O Teorema da Incompletude de Gödel

Amanda Figur*

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
ac.figur@gmail.com

Resumo

Com uma demonstração do primeiro Teorema da Incompletude de Gödel que exige somente conhecimentos básicos sobre números naturais, funções e uma linguagem de programação, você vai descobrir que existem afirmações que podem ser demonstradas sem você ter certeza se o que entendeu foi o oposto do que te disseram.

1. Introdução

Provado no início de 1930 e publicado no *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme*, o primeiro Teorema da Incompletude de Gödel foi um marco para a matemática no século 20, mostrando que a matemática não era um objeto finalizado e consistente, como muitos[1] acreditavam.

O Teorema enuncia que:

Toda teoria rica o suficiente, axiomatizável e completa é inconsistente.

... e esse enunciado imediatamente nos leva a algumas perguntas:

- *Quando uma teoria é completa?* Se para qualquer afirmação na teoria existe uma demonstração (na teoria) para a afirmação ou para a negação da afirmação.
- *Quando ela é inconsistente?* Quando uma afirmação e sua negação podem ser ambas demonstradas (chamamos isso de uma contradição).
- *O que é uma teoria axiomatizável?* Para nós, uma teoria "axiomatizável" seria a existência de um programa executado em tempo finito que verifica se algo é uma demonstração. Formalizar o que é uma teoria axiomatizável não cabe ao escopo deste texto.

Escrevendo de outro modo, o teorema prova que toda teoria "rica o suficiente", axiomatizável, que prove ou refute qualquer afirmação da própria teoria, gera contradições.

Entenderemos melhor o que "rica o suficiente" quer dizer ao longo do texto.

*Agradecimentos ao Leandro F. Aurichi e a todos os revisores

[1] David Hilbert, por exemplo. Dá uma olhada na lista de problemas que o próprio fez em 1900. Você também pode tentar resolver alguns. ;)

1.1. Comentários sobre a demonstração

Neste texto vamos apresentar uma prova simplificada do primeiro Teorema da Incompletude de Gödel, diferente da original formulada pelo próprio Gödel.

A seguinte demonstração foge de grandes discussões filosóficas acerca de sua prova e evita o maquinário pesado de lógica matemática presente em demonstrações mais formais. Para entendê-la é recomendado que você tenha alguns conhecimentos básicos sobre uma linguagem de programação, números naturais e funções.

O que vamos fazer ao longo deste texto é[2], na seção 2, fixar uma linguagem de programação à nossa escolha e então exibir uma função f não computável. Em seguida, na seção 3, vamos definir uma teoria onde seja possível construir a função f.[3] Munidos disso (e de algumas outras coisas), na seção 4 utilizaremos um programa, chamado `EhUmaDemonstracao`, que recebe uma afirmação e verifica se ela é uma demonstração. O intuito de tudo isso é chegar em uma contradição sobre a não computabilidade de f: se nossa teoria fosse completa e consistente seria possível descrever um programa que calcula os valores de f. Assim concluíremos que nossa teoria axiomatizável e completa é, na verdade, inconsistente.

2. Uma função não-computável

Comece fixando uma linguagem de programação P, pode ser algo como C, Pascal, Fortran, Python, etc. A linguagem é finita, isto é, tem finitos símbolos e cada programa gerado é finito. Mas isso não quer dizer que a quantidade de programas possíveis tenha tamanho limitado; pense nos números naturais - temos finitos algarismos, cada número usa finitos algarismos e a quantidade de números é infinita. De fato a quantidade de programas possíveis de serem feitos em P é enumerável. Um conjunto C é dito **enumerável** quando há uma injeção do conjunto C para os naturais $\mathbb{N}$.

Para provar a enumerabilidade dos programas vamos dar para cada símbolo da nossa linguagem um número. Note que unindo vários símbolos estaremos unindo também vários números e formando um número maior. Poderíamos associar então para cada programa o número gerado pela concatenação dos números de cada símbolo. Mas como garantir que esse número seja único[4]? Veja essa sequência de símbolos:

$\wedge$	$'$	0	$\sim$	$\neg$	!	)	(	=	+	.	$*$
100	101	102	103	104	105	106	107	108	109	110	111

Associamos para cada um símbolo um número de três dígitos. Isso se deve ao fato de não querermos dupla interpretação para uma frase. Suponha que tivéssemos os símbolos A e B com os números 1 e 11, respectivamente. Não saberíamos se o número 111 representaria "AB", "AAA", "BA" ou "$*$". Tendo cada número representado por três dígitos saberemos que 111103 representa somente "$* \sim$". Claro que 3 dígitos seria a melhor representação para uma linguagem com até 900 símbolos. Para maiores quantidades poderiam ser 4 dígitos, 5 ou o "n" que melhor se adequasse. Assim poderíamos associar para cada programa de nossa linguagem um número único formado pela concatenação de números de n dígitos.

[2] Spoiler alert!

[3] A nossa teoria (e não a nossa linguagem de programação) é, portanto, "rica o suficiente" para ser possível construir e calcular cada valor da função f.

[4] Perceba que, se existe uma função associando cada programa a um único número, tal função é uma injeção de P para os naturais $\mathbb{N}$. Isso prova que a quantidades de programas em P é enumerável.

Definição 1. *Seja $f : \mathbb{N} \to \{0,1\}$. Vamos dizer que f é uma **função computável** se existe um programa feito na linguagem P que, ao receber o valor n, devolve $f(n)$.*

Como a quantidade de programas possíveis em P é enumerável, podemos ter apenas uma quantidade enumerável de funções computáveis, que podem ser enumeradas. Mas não precisamos nos restringir apenas a isso: é possível *ordená-las*.

Uma função computável f possui um ou mais programas que atestam sua "computabilidade". Se usarmos uma enumeração como a definida acima para os programas da nossa linguagem P, poderíamos tomar o menor número a associado aos programas que computam f e então "renomear" f para f_a. Deste modo[5] nossas funções computáveis podem ser ordenadas de acordo com a ordem da enumeração dos programas de P.

Da definição sabemos o que é uma **função não computável**: uma função para a qual não existe programa feito na linguagem P que ao receber o valor n, devolve $f(n)$. Para provar que existem tais funções vamos olhar a ilustração abaixo:

		0	1	2	3	4	5	6	...
f_0	=	$\underline{0}$	1	1	0	1	0	0	...
f_1	=	0	$\underline{0}$	1	0	0	1	1	...
f_2	=	1	1	$\underline{1}$	0	1	1	0	...
f_3	=	1	0	0	$\underline{1}$	0	0	0	...
f_4	=	0	0	1	1	$\underline{1}$	0	1	...
f_5	=	1	1	0	0	1	$\underline{0}$	0	...
f_6	=	0	1	0	0	1	0	$\underline{1}$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
f	=	$\underline{1}$	$\underline{1}$	$\underline{0}$	$\underline{0}$	$\underline{0}$	$\underline{1}$	$\underline{0}$	...

Figura 1: *A ilustração representa uma lista de todas as funções computáveis (cada uma aparece na lista apenas uma vez). Para entender melhor olhe para cada igualdade: à esquerda, f_i é uma dessas funções, à direita temos o valor de f_i em cada número natural (representado pelas colunas) de seu domínio. A função f foi escrita com base nos valores de cada $f_i(i)$, indicados pelo sublinhado.*

Listamos acima a imagem de todas as funções computáveis (em f_0 temos $f_0(0) = 0, f_0(1) = 1, f_0(2) = 1, f_0(3) = 0, f_0(4) = 1, f_0(5) = 0, f_0(6) = 0, ...$). Podemos definir então um novo elemento f tomando para cada um dos elementos listados um elemento diferente do conjunto: $f(0) = 1$, diferente de $f_0(0) = 0$, ou seja, f é diferente de f_0, em seguida $f(1) = 1$, diferente de $f_1(1)$. Continuando deste modo teremos que cada $f(i)$ será diferente de $f_i(i)$, ou seja, f será diferente de cada uma das enumerações f_i e é, portanto, um novo elemento.

Note que esse novo elemento, definido de tal forma, é uma função não computável, pois havíamos listado *todas* as funções computáveis e definimos uma que não estava nessa lista[6].

[5] É possível formalizar esse argumento construindo injeções. É um exercício interessante, mas o importante aqui é entender que essa ordenação existe. Argumentos desse tipo podem ser mais intuitivos se pensados assim: se existe uma injeção de um conjunto C para um conjunto D, então a quantidade de elementos em C é limitada (menor ou igual) pela quantidade de elementos em D.

[6] Com um raciocínio semelhante (chamado método de diagonalização de Cantor) poderíamos provar a não enumerabilidade do conjunto das funções $f : \mathbb{N} \to \{0,1\}$, ou seja, provar que não existe uma injeção das f's para os naturais, isto é, "sobram" funções e "faltam" números. Aqui essas funções "excedentes" são chamadas de funções não computáveis. Estas, de fato, são infinitas, mas para nós a existência de pelo menos uma já é suficiente.

Vamos então fixar $f : \mathbb{N} \to \{0,1\}$, uma função não computável. Note que para podermos explicitar a função f, basta utilizar um método de ordenação para nossas funções f_n. Já sabemos que esse método existe. Estando elas ordenadas, saberemos o valor de $f(n)$ pois sabemos o valor de cada $f_k(k)$.

3. Uma teoria para definir a função f

Definição 2. *Vamos chamar de uma **teoria** uma coleção de axiomas e algumas regras de inferência.*

Informalmente, uma regra de inferência é uma maneira de se "obter" afirmações a partir de anteriores. Por exemplo, uma possível (e bastante comum) regra é a que diz que a partir das afirmações "A" e "$A \to B$", obtemos a afirmação "B" (essa regra é conhecida por *modus ponens*, que em português podemos ler como "sabemos que A vale e que toda vez que A vale, B vale, então B vale"). Outro exemplo é a regra de substituição: se temos "$\varphi(x)$" e temos que "$x = y$", então temos "$\varphi(y)$", onde φ é uma propriedade qualquer.

Ao fixar uma teoria, também estamos fixando os símbolos que podem ser usados nas afirmações (em geral, são coisas como $\to$, $\forall$, v, $'$, etc).

Vamos fixar para o decorrer deste texto uma teoria T que seja "rica o suficiente" para que possamos definir e explicitar a função f fixada acima[7].

Definição 3. *Fixada uma teoria, uma **demonstração** nada mais é que uma sequência $\varphi_1, ..., \varphi_n$ onde cada φ_k é um axioma ou existem φ_i's com $i < k$ de forma que φ_k é obtida a partir de uma das regras de inferência da teoria a partir das φ_i's.*

Para entender melhor o que é uma demonstração vejamos um exemplo na teoria de grupos. Podemos demonstrar, a partir de um dos axiomas de grupo (o que garante a existência do elemento neutro), a unicidade do elemento neutro da operação definida. Veja:

$\exists\, e \in G$	$\forall\, a \in G$	$(e \cdot a = a) \wedge (a \cdot e = a)$	axioma da existência do elemento neutro
$(\exists\, x \in G$	$\forall\, a \in G$	$(x \cdot a = a)) \to (x = e)$	afirmação: se x é um elemento neutro, $x = e$
		$x \cdot e = x$	uso do axioma para e
		$x \cdot e = e$	uso do axioma para x
		$x = e$	uso de um axioma lógico
$(\exists\, x \in G$	$\forall\, a \in G$	$(x \cdot a = a)) \to (x = e)$	afirmação demonstrada

4. Programando

Agora com a nossa linguagem de programação P fixada vamos fazer alguns programas sobre resultados em nossas teorias, pois precisamos de métodos para saber se uma afirmação é ou não uma demonstração. É importante destacar aqui que é crucial que os programas possam ser executados em tempo finito, afinal seria difícil encontrar um computador que rodasse um programa para sempre.

[7] Como exemplo de teoria "rica o suficiente" para explicitar f poderíamos citar o sistema de axiomas da teoria dos conjuntos conhecido como **ZFC**. Citaremos ele novamente mais adiante.

4.1. Um programa que gera provas

Façamos um programa `Provas` na nossa linguagem de programação P que gere todas as possíveis demonstrações da teoria T. Não é necessário que o programa dê como resultado somente demonstrações, ele pode gerar bastante lixo[8]. O problema é que tal programa não seria executado em tempo finito, afinal são infinitas demonstrações. Para melhorá-lo, vamos estabelecer ele de maneira que receba um $n \in \mathbb{N}$ e devolva uma sequência de símbolos presentes em T. Assim, para qualquer demonstração d na nossa teoria T existe n tal que o programa `Provas` aplicado a n resulte em d.

Tal programa `Provas` pode ser definido da maneira análoga à listagem de todos os programas possíveis que exemplificamos, visto que as demonstrações são finitas.

Por exemplo, suponha que nossa linguagem (de T) possui menos que 900 símbolos. Associe a esses símbolos números distintos entre 100 e 999. Quando nosso programa `Provas` receber um número $100 \leq a \leq 999$ ele retorna o simbolo correspondente. Depois, ao receber o número $a_1a_2a_3b_1b_2b_3$ o programa retorna o símbolo de número $a_1a_2a_3$ concatenado com o símbolo de número $b_1b_2b_3$, e assim por diante. Assim, sendo o símbolo "A" associado ao número 739 e o símbolo "B" associado ao número 387, o programa ao receber o número 739387 retornaria "AB". Ou seja, nosso programa verificaria se nosso número tem uma quantidade de algarismos múltipla de 3 e em seguida verificaria se grupos de 3 números da esquerda para a direita (ou ao contrário, vai de sua preferência) estão associados a um símbolo.

Quando um número não satisfaz esses requisitos, como aconteceria com o número 4253, nosso programa retornaria uma sequência de símbolos fixa, como por exemplo "##". Precisamos acrescentar também um símbolo extra, com o número, digamos, 999 (o último), a ser colocado no final de cada afirmação, assim nosso programa representará as demonstrações em afirmações do tipo $\varphi_1, \varphi_2, ..., \varphi_n$.

4.2. Um programa que verifica demonstrações

Vamos implementar outro programa chamado `EhUmaDemonstracao` que recebe dois parâmetros: uma sequência de símbolos e uma afirmação φ de T. Daí ele deve

- Retornar 1 se a sequência é uma demonstração para φ;
- Retornar 0 se a sequência é uma demonstração para $\neg\varphi$ (negação de φ).

Uma demonstração para φ nada mais é que uma demonstração cuja última afirmação é a própria φ. Lembre-se do nosso exemplo da unicidade do elemento neutro, a afirmação final era a afirmação que queríamos provar.

Esse é o programa que atesta que nossa teoria é axiomatizável[9].

4.3. Um programa que verifica afirmações

Definição 4. *Dizemos que uma teoria é* ***completa*** *se, para toda afirmação φ existe uma demonstração para φ ou uma para $\neg\varphi$.*

[8] Dependendo da complexidade da linguagem da teoria T, nada impede que o programa gere uma "prova" como aquelas que os professores dão aos alunos no fim do semestre.

[9] Caso você tenha problemas em acreditar que um programa como esse existe lembre-se que compiladores (que são programas) executam uma tarefa semelhante através de análise léxica, sintática e semântica.

Suponhamos T completa. Existe um programa `Provou` que recebe uma afirmação φ de T e deve

- Retornar 1 se ela admite uma demonstração;
- Retornar 0 se $\neg\varphi$ admite uma demonstração.

Na verdade podemos explicitar tal programa:

```
PARA n = 0, ENQUANTO n == n, FAÇA :
      SE(EhUmaDemonstracao(Provas(n), φ) == 1)
          RETORNA 1;
      SE(EhUmaDemonstracao(Provas(n), ¬φ) == 1)
          RETORNA 0;
      n++;
FIM-PARA
```

Note que o programa só roda em tempo finito porque sabemos que T é completa.

4.4. Um programa especial para a função f

Notamos que se T é completa, para cada afirmação do tipo "$f(n) = 1$" ou "$f(n) = 0$" (f é a fixada acima), existe uma demonstração para "$f(n) = 1$" ou uma para "$f(n) = 0$".

Isso acontece porque, como nossa teoria é completa, se não existe uma demonstração para "$f(n) = 1$", existe uma demonstração para "$f(n) \neq 1$". Mas "$f(n) \neq 1$" é justamente "$f(n) = 0$", pois o contra-domínio da nossa função f possui somente esses dois elementos: 0 e 1.

Definição 5. *Dizemos que uma teoria é* ***consistente*** *se não existe uma afirmação φ tal que existam provas para φ e para $\neg\varphi$.*

Suponhamos T completa e consistente. Notamos então que, para cada afirmação do tipo "$f(n) = 1$", ou existe uma demonstração para ela, ou existe uma demonstração para "$f(n) = 0$", mas não ambos os casos.

Note que com isso seria possível fazer um programa que calcule $f(n)$.

Faça um programa chamado `String` que recebe n e retorna a string "$f(n) = 1$". Feito isso, basta olhar para o seguinte programa:

```
SE(Provou(String(n)) == 1)
  RETORNA 1;
SE NAO
  RETORNA 0;
FIM-SE
```

Quando a afirmação "$f(n) = 1$" é demonstrável o programa retorna o valor 1 e quando a afirmação "$f(n) = 0$" é demonstrável o programa retorna o valor 0. Mas isso é um programa que recebe o valor n e retorna $f(n)$, um absurdo, já que nossa f é não computável.

Esse programa depende que nossa teoria T seja completa e consistente, mas ele só gera contradições quando assumimos a consistência de T: no caso de uma teoria inconsistente o programa não necessariamente calcula o valor de $f(n)$.

Deste modo, provamos que uma teoria axiomatizável, rica o suficiente para definir f e completa não pode ser consistente sem gerar contradições.

5. Uma vez incompleta, sempre incompleta

Resumindo, mostramos que T é uma teoria tal que:

1. É suficientemente rica para podermos definir e exibir a função f;
2. é axiomatizável;
3. é consistente;
4. é completa;

E chegamos a uma contradição. Desta forma, não existe uma teoria axiomatizável, rica o suficiente para definir f, que prove ou refute qualquer afirmação da própria teoria e não gere contradições.

Como uma consequência do primeiro Teorema da Incompletude de Gödel, podemos pensar então que dada uma afirmação φ que não possui demonstração para φ nem para $\neg\varphi$, bastaria acrescentar a afirmação φ como um axioma. Assim, T continuaria rica o suficiente para definir a função f, assim como consistente. Mas, se ela for completa então poderíamos aplicar o teorema novamente e chegar a uma nova contradição. Concluímos que T é tal que podemos aplicar o teorema feito neste texto e não adianta acrescentarmos algum axioma que ela continuará sendo incompleta.

6. Indo um pouco além

6.1. Um breve histórico

Em setembro de 1930, Gödel anunciou o resultado desse teorema em uma conferência em *Königsberg* (a cidade se chama Kaliningrado atualmente[10]).

Na época do acontecimento, a demonstração deste teorema foi um grande impacto nas tentativas de fundamentar as bases da matemática, que sofriam de paradoxos [11] e inconsistências.

Essas tentativas já aconteciam na época de Bertrand Russell, que tentou fundamentar a aritmética no seu *Principia Mathematica*, publicado em três volumes entre 1910 e 1913, mas foram melhor definidas quando David Hilbert, no início da década de 1920, formulou o que viria a ser conhecido como Programa de Hilbert.

Entre as coisas que o programa de Hilbert almejava estavam, além da formalização, a prova da completude e consistência de uma teoria que seria a base da matemática[12].

6.2. Resultados subsequentes

Um resultado curto sobre o Teorema da Incompletude de Gödel nos mostra que a axiomática de Tarski para a geometria euclidiana, que é completa e consistente, não é, portanto, "rica o suficiente"[13].

Hoje em dia a matemática se fundamenta no sistema de axiomas da Teoria dos Conjuntos *Zermelo-Fraenkel*, denominado **ZF**, ou **ZFC**, como ele é mais conhecido quando em conjunto do Axioma da Escolha[14].

[10] 2017

[11] Um paradoxo "famoso" até hoje é o chamado *Paradoxo de Russell*.

[12] Em sua última análise, toda teoria matemática poderia ser reduzida à aritmética.

[13] Tal teoria não seria capaz de descrever os números naturais e todas suas propriedades (a aritmética citada anteriormente) e, portanto, nela não é possível construir a função não computável utilizada na demonstração.

[14] *Axiom of Choice*, em inglês. O "C" vem de "Choice".

Paul Cohen provou em 1963, em conjunto a resultados anteriores de Kurt Gödel, que o Axioma da Escolha é independente dos outros axiomas de Teoria dos Conjuntos, isto é, os axiomas de Teoria dos conjuntos não provam a afirmação nem a negação do Axioma da Escolha.

ZFC não pode provar sua própria consistência de acordo com o resultado conhecido como *segundo Teorema da incompletude de Gödel*. Mas isso aí já é papo pra um outro texto. ;)

Referências

[1] Leandro Aurichi. Lista: Teorema da incompletude de gödel. `http://conteudo.icmc.usp.br/pessoas/aurichi/exerc/doku.php?id=lista:incompletude`, 2015.

[2] D. Gusfield. Gödel for Goldilocks: A Rigorous, Streamlined Proof of (a variant of) Gödel's First Incompleteness Theorem. *ArXiv e-prints*, September 2014.

[3] Richard Zach. Hilbert's program. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2016 edition, 2016.