa:")

programa = str(input()).split(',')

print("Insira o input:")
x_input = str(input())

config = '$q_0$'+ x_input #configuração inicial
```

```
est_atual = 'q_0'

print( "------------------\n\nPrograma:\n", programa,
↪  "\n\nConfiguração atual:\n", config, "\nEstado atual:\n",
↪  est_atual, "\nEstado final:\n",  est_final, '\nAlgum
↪  problema? (Se não, tecle quaquer tecla.)')

input()

print("Instrução", "\t x\t", "Configuração")

while (est_atual != est_final):
#acha o string ao lado direito e esquerdo de q_i na configuração
↪  atual.
        direita = config.split('$')[2]
        esquerda = config.split('$')[0]
#se q_i está no final do string, a máquina está com a cabeça
↪  lendo B.
        if(direita == ''):
                cabeca = 'B'
        else:
                cabeca = direita[0]
#se q_i está no começo do string, a máquina tem o rabo B
        if(esquerda == ''):
                rabo = 'B'
        else:
                rabo = esquerda[-1]
#acha a instrução correspondente
        j=0
        for i in programa:
                if('$'+ est_atual + '$' + cabeca in i):
                        instr_atual = i
                else:
                        j = j+1
#se não há instrução correspondente, a máquina pára e temos o
↪  output.
        if j == len(programa):
                break
        print( instr_atual, '\t x \t', config)
        novo_est = instr_atual.split("$")[3]
        novo_simbolo = (instr_atual.split("$")[2])[1]
#substituições (note que se não existir letra à direita, isso não
↪  nos causará problema) (definições de passo: item 1):
        if((novo_simbolo == '0' or novo_simbolo == '1' or
        ↪  novo_simbolo == 'B') and direita!=''):
```

```
                config = config.replace('$'+ est_atual + '$' +
                ↪  cabeca , '$'+ novo_est + '$' + novo_simbolo)
#substituição (definição: item 2)
        if((novo_simbolo == '0' or novo_simbolo == '1' or
        ↪  novo_simbolo == 'B') and direita==''):
                config = config.replace('$'+ est_atual + '$'
                ↪  ,'$'+ novo_est + '$' + novo_simbolo)
# temos um R: há letra à direita (definição: item 3):
        if(novo_simbolo == 'R' and direita!=''):
                config = config.replace('$'+ est_atual + '$' +
                ↪  cabeca , cabeca + '$'+ novo_est + '$')
# temos um R: não há letra à direita, coloca-se um B à esquerda
↪  (definição: item 4):
        if(novo_simbolo == 'R' and direita==''):
                config = config.replace('$'+ est_atual + '$', 'B'
                ↪  + '$'+ novo_est + '$')
# temos um L: há letra à esquerda (definição: item 5):
        if(novo_simbolo == 'L' and esquerda !='' and direita !=
        ↪  ''):
                config = config.replace(rabo + '$'+ est_atual +
                ↪  '$' + direita , '$'+ novo_est + '$' + rabo
                ↪  +direita)
#temos um L: não há letra à direita (definição: item 6)
        if(novo_simbolo == 'L' and esquerda !='' and direita ==
        ↪  ''):
                config = config.replace( rabo + '$'+ est_atual +
                ↪  '$', '$'+ novo_est + '$' + rabo)
# temos um L: não há letra à esquerda, coloca-se um B à esquerda
↪  (definição: item 7 e 8):
        if(novo_simbolo == 'L' and esquerda ==''):
                config = config.replace('$'+ est_atual + '$',
                ↪  '$'+ novo_est + '$' + 'B')
#atualizando o estado atual
        est_atual = novo_est

# saímos do loop, devemos agora retornar o output
print("configuração de parada: ", config,
↪  "\n--------------------------------------")

x_output = config.split("$")[0] + config.split("$")[2]

print("Output:", x_output)

print("fim")
```

Bibliografia

[Bek06] Lev Beklemishev. *Provability, Computability and Reflection*. Unpublished manuscript, 2006. URL: http://www.mi-ras.ru/~bekl/Prf-Ar/book-drf1.pdf.

[BJB12] George S. Boolos, Richard C. Jeffrey e John P. Burgess. *Computabilidade e Lógica*. São Paulo: Unesp, 2012. ISBN: 978-85-393-0366-3.

[Cal02] Cristian Calude. *Information and Randomness: An Algorithmic Perspective*. 2 ed. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2002. ISBN: 9783540434665, 3540434666.

[Cal07] Cristian Calude. *Randomness and Complexity: from Leibniz to Chaitin*. World Scientific Publishing, 2007.

[CCD11] Gregory J. Chaitin, Newton da Costa e Francisco Antonio Doria. *Gödel's Way. Exploits into an undecideble world*. Taylor & Francis, 2011.

[CE09] Walter Carnielli e Richard L. Epstein. *Computabilidade, Funções Computáveis, lógica e os Fundamentos da Matemática*. 2ª ed. editora UNESP, 2009.

[Cha02] Gregory J. Chaitin. *The LIMITS of MATHEMATICS: A Course on Information Theory and the Limits of Formal Reasoning*. Discrete Mathematics and Theoretical Computer Science. Springer London, 2002.

[Cha03] Gregory J. Chaitin. *Algorithmic Information Theory*. Cambrigde University Press, 1987 (impressão 2003).

[Cha09] Gregory J. Chaintin. *Metamat!: Em Busca do Ômega*. São Paulo: Perspectiva, 2009.

[Cha90] Gregory J. Chaitin. *Information, Randomness and Incompletness*. 2ª ed. World Scientific, 1990.

[Cha92] Gregory J. Chaitin. *Information-Theoretic Incompleteness*. Series in Computer Science Series. World Scientific, 1992.

[Cha99] Gregory J. Chaitin. *The Unknowable*. Discrete Mathematics and Theoretical Computer Science. Springer Singapore, 1999.

[CJ05] Cristian S. Calude e Helmut Jürgensen. “Is complexity a source of incompleteness?” Em: (2005).

[Dav04] Martin Davis. *The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions*. Dover Ed. Dover Publications, 2004. ISBN: 0486432289, 9780486432281.

[Des96] René Descartes. *Discurso do Método*. 2ª. Livraria Martins Fontes Editora Ltda., 1996.

[Fra05] Torkel Franzén. *Gödel's Theorem. An Incomplete Guide to its Use and Abuse*. Wellesley, Massachuseetss: A K Peters, 2005.

[Hal98] P. R. Halmos. *Naive set theory*. 1ª ed. Undergraduate texts in mathematics. Springer-Verlag, 1998. ISBN: 0387900926,9780387900926.

[Hin05] Peter G Hinman. *Fundamentals of Mathematical Logic*. Wellesley: A K Peters, 2005.

[Lam89] Michiel van Lambalgen. “Algorithmic information theory”. Em: *Jorunal of Symbolic Logic, 1389-1400* (1989). DOI: 10.1017/S0022481200041153.

[Law06] F. William Lawvere. “Diagonal Arguments and cartesian Closed Categories”. Em: 15 (2006). URL: http://www.tac.mta.ca/tac/reprints.

[LP98] Harry Lewis e Christos H. Papadimitriou. *Elements of the Theory of Computation*. 2 ed. Prentice-Hall, 1998. ISBN: 0132624788,9780132624787.

[LV19] Ming Li e Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications (Texts in Computer Science)*. 4th ed. 2019. Texts in Computer Science. Springer, 2019. ISBN: 3030112977, 9783030112974.

[Odi99] Piergiorgio Odifreddi. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers, Vol. 1*. 1 ed. Vol. 1. Studies in Logic and the Foundations of Mathematics 125. Elsevier, 1999. ISBN: 9780444894830, 0444894837.

[Owi73] James C Owings. “Diagonalization and the recursion theorem, 14, no. 1, 95–99”. Em: *Notre Dame J. Formal Logic* (1973). DOI: 10.1305/ndjfl/1093890812.

[Raa98] Panu Raattikainen. “On interpreting Chaitin's incompleteness theorem”. Em: (1998).

[Rog87] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. 1 ed. The MIT Press, 1987. ISBN: 0262680521, 9780262680523.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[Smi13] Peter Smith. *An Introduction to Gödel's Theorem*. 2ª ed. Cambridge University Press, 2013.

[Smo88] C.A. Smoryński. “Hilbert's Programme”. Em: Logic Group preprint series (1988).

[Smu92] Raymond M. Smullyan. *Gödel's Incompleteness Theorems*. Oxford Logic Guides. Oxford University Press, 1992. ISBN: 0195046722-97814-2373-5199-9780195046724-0814758169.

[Smu94] Raymond M. Smullyan. *Diagonalization and Self-Reference*. Oxford Logic Guides 27. Clarendon Press, 1994. ISBN: 0198534507-9780198-534501.

[Soa16] Robert I. Soare. *Turing Computability: Theory and Applications*. 1 ed. Theory and Applications of Computability. Springer, 2016. ISBN: 978-3-642-31932-7.

[Web12] Rebecca Weber. *Computability theory*. Student Mathematical Library 062. American Mathematical Society, 2012. ISBN: 082187392X,978-0-8218-7392-2.

[Wey49] Hermann Weyl. *Philosophy of Mathematics and Natural Science*. Princeton University Press, 1949.

[Wit08] Ludwig Wittgenstein. *Tractatus Logico-Philosophicus*. 3ª ed. edusp, 2008.

[Yan03] Noson S. Yanofsky. "A Universal Approach to Self-Referential Paradoxes, Incompleteness and Fixed Points". Em: 9 (2003). DOI: 10.2178/bsl/1058448677.