Teoria da Computação

Computabilidade e Complexidade

Neto Coelho

19 de Novembro de 2009

Conteúdo

Capítulo 1

Introdução

1.1 Um pouco de História...

A necessidade humana da matemática recua, pelo menos, ao início da História. Há registos de conceitos matemáticos em sociedades tão antigas como as da Suméria, da China, do Egipto, da Índia e da Grécia, não só aplicadas em tarefas práticas, como a contabilidade ou a astronomia — na adivinhação astrológica, na previsão dos os ciclos das colheitas, essenciais às sociedades agrícolas de então — como também em actividades mais abstractas como a exploração de padrões e de relações entre números ou formas geométricas.

Talvez a mais antiga máquina de cálculo que conhecemos seja um artefacto conhecido como **Antikythera**. Após análises recentes (publicadas na revista *Nature* em 2006 e em 2008) admite-se que seja uma calculadora astronómica para seguir os ciclos solares e lunares e talvez até as órbitas dos planetas conhecidos na altura. Assim, será um dos primeiros planetários conhecidos[1].

Muitos conceitos matemáticos foram pensados e ensinados como pro-

[1] A história das máquinas celestes como os relógios, planetários, astrolábios, menires e outras construções é riquíssima e diversificada pelas culturas e pelos séculos. Nesta breve introdução apenas temos oportunidade para referir alguns dos exemplos mais destacados.

Figura 1.1: Tábua YBC7289, *circa* 1800–1600 a.C. Colecção Universidade de Yale. Esta tábua babilónica mostra 1.41421296 como uma aproximação da raiz quadrada de dois ($\sqrt{2} = 1.41421356\ldots$). A qualidade da aproximação dá-nos um vislumbre da matemática que se fez há três mil e quinhentos anos.

cessos invariáveis para que o leitor interessado, executando correctamente os passos descritos, possa obter a solução de um dado problema. A este tipo de processo chamamos, hoje em dia, «**algoritmo**» e à pessoa ou máquina que aplica tais algoritmos chamamos «**computador**»[2].

Os Chineses foram pioneiros no uso de sistemas físicos como apoio ao cálculo manual. Nas mesas de cálculo, por volta do século IV a.C, as operações eram realizadas com pauzinhos para representar os dígitos, colocados sobre uma área quadriculada, para organizar e manipular os números. Esta forma de realizar cálculos persistiu durante muito tempo até serem definitivamente substituídos pelo **ábaco**, introduzido três séculos antes e que fora ganhando influência pelo território do Império,

[2]Desde o século XVII até meados do século XX, «computador» — literalmente, *aquele que computa* — era a profissão da pessoa que aplica algoritmos, com lápis e papel.

Figura 1.2: O mecanismo da *Antikythera*, um planetário portátil com dois mil anos, encontrado no fundo do Mediterrâneo em 1901 e, pensa-se, fabricado no século II a.C.

por ser mais rápido e eficiente. Muitas obras clássicas ensinam a manipular estes sistemas não só para efectuar as quatro operações básicas mas também para outros cálculos mais sofisticados.

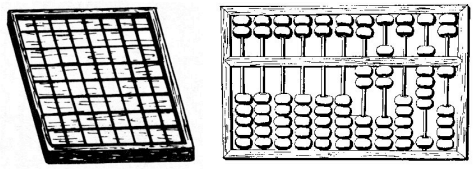


Figura 1.3: Uma tábua de cálculo e um Ábaco.

O termo «algoritmo» deriva do nome latinizado *Algoritmi* do matemático persa **Abdullah Muhammad bin Musa al-Khwarizmi** (750–830) cujas obras estudam vários problemas algébricos e mostram métodos «algorítmicos» de resolução algébrica. Não sendo ele o primeiro fazê-lo (pois conhecemos descrições de algoritmos milhares de anos mais

velhas) os seus livros, como o *Álgebra* ou o *Aritmética*, foram fulcrais na nova disseminação da Matemática pela Europa Medieval, após a Idade das Trevas que se seguiu à queda do Império Romano. É graças a estas obras que foi introduzida, nas nações europeias, a **notação posicional** indiana usada hoje em dia.

Antes de surgirem máquinas para executar algoritmos estes procedimentos eram necessariamente efectuados por pessoas, fossem matemáticos, contabilistas, navegadores... Era somente necessário seguir as instruções do livro ou dadas pelo professor que os ensinasse. Com o progresso tecnológico começaram a surgir, primeiro esboços, depois protótipos, de máquinas automáticas para realizar tais cálculos, com maior ou menor assistência humana.

Figura 1.4: Uma Pascaline, uma das máquinas construídas por Pascal (exibição no *Musée des Arts et Métiers* em Paris.) Cada roda corresponde a um dígito. Basicamente uma máquina de somar, permite também fazer subtracção através do cálculo de complementos.

O matemático e filósofo francês **Blaise Pascal** (1623–1662) foi um dos primeiros a construir uma máquina calculadora, em 1640. A **Pascaline**, ilustrada na figura 1.4, realiza somas e subtracções e foi (então) concebida para facilitar a contabilidade das empresas. É operada através de rodas e cada uma corresponde a um dígito do resultado final. Tem uma roda para as unidades, outra para as dezenas, *etc.*, num total de seis. Cada roda age sobre a seguinte, no caso de haver um excesso na soma. Curiosamente, a forma como está construída permite ape-

nas somar (e somente números com até seis dígitos, dado haver apenas seis rodas). Para subtrair é necessário passar por um passo intermédio: determinar o complemento do número a subtrair e *somar* esse complemento, obtendo-se, dessa soma, o resultado da subtracção[3].

Houve várias propostas de melhorar a Pascaline, mais ou menos implementadas e de variável sucesso, até que, por volta de 1820, o engenheiro e matemático inglês **Charles Babbage** (1791–1871) projectou e construiu parcialmente o que é hoje considerado o primeiro computador no sentido moderno do termo: uma máquina capaz de efectuar todos os passos de um dado algoritmo, sem assistência humana. Ligada a este projecto está a primeira mulher historicamente associada à computação, a inglesa **Ada Lovelace** (1815–1852), considerada a primeira programadora (e programador) devido ao trabalho que produziu para Babbage, criando programas para serem executados num outro projecto que, porém, nunca chegou a ser concluído: uma máquina analógica, em oposição ao carácter discreto das máquinas da altura, e também, dos computadores modernos. Ada extrapolou as possibilidades destes sistemas, argumentando que poderiam ser usados para outras tarefas além da mera manipulação contabilística, como por exemplo em aplicações gráficas ou na criação de música.

Apesar do trabalho de Babbage, o primeiro computador efectivamente construído e usado para realizar diversas tarefas práticas programáveis só surge em 1936, na Alemanha: o **Z1** de **Konrad Zuse** (1910–1995). Logo de seguida, durante a 2.ª Grande Guerra, surgem os computadores americanos **Mark 1** e **ENIAC**. Estes computadores podem ser programados por um conjunto de instruções individualmente muito simples que manipulam directamente posições de memórias e registos

[3]O complemento de n é o sucessor de um certo número m, tal que $n + m$ seja composto apenas por noves. Por exemplo, para calcular $200 - 123$, primeiro obtém-se o complemento de $n = 123$, que é (para três dígitos), $m + 1 = 877$ (porque $n + m = 123 + 876 = 999$). De seguida, somando $200 + 877$ obtém-se 1077, um número com quatro dígitos. Finalmente, usando apenas três dígitos, o 1 da esquerda perde-se e fica 77, o resultado da conta $200 - 123$. Este procedimento é semelhante ao usado actualmente nos computadores modernos para realizar subtracções (mas usando a notação binária em vez da decimal).

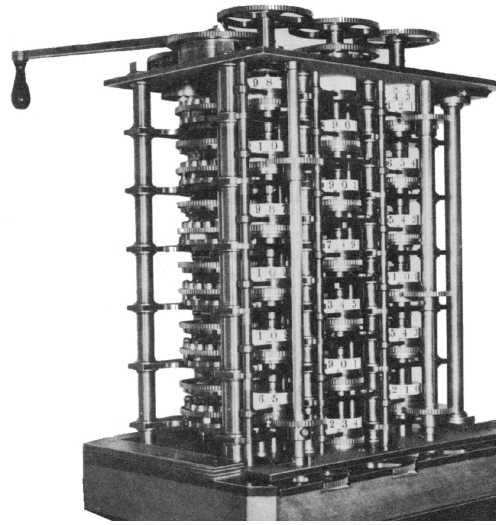


Figura 1.5: A máquina de Babbage. Existe uma réplica moderna em exibição no Museu de Ciências de Londres.

das respectivas máquinas (uma linguagem de programação composta por este tipo de instruções costuma designar-se por *assembler*).

1.2 Computabilidade

Paris, 1900. O matemático alemão **David Hilbert** (1862–1943) apresenta aos seus colegas uma lista de 23 problemas chave, um desafio aos matemáticos do século XX. Esta lista inclui o que Hilbert considera serem os mais importantes problemas em aberto, e que os matemáticos devem esforçar-se por resolver. Um dos problemas dessa lista pergunta, *grosso modo*, se existe algum processo mecânico para descobrir a verdade (ou falsidade) das conjecturas matemáticas. Isto é, Hilbert quer saber se *toda a matemática pode ser explorada, e descoberta, por algum processo automático*, um problema que passou a ser conhecido por **entscheidungsproblem** (literalmente, «o problema da decisão»).

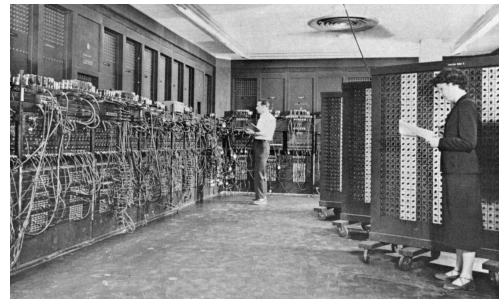


Figura 1.6: O ENIAC, toneladas de computador em acção.

Três décadas mais tarde, em 1931, **Kurt Gödel** (1906–1978) formaliza matematicamente esta questão e encontra a resposta à questão de Hilbert, mas não da forma que este estaria à espera. Gödel conclui que qualquer teoria que abranja a aritmética[4] tem de ser ou inconsistente ou incompleta. Sendo inconsistente, uma teoria é inútil pois não distingue «verdade» de «falso». Sendo incompleta, certas verdades não podem ser provadas. Assim, descontando as teorias inconsistentes, os matemáticos têm, para sempre, de trabalhar em teorias incompletas.

No entanto, Gödel deixou por resolver a seguinte questão: *dada uma teoria, será possível descobrir mecanicamente as proposições que não podem ser provadas?* Se esta pergunta tiver resposta positiva, tais proposições podem ser isoladas e trabalhar-se somente com as restantes. Caso contrário, a aritmética ficará para sempre «infectada» de proposições cujo valor lógico não pode ser apurado.

Outro problema relacionado com a questão de Hilbert é o de atribuir um significado a «descobrir mecanicamente». Desde há muitos séculos que se usam operações mecânicas para produzir resultados numéricos. Ou seja, conhecendo uma «receita», basta seguir os seus passos, um a um, para que, a partir dos dados iniciais, se chegue ao resultado final desejado. E isto ignorando de todo o conhecimento subjacente que orienta

[4]Uma teoria sem aritmética não suporta as noções de «número» nem de «soma», ingredientes imprescindíveis para *começar* a lidar com problemas gerais.

a escrita dessa receita. Nós próprios, na escola, aprendemos algumas receitas destas: como somar, subtrair, multiplicar, dividir... Em resumo, após o contributo de Gödel, continuou em aberto saber exactamente o que significa «**algoritmo**». Interessa dar uma definição matemática rigorosa à ideia intuitiva[5]

> *Um algoritmo é um conjunto de regras bem definidas que nos informa, a cada momento, o que fazer a seguir.*

O problema com esta descrição é que se baseia na capacidade da pessoa para executar o algoritmo; para interpretar correctamente as regras que o regem. Alguém incapaz poderá não entender a explicação do algoritmo ou, por outro lado, um observador demasiado capaz (ou totalmente fora do contexto) pode introduzir factores secundários não previstos por quem escreveu essas regras.

Uma forma melhor de definir algoritmo, para além do conjunto das regras, passa por fornecer igualmente os detalhes da *interpretação* dessas mesmas regras. Assim, para definir algoritmo é necessário explicitar

a) uma linguagem que introduz a sintaxe das regras;
b) um mecanismo de interpretação de regras nessa linguagem;
c) um conjunto de regras escritas na linguagem anterior.

O próprio Hilbert havia introduzido o conceito de **funções primitivamente recursivas** e conjecturara que este seria equivalente ao conjunto das *funções algorítmicas*. No entanto, em 1928, **Wilhelm Ackermann** (1896–1962) encontrou uma função algorítmica que não é primitiva recursiva: a função de Ackermann.

Depois do trabalho de Gödel, surgiram outras formalizações para o conceito de algoritmo. **Alonzo Church** (1903–1995), entre 1933 e 1936, introduziu o **cálculo** λ como nova formalização de algoritmo. Em 1936 **Stephen Kleene** (1909–1994) derivou o conceito de **função parcialmente recursiva**, mostrando a diferença relativamente às funções

[5]Embora não seja óbvio que se possa formalizar completamente esta complexa noção intuitiva.

primitivamente recursivas e ao mesmo tempo, provando a equivalência com o cálculo λ. Uma função dir-se-ia computável se e só se fosse uma função parcialmente recursiva. Outros formalismos surgiram, como a computação proposta por **Emil Post** (1897–1954) que também se demonstrou ser equivalente às restantes propostas.

Quando estudava em Cambridge, o matemático inglês **Alan Turing** (1912–1954) ouviu falar do *entscheidungsproblem* e tentou resolvê-lo. Levando à letra a expressão «processo mecânico» de Hilbert, concebeu, em 1936, um modelo de uma máquina capaz de executar algoritmos, no sentido das alíneas a)-c) acima. Hoje em dia designamo-la por **Máquina de Turing**. Esta é definida por uma estrutura de controlo com capacidade de ler e escrever símbolos sobre uma fita onde é guardada e processada informação. Turing também provou a equivalência da sua máquina com o cálculo λ, e portanto, com as restantes formalizações de algoritmo.

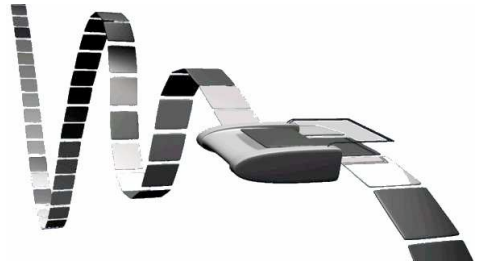


Figura 1.7: Representação artística de uma Máquina de Turing. Uma cabeça de controlo, leitura e escrita desloca-se sobre uma fita potencialmente infinita.

Nesta sua investigação, Turing respondeu à questão deixada aberta por Gödel: *será possível saber, através de um algoritmo, se uma proposição numa dada teoria é demonstrável*? A resposta que Turing descobriu foi o golpe final no programa de Hilbert: não, não é possível!

Para responder a esta pergunta, Turing mostrou que a intuição da época, ao sugerir que problemas progressivamente mais complexos neces-

sitariam de máquinas cada vez mais complexas, estava errada. Primeiro criou uma máquina especial, que designou **Máquina Universal**, *capaz de simular qualquer máquina de Turing dada* (uma tarefa, só por si, nada trivial). Depois provou que o *entscheidungsproblem* proposto por Hilbert pode ser traduzido num outro problema, o **problema da paragem**:

> *Sejam M uma máquina de Turing e I a informação de um dado problema. Se colocarmos I na fita de M e iniciarmos a sua execução, será que M termina (com a solução do problema), ou continua a trabalhar para sempre?*

Ele demonstrou, por redução ao absurdo, que uma máquina capaz de resolver o problema da paragem, ao executar-se nela própria (isto é, fazendo $I = M$ no enunciado do problema da paragem), em certos casos, implica uma contradição. Logo, nenhuma máquina é capaz de resolver o problema da paragem nem, assim, o *entscheidungsproblem*.

Este resultado encontra uma resposta negativa à questão em aberto deixada por Gödel: os matemáticos têm inevitavelmente de trabalhar em teorias incompletas sem saber, no geral, se existem demonstrações para as suas conjecturas.

Para além destas questões matemáticas, o trabalho de Turing teve um efeito colateral importante. As máquinas universais mostram que é possível construirem-se máquinas muito flexíveis e capazes de resolver problemas de tipos muito diferentes: os computadores, no sentido contemporâneo do termo.

Assim, a Teoria da Computação surge nas décadas anteriores aos primeiros computadores programáveis e deriva do trabalho pioneiro de vários matemáticos, como David Hilbert, Alan Turing, Alonzo Church, Kurt Gödel, Emil Post, Stephen Kleene entre muitos outros. Ao pretenderem formalizar conceitos informais como «algoritmo» ou «computação» para estudar as suas propriedades e limitações, tiveram um sucesso assinalável que abriu as portas para a Informática moderna e para as Ciências da Computação que já tanto transformaram o nosso mundo.

Figura 1.8: Por ordem de leitura: Hilbert, Gödel, Kleene, Post, Turing e Church.

Este texto propõe-se expor, discutir, exemplificar e, se possível, honrar, algum do trabalho desenvolvido por estes ilustres matemáticos.

1.3 Público-Alvo e Âmbito do Livro

Os autores de manuais ou livros técnicos têm uma preocupação decisiva que norteia o que e como se escreve: os leitores para qual o respectivo texto se direcciona. Este livro é sobre computação — não propriamente

uma surpresa dado o título — e o público-alvo é quem pretende aprender sobre o assunto e que tenha já algum conhecimento matemático de base.

Em princípio estes leitores são estudantes e professores de cursos superiores de ciência, de engenharia ou de matemática cujo currículo contém, pelo menos, uma disciplina sobre Teoria da Computação. Isto não significa excluir outros leitores interessados, mas é necessário desde já afirmar que este texto não pretende ser uma obra enciclopédica ou histórica mas um apoio à aprendizagem de *uma determinada forma* de explicar a computação.

Esta forma deriva, no seu essencial, do uso de máquinas de registos e de funções recursivas para mostrar um conjunto de propriedades e teoremas sobre o tema, inspirada principalmente num livro clássico da computação, escrito pelo matemático **Nigel Cutland**: *Computability: An Introduction to Recursive Function Theory*, cujo teor e método tem sido utilizado, há já vários anos, no Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa, tendo o professor José Félix Costa sido um dos pioneiros no seu uso. Isto não significa que o actual texto seja uma tradução ou adaptação deste livro, mas sim que utiliza uma notação próxima e tem o desejo de explicar de forma adequada ao público-alvo, e em português, os mesmos conceitos da Teoria da Computação.

Uma das principais diferenças em relação ao método mais tradicional de expor a computação é o uso da máquina de registos — excluindo assim a máquina de Turing — para explicar a computação do ponto de vista *operacional*, onde a computação é vista como o efeito de um programa sobre registos de memória. Aqui, como veremos, a computação é uma sequência finita ou infinita de estados, desde o seu estado inicial (definido pelos argumentos do problema) até a um eventual estado final (onde se encontrará a resposta). Este será o tema central do segundo capítulo.

Também o uso de funções para descrever computação não é comum, apesar do sucesso do cálculo λ, inicialmente concebido por Alonzo Church e que inspirou várias linguagens de programação funcionais, como o LISP, o ML ou o Haskell. Iremos usar o formalismo aperfeiçoado por Stephen Kleene e designado aqui por *funções parciais recursivas*, que

proporcionam um ponto de vista *axiomático* da computação. Este será o tema central do terceiro capítulo.

Estas duas visões, à primeira vista radicalmente diferentes, são na verdade complementares (até equivalentes, como veremos adiante) e essenciais para proporcionar uma perspectiva abrangente e fundamentada das questões que esta área investiga e tenta resolver.

Entraremos igualmente nas fronteiras da computabilidade e veremos as limitações teóricas e efectivas da computação. Este assunto será o objecto dos dois últimos capítulos, onde usaremos e ampliaremos os formalismos até ali estudados para perceber que algumas das questões, alguns dos problemas que um Informático ou um Engenheiro de Sistemas se poderá ver defrontado durante a sua vida profissional, não têm resolução, pela inexistência dos recursos necessários ou mesmo pelas próprias limitações inerentes às noções de algoritmo e de computação. Saber reconhecer estas limitações, só por si, justificaria a existência e pertinência desta área do conhecimento. Ao leitor interessado recomendamos a leitura do livro *Computers and Intractability*, de **Michael Garey** e **David Johnson**.

Para ajudar a que este texto seja o mais auto-contido possível, na secção final de anexos revêem-se as noções matemáticas necessárias à adequada compreensão dos restantes capítulos.

1.4 Convenções

Neste texto usamos regularmente uma série de conceitos e pretendemos que a leitura seja tão clara quanto possível. Para esse efeito usamos algumas convenções sobre a notação ilustradas a seguir.

variáveis $x, P, f_1, \varphi,$ *etc.*;

vectores $\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{f},$ *etc.*;

funções designadas $\mathsf{soma}, \mathsf{monus}, \mathsf{rm},$ *etc.*;

predicados designados $\mathsf{par}, \mathsf{primo}, \mathsf{maior},$ *etc.*;

conjuntos designados $\mathcal{R}, \mathcal{C}, \mathcal{PR}$, *etc.*;

axiomas e regras para funções $\mathsf{Z}, \mathsf{S}, \mathsf{C}$, *etc.*;

instruções `Z`, `S`, `J`, *etc.*;

programas `Soma`, `Monus`, `Rm`,*etc.*;

nomes tradicionais certos conceitos são denotados de certa forma por força da tradição; por exemplo `cod` codifica um programa num número natural, **P** representa uma certa classe de linguagens, $\mathbb{N}$ o conjunto dos números naturais, *etc.* Nestes casos deixamos que o peso da história quebre as convenções anteriores.

Sempre que introduzirmos um conceito, ou dele fizermos um uso importante, este aparecerá a **negrito** e com uma entrada no índice remissivo. O ênfase em fragmentos de texto é dado *desta forma*. No entanto é necessário acautelar no enunciado de definições ou de teoremas, pois aí *o texto corre em itálico e* o ênfase *destaca-se ao contrário*.

1.5 Agradecimentos

Gostaríamos de agradecer aos professores José Félix Costa, Carlos Lourenço e Augusto Franco de Oliveira pelo excelente trabalho de docência, de organização e de produção pedagógica que se mostraram essenciais para que este texto tenha chegado à sua forma actual.

O apoio logístico e profissional do Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa e dos Departamentos de Matemática e de Informática da Universidade de Évora, sem os quais nunca teria sido concretizada este trabalho.

Às nossas respectivas famílias por nos terem impedido um trabalho mais rápido, de que resultou mais tempo de reflexão, discussão e cuidado científico.

Cada vida é uma trajectória demasiado complexa na rede de relações que é a sociedade onde se vive e, por isso, agradecer a alguém implica

esquecer outros que, de forma mais indirecta ou subtil, nos deram o empurrão ou nos seguraram no momento certo e na velocidade correcta. A todos esses, um sincero obrigado.

Lisboa, 2009

Francisco Coelho,
João Pedro Neto

Capítulo 2

Computação com Máquinas

Neste capítulo apresentaremos, entre outros conceitos importantes para a Teoria da Computação, a **Máquina de Registos Ilimitados**, ou abreviadamente **URM** (do inglês, *Unlimited Register Machine*). Esta é uma máquina abstracta muito simples, capaz de manipular registos de números naturais através de um conjunto reduzido de instruções.

Este modelo de computação, cujos programas são semelhantes aos de um *assembler*, foi introduzido por Shepherdson e Sturgis em 1963[1]. Como outros modelos de computação, formaliza as noções intuitivas de **algoritmo** e de **computação**, proporcionando:

1. *uma linguagem que introduza a sintaxe das regras*: definindo sequências finitas de comandos, cada um escolhido de um conjunto finito de tipos;
2. *um mecanismo de interpretação das regras dessa linguagem*: descrevendo o efeito de cada comando sobre um certo suporte de informação;
3. *um conjunto de regras escritas na linguagem anterior*: os «programas»;

[1] J. C. Shepherdson, H. E. Sturgis, *Computability of Recursive Functions*, Journal of the ACM 10, pp. 217–255, 1963

A URM é um modelo formal, *i.e.*, bem definido, que permite dar validade matemática às conclusões que resultarem do estudo da computação em geral. Pretendemos, assim, uma definição clara do significado de «computação». Outra característica importante da URM é a sua simplicidade, o que facilita as nossas explorações aos problemas desta área.

2.1 O Modelo da URM

Cada URM tem um **programa** e uma **memória**. Esta é uma sequência (potencialmente ilimitada) de **registos**, designados $R_1, R_2, R_3, \ldots$ Cada registo contém um **número natural**[2]. Representamos o valor no registo R_i por r_i. A memória de uma URM pode ser descrita graficamente pelo diagrama

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
r_1	r_2	r_3	r_4	r_5	r_6	$\cdots$

O conteúdo de cada registo pode ser alterado pelas **instruções** que formam os programas da URM (as instruções também são designadas por **comandos** ou **operações**).

Vamos fazer, provisoriamente, uma definição intuitiva das instruções URM. Mais tarde, quando já tivermos acumulado alguma experiência, explorado alguns programas e as respectivas computações, faremos uma definição rigorosa que ligue a sintaxe (isto é, as regras da formação de programas) à semântica (as computações dos programas) nas máquinas URM.

Definição 2.1.1 ((provisória) Programas e computações URM)

*Um **programa** URM é uma lista (finita) numerada de **instruções** URM. Sejam n, m e q inteiros positivos. As possíveis **instruções** de um programa URM são dos tipos seguintes:*

[2] O conjunto dos números naturais é $\mathbb{N} = \{0, 1, 2, \ldots\}$. Ver uma exposição mais completa no anexo A.3, p.248.

- ***Zero***, *escrita abreviadamente* `Z`, *com argumento* n, *coloca no registo* R_n *o valor zero:* $R_n \leftarrow 0$;
- ***Sucessor***, *escrita abreviadamente* `S`, *com argumento* n, *adiciona uma unidade ao valor do registo* R_n*:* $R_n \leftarrow r_n + 1$;
- ***Transfere***, *escrita abreviadamente* `T`, *com argumentos* (n, m), *copia o valor do registo* R_n *para* R_m*:* $R_m \leftarrow r_n$;
- ***Salto***, *escrita abreviadamente* `J` *(do inglês* jump*), com argumentos* (n, m, q), *compara os valores dos registos* R_n *e* R_m*; se estes forem iguais, o programa passa para a* q*-ésima instrução; caso contrário, se* $r_n \neq r_m$*, o programa passa para a instrução seguinte.*

Como é feita uma ***computação*** *por um programa URM?* Digamos que o programa é $[I_1, I_2, \ldots, I_k]$. As instruções são processadas sequencialmente e a computação termina quando não houver mais nenhuma instrução: primeiro I_1, depois I_2, *etc.* até I_k. Em cada instrução a memória é alterada conforme o tipo e argumento da instrução. Há apenas uma excepção à ordem de processamento: as instruções `J` podem «saltar». Supondo que a instrução a processar é $I_a = \mathtt{J}\,(n, m, q)$. Se $r_n \neq r_m$ então a instrução seguinte é I_{a+1}; mas, se $r_n = r_m$, a computação continua a partir da instrução I_q, em vez de continuar em I_{a+1}. Se acontecer que $q > k$, isto é, se o «salto» for para além da última instrução do programa, a computação termina.[3]

[3]No modelo original de 1963 era usado um registo especial R_0, com o índice da próxima instrução a executar (designa-se, em inglês, ***program counter***). Porém, desta forma, não só os diagramas ficam um pouco mais sobrecarregados, como também se está a misturar a memória com o controlo de uma computação. No entanto, podemos usar o *program counter* para ilustrar os efeitos das instruções numa computação. Antes de iniciar um programa, tem-se $R_0 \leftarrow 1$:

- $\mathtt{Z}\,(n) : R_n \leftarrow 0, R_0 \leftarrow r_0 + 1$;
- $\mathtt{S}\,(n) : R_n \leftarrow r_n + 1, R_0 \leftarrow r_0 + 1$;
- $\mathtt{T}\,(n, m) : R_m \leftarrow r_n, R_0 \leftarrow r_0 + 1$;
- $\mathtt{J}\,(n, m, q)$: se $r_n = r_m$ então $R_0 \leftarrow q$ senão $R_0 \leftarrow r_0 + 1$.

Nesta representação o estado da memória seria dado pela sequência $(r_0, r_1, \ldots)$.

Vejamos agora um exemplo da execução do programa

$$[\mathtt{S}\,(2)\,,\mathtt{Z}\,(1)\,,\mathtt{J}\,(1,2,5)\,,\mathtt{T}\,(2,4)]$$

com memória $(4,8,3,1,0,0,\ldots)$:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$	
4	8	3	1	0	0	$\cdots$	Estado inicial
4	9	3	1	0	0	$\cdots$	Após $I_1 : \mathtt{S}\,(2)$
0	9	3	1	0	0	$\cdots$	Após $I_2 : \mathtt{Z}\,(1)$
0	9	3	1	0	0	$\cdots$	Após $I_3 : \mathtt{J}\,(1,2,5)$
0	9	3	9	0	0	$\cdots$	Após $I_4 : \mathtt{T}\,(2,4)$

A instrução $\mathtt{J}\,(1,2,5)$ não produziu qualquer efeito nos registos. Como $r_1 \neq r_2$ a URM passou à próxima instrução do programa. Se r_1 fosse igual a r_2 a URM tentaria passar a uma quinta instrução, porém, como o programa só tem quatro instruções, terminaria de imediato, sem passar pela instrução $I_4 : \mathtt{T}\,(2,4)$.

Em resumo, os programas URM são listas finitas de instruções de apenas quatro tipos, três dos quais (`Z`, `S` e `T`) alteram registos e um (`J`) permite controlar a ordem do processamento do programa, através da comparação dos valores de registos.

Na verdade, as instruções de tipo `T` podem ser definidas à custa das restantes porque cada instrução $\mathtt{T}\,(n,m)$ pode ser substituída pelo esquema de programa

$$[\mathtt{Z}\,(m)\,,\mathtt{J}\,(n,m,5)\,,\mathtt{S}\,(m)\,,\mathtt{J}\,(1,1,2)]\,.$$

Como funciona um programa destes? Inicializa a zero o registo R_m, para onde será transferido o valor r_n, compara os dois registos R_n e R_m e, enquanto não forem iguais, adiciona 1 a R_m. Repare-se no salto incondicional de I_4: a comparação $r_1 = r_1$ é sempre verdadeira, saltando sempre para I_2 e repetindo o ciclo até que fique $r_n = r_m$ (obtendo, desta forma, o efeito final da instrução $\mathtt{T}\,(n,m)$). Apesar de podermos

definir estas instruções como programas que usam apenas os restantes três tipos, consideramos T como um dos tipos das instruções originais.

Na memória de uma URM podemos usar um número ilimitado de registos. Para aplicar um algoritmo e encontrar a sua (eventual) resposta, os recursos empregues (o tempo gasto e a memória usada) são sempre finitos. Como cada programa URM é uma lista finita de instruções $\mathtt{Z}, \mathtt{S}, \mathtt{T}$ e J, o número de referências a registos, nesse programa, é finito e também o seu espaço de trabalho é limitado ao maior registo referido[4]. Cada programa que termine, computado pela URM, usará sempre uma quantidade finita de recursos, por maiores que seja. É isso que a URM proporciona com os seus registos: memória segundo as necessidades.

O mesmo se pode dizer em relação à capacidade de cada registo. Na especificação da URM, cada um tem um número natural de magnitude ilimitada. Daqui surge um problema subtil, pois certos programas poderiam acumular um valor infinito num registo (e.g., $[\mathtt{S}(1), \mathtt{J}(1,1,1)]$). Porém, tal acontece somente após um número infinito de passos elementares, *i.e.*, se o programa não terminar. Por outro lado, num programa que termina foi processado apenas um número finito de instruções (e, em particular, um número finito de instruções S). Assim, o tempo (ou comprimento) de uma computação dá-nos um limite superior finito ao maior número registado no espaço de trabalho por essa computação.

Claro que um computador físico tem memória finita. Os nossos computadores não podem representar qualquer $n \in \mathbb{N}$, mas apenas valores até ao limite das suas memórias (**exercício**: descubra qual é o maior número que pode ser guardado na memória do seu computador). Esta diferença em relação à URM não nos preocupa (muito) pois o nosso interesse é descobrir o que, *em princípio*, se pode calcular se forem reunidos recursos (finitos) *suficientes*.

[4]Isto é verdade porque as instruções URM não usam endereçamento indirecto, *i.e.*, numa instrução como $\mathtt{Z}(n)$, o valor n é fixo e não depende do estado da computação ou do valor de um qualquer registo. Se assim não fosse, seria possível construir programas para alterar infinitos registos de memória. Por exemplo, o programa — não URM — $[\mathtt{S}(1), \mathtt{S}(r_1), \mathtt{J}(1,1,1)]$ altera um número infinito de registos.

Os programas URM, com o seu processamento bem definido e por usarem recursos finitos, podem ser vistos como (descrições formais, abstractas, de) algoritmos.

2.2 Computação segundo a URM

Temos por enquanto uma noção provisória do que são os programas URM e de como funcionam. Vamos agora aprofundar um pouco a forma como as instruções de um programa URM mudam a memória (*i.e.* os registos) e como chegamos à noção de «**computação**». De seguida apresentamos a definição formal de instrução, programa e computação numa URM. Até ao fim do capítulo, se nada mais for dito, «**programa**» significa programa URM e «**computável**» significa URM-computável.

Uma noção importante é o que vamos designar por **estado** de uma URM: a sequência $(r_1, r_2, \ldots)$ de valores dos registos, juntamente com o índice da instrução actual. Quando pretendemos executar um programa P com **argumentos** $(x_1, \ldots, x_n)$ convencionamos que antes do estado inicial os registos da URM têm valor zero e, quando a computação vai começar, inicializam-se os primeiros n registos com os argumentos dados

$$R_i \leftarrow x_i, i = 1, \ldots, n$$

ou seja:

R_1	R_2	$\cdots$	R_n	R_{n+1}	$\cdots$	$\cdots$	$\cdots$
x_1	x_2	$\cdots$	x_n	0	$\cdots$	0	$\cdots$

Com estes valores iniciais nos registos da URM começamos por executar a primeira instrução de P e assim por diante, seguindo a semântica de cada instrução. As instruções, ao serem executadas, alteram o conteúdo da memória. Assim, geramos uma sucessão de estados da memória, em que cada um depende do anterior, *i.e.*, dos valores dos registos $R_1, R_2 \ldots$ e da próxima instrução. A esta sequência — que pode ser finita (para os programa que terminam) ou infinita (para programas como, por exemplo, $[\mathtt{J}\,(1,1,1)]$) — chamamos a **computação** do programa P, com argumentos $(x_1, \ldots, x_n)$.

Podemos agora fazer uma definição rigorosa das principais noções da computação URM.

Definição 2.2.1 (Programas e Computações URM)
*As **instruções** são, para $a, b, c \in \mathbb{N} \setminus \{0\}$:*

$$\mathtt{Z}(a)\,;\quad \mathtt{S}(a)\,;\quad \mathtt{T}(a,b)\,;\quad \mathtt{J}(a,b,c) \tag{2.1}$$

*Um **programa** é uma lista (finita) de instruções*

$$P = [I_1, \ldots, I_m] \tag{2.2}$$

*O **espaço de trabalho** do programa P é o maior índice de registo nas instruções de P*

$$\rho(P) \;=\; \min\{i \mid \forall \mathtt{Z}(a), \mathtt{S}(a), \mathtt{T}(a,b), \mathtt{J}(a,b,c) \in P,\ i \geq a, b\} \tag{2.3}$$

*Um **estado instantâneo** para um programa P é uma sequência (finita) de números naturais*

$$S = (q, r_1, \ldots, r_k) \tag{2.4}$$

em que $k \geq \rho(P)$ e $q \geq 1$;

*Uma **computação** do programa $P = [I_1, \ldots, I_m]$ com argumentos $x_1, \ldots, x_n$ é uma sequência (potencialmente infinita) de estados instantâneos*

$$P(x_1, \ldots, x_n) = (S_0, S_1, \ldots, S_i, S_{i+1}, \ldots) \tag{2.5}$$

definida recursivamente por

1. *o **estado inicial** é*

$$S_0 = (1, x_1, \ldots, x_n, 0, \ldots, 0) \tag{2.6}$$

2. *no estado* $S_i = (q, r_1, \ldots, r_k, \ldots)$, *se* $q > m$ *então* S_i *é um* ***estado terminal*** *e* S_{i+1} *não está definido; caso contrário (se* $q \leq m$*)* S_i *é um* ***estado intermédio*** *e o* ***próximo estado*** *é*

$$S_{i+1} = (q', r'_1, \ldots, r'_{k'}, \ldots)$$

que difere de S_i *apenas onde indicado, quando a instrução* I_q *é*

$$\mathtt{Z}\,(a) \;:\; \begin{cases} r'_a &= 0 \\ q' &= q+1 \end{cases}; \tag{2.7}$$

$$\mathtt{S}\,(a) \;:\; \begin{cases} r'_a &= r_a + 1 \\ q' &= q+1 \end{cases}; \tag{2.8}$$

$$\mathtt{T}\,(a,b) \;:\; \begin{cases} r'_b &= r_a \\ q' &= q+1 \end{cases}; \tag{2.9}$$

$$\mathtt{J}\,(a,b,c) \;:\; \begin{cases} q' = c & \text{se } r_a = r_b \\ q' = q+1 & \text{se } r_a \neq r_b \end{cases} \tag{2.10}$$

Se a computação $P\,(x_1, \ldots, x_n)$ *for finita dizemos que* ***termina****;*

O ***resultado*** *de uma computação que termina é o valor que se encontra no primeiro registo,* r_1*, do estado terminal. Neste caso escreve-se*

$$P\,(x_1, \ldots, x_n) \downarrow y \tag{2.11}$$

para indicar que y *é o resultado da computação* $P\,(x_1, \ldots, x_n)$*.*

Ilustremos, com um exemplo, o cálculo do resultado de uma computação: o programa

$$\mathtt{Soma} \;=\; [\mathtt{J}\,(3,2,5)\,, \mathtt{S}\,(1)\,, \mathtt{S}\,(3)\,, \mathtt{J}\,(1,1,1)] \tag{2.12}$$

com argumentos x, y calcula o valor de $x + y$, deixando-o em R_1. Em particular, na computação de $\mathtt{Soma}\,(4,2)$, temos:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
4	2	0	0	0	0	$\cdots$ Estado inicial
5	2	1	0	0	0	$\cdots$ Após $\mathtt{J}(3,2,5)\,, \mathtt{S}(1)\,, \mathtt{S}(3)$
5	2	1	0	0	0	$\cdots$ Após $\mathtt{J}(1,1,1)$ voltamos a I_1
6	2	2	0	0	0	$\cdots$ Após $\mathtt{J}(3,2,5)\,, \mathtt{S}(1)\,, \mathtt{S}(3)$
6	2	2	0	0	0	$\cdots$ Após $\mathtt{J}(1,1,1)\,, \mathtt{J}(2,3,5)$ terminamos

Esta computação termina após este último $\mathtt{J}(2,3,5)$ já que neste estado temos $r_2 = r_3$, o que implica um salto para uma quinta instrução, que não existe. Assim a computação termina e o resultado pode ser lido em R_1 (o que está certo, $4+2=6$):

$$\mathtt{Soma}\,(4,2) \downarrow 6.$$

Vejamos novo exemplo, agora para a função

$$\mathsf{pred}\,(x) \;=\; \begin{cases} x-1 & \text{se } x>0 \\ 0 & \text{caso contrário} \end{cases} \tag{2.13}$$

que calcula o **predecessor** de um inteiro *positivo* (sendo $\mathsf{pred}\,(0)=0$).

Para encontrarmos o resultado, procuramos manter o seguinte **estado típico**[5]:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	$k-1$	k	0	0	0	$\cdots$

Um programa URM para calcular esta função é dado por

$$\mathtt{Pred} \;=\; \left[\begin{array}{l|l|l} \mathtt{J}(1,4,9) & \mathtt{J}(1,3,7) & \\ \mathtt{S}(3) & \mathtt{S}(2) & \mathtt{T}(2,1) \\ & \mathtt{S}(3) & \\ & \mathtt{J}(1,1,3) & \end{array}\right] \tag{2.14}$$

Calculemos um exemplo, com o argumento 3:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$	
3	0	0	0	0	0	$\cdots$	Estado inicial
3	1	2	0	0	0	$\cdots$	Após $\mathtt{J}(1,4,9)\,,\mathtt{S}(3)\,,\mathtt{J}(1,3,7)\,,\mathtt{S}(2)\,,\mathtt{S}(3)$
3	2	3	0	0	0	$\cdots$	Após $\mathtt{J}(1,1,3)\,,\mathtt{J}(1,3,7)\,,\mathtt{S}(2)\,,\mathtt{S}(3)$
2	2	3	0	0	0	$\cdots$	Após $\mathtt{J}(1,1,3)\,,\mathtt{J}(1,3,7)\,,\mathtt{T}(2,1)$

[5]Um **estado típico** é uma parte da memória cujos valores, em determinados momentos da computação (como no início ou fim de um ciclo), mantêm uma certa relação numérica.

Após a instrução $\mathtt{J}(1,1,3)$, a URM passa para $\mathtt{J}(1,3,7)$ e, a seguir, dado que $r_1 = r_3$, salta para a sétima instrução, onde o valor de R_2 é transferido para R_1:

$$\mathtt{Pred}(3) \downarrow 2.$$

Como funciona este algoritmo? Existe um **ciclo** neste programa, com uma **condição de paragem** (na instrução I_3) e um **corpo** que define um conjunto de acções que aproximam o estado da memória do resultado desejado (as instruções I_4 e I_5). O programa usa os registos R_2 e R_3, incrementado-os a cada iteração desse ciclo. Devemos também reparar que, no início do programa, o registo R_3 está um valor adiantado em relação a R_2. Assim, quando r_3 for igual a r_1, em R_2 estará o valor anterior ao argumento inicial, *i.e.*, o resultado desejado. Por fim, para o resultado ficar em R_1, transfere-se para aí, na última instrução, o valor de R_2.

Vejamos outro exemplo. A função

$$\mathsf{monus}(x,y) = \begin{cases} x - y & \text{se } x > y \\ 0 & \text{caso contrário} \end{cases} \tag{2.15}$$

calcula a **subtracção natural** (abreviadamente, $x \ominus y$) e funciona como a subtracção $x - y$ excepto se $x \leq y$ (quando resultado da subtracção normal é negativo), $\mathsf{monus}(x,y) = x \ominus y = 0$.

Para esta função começamos por determinar qual é o maior dos dois argumentos. Na resolução seguinte usamos um contador em R_4 que cresce até igualar um dos argumentos. Nesse estado o valor de R_4 será, também, o menor dos dois argumentos.

Para este passo usamos o seguinte estado típico:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	y	0	k	0	0	$\cdots$

De seguida precisamos de considerar duas possibilidades: ou $x \leq y$ (*i.e.* $r_4 = r_1$) e então $\mathsf{monus}(x,y) = 0$; ou, pelo contrário, $x > y$ (*i.e.*

$r_4 = r_2$) e podemos calcular a diferença $x - y$ mantendo o estado típico

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	y	k	$y+k$	0	0	$\cdots$

Aqui, quando $r_4 = r_1$ temos $y + k = x$, ou seja, $k = x - y$. Neste estado típico o valor k está em R_3. Em ambas as possibilidades que considerámos ($x \leq y$ e $x > y$) o resultado de $\mathsf{monus}\,(x, y)$ está no registo R_3. Podemos então juntar os dois passos num programa URM:

$$\texttt{Monus} \;=\; \left[\begin{array}{l|l|l} 1 : \mathtt{J}\,(1,4,9) & 5 : \mathtt{J}\,(1,4,9) & \\ 2 : \mathtt{J}\,(2,4,5) & 6 : \mathtt{S}\,(3) & \\ 3 : \mathtt{S}\,(4) & 7 : \mathtt{S}\,(4) & 9 : \mathtt{T}\,(3,1) \\ 4 : \mathtt{J}\,(1,1,1) & 8 : \mathtt{J}\,(1,1,5) & \end{array}\right] \tag{2.16}$$

As primeiras quatro instruções encontram o menor dos argumentos. Se for x, $\mathtt{J}\,(1,4,9)$ salta para a última instrução, colocando zero em R_1; se o menor argumento for y, $\mathtt{J}\,(2,4,5)$ passa para o cálculo da diferença dos argumentos, no ciclo das instruções $I_5, \ldots, I_8$.

O que podemos concluir com estes exemplos? As funções $\mathsf{soma}\,(x, y)$, $\mathsf{pred}\,(x)$ e $\mathsf{monus}\,(x, y)$ são calculadas pela URM pois existem programas que, quando executados com os argumentos dados, calculam os valores correctos das respectivas funções. Vamos formalizar esta ideia.

Definição 2.2.2 (Convergência)

*Sejam P um programa URM, $a = (a_1, \ldots, a_n) \in \mathbb{N}^n$ e $b \in \mathbb{N}$. Diz-se que a computação de P com argumento a, $P\,(a)$, **converge** para b se $P\,(a) \downarrow b$, isto é, se b é o resultado da computação $P\,(a)$. Para exprimir que a computação $P\,(a)$ é convergente, sem indicar o resultado, escrevemos apenas*

$$P\,(a) \downarrow \tag{2.17}$$

Se a computação $P\,(a)$ não convergir escrevemos

$$P\,(a) \uparrow \tag{2.18}$$

*e dizemos que **diverge**.*

Olhando para os exemplos anteriores, quando escrevemos $\texttt{Soma}\,(4,2)\downarrow$ estamos a dizer que esta computação termina, ignorando o seu resultado. Ou seja, `Soma` é um programa e $\texttt{Soma}\,(4,2)$ — a computação que resulta do programa `Soma` com o argumento $(4,2)$ — termina. Também observámos que um programa URM pode não terminar. O programa

$$\texttt{Omega} \;=\; [\texttt{J}\,(1,1,1)] \tag{2.19}$$

é o caso mais simples de um programa que não termina: para qualquer $x \in \mathbb{N}, \texttt{Omega}\,(x)\uparrow$.

Agora podemos ligar o cálculo dos valores de funções aos resultados das computações.

Definição 2.2.3 (Função computável)
*Dizemos que a função $f : \mathbb{N}^n \to \mathbb{N}$ é **computável** se existir um programa P que, com argumento $(a_1, \ldots, a_n)$, tenha resultado $f\,(a_1, \ldots, a_n)$,* i.e.*, para cada $(a_1, \ldots, a_n)$ no domínio de f,*

$$P\,(a_1, \ldots, a_n)\downarrow f\,(a_1, \ldots, a_n) \tag{2.20}$$

e se $(a_1, \ldots, a_n)$ não pertencer ao domínio de f então

$$P\,(a_1, \ldots, a_n)\uparrow . \tag{2.21}$$

*Neste caso dizemos que o programa P **computa** a função f.*

Pelos exemplos anteriores, as funções soma, pred e monus são computáveis[6]. Nesta definição é ainda necessário algum cuidado com os argumentos que não estão no domínio da função. Por exemplo, podemos fazer um programa `QuaseDividir` de forma que, por exemplo,

[6]Como referido no início da secção, dizer «f é computável» significa, neste contexto, dizer que f é computável por uma URM ou f é URM-computável. Há livros onde se usa o termo função algorítmica para descrever qualquer função cujo processo de cálculo é descrito por um algoritmo (numa qualquer linguagem sem ambiguidade). Dizer que as funções algorítmicas são exactamente as funções computáveis é uma tese razoável (a investigação e a experiência confirmam-no) mas não é um teorema que possa ser demonstrado. Este assunto será tratado na secção 2.6.

$\texttt{QuaseDividir}\,(x,y)\downarrow \frac{x}{y}$, se $y \neq 0$ e $\texttt{QuaseDividir}\,(x,0)\downarrow 0$. Ora este programa *não* computa a função $\mathsf{quociente}\,(x,y) = \frac{x}{y}$ porque, por exemplo $(1,0)$ não está no domínio de quociente mas $\texttt{QuaseDividir}\,(1,0)\downarrow 0$.

Uma função computável pode ser calculada por diversos programas URM. Na verdade, existem infinitos para cada função computável. Basta verificar que na seguinte sequência infinita de programas cada um calcula a função soma:

$$\begin{aligned}
\texttt{Soma}_0 &= [\texttt{J}\,(3,2,5)\,,\texttt{S}\,(1)\,,\texttt{S}\,(3)\,,\texttt{J}\,(1,1,1)] \\
\texttt{Soma}_1 &= [\texttt{J}\,(3,2,5)\,,\texttt{S}\,(1)\,,\texttt{S}\,(3)\,,\texttt{J}\,(1,1,1)\,,\texttt{Z}\,(2)] \\
\texttt{Soma}_2 &= [\texttt{J}\,(3,2,5)\,,\texttt{S}\,(1)\,,\texttt{S}\,(3)\,,\texttt{J}\,(1,1,1)\,,\texttt{Z}\,(2)\,,\texttt{Z}\,(2)] \\
\texttt{Soma}_3 &= [\texttt{J}\,(3,2,5)\,,\texttt{S}\,(1)\,,\texttt{S}\,(3)\,,\texttt{J}\,(1,1,1)\,,\texttt{Z}\,(2)\,,\texttt{Z}\,(2)\,,\texttt{Z}\,(2)] \\
&\vdots
\end{aligned}$$

Ou seja, $\texttt{Soma}_i$ calcula a soma dos argumentos mas, antes de terminar, executa i instruções $\texttt{Z}\,(2)$, que em nada alteram o resultado final. De forma semelhante, é possível encontrar infinitas variações para calcular qualquer função computável[7].

Embora existam infinitos programas para calcular uma dada função computável, para cada programa P e cada $n \in \mathbb{N}$ existe uma única função de aridade n calculada por P, representada por

$$\Phi_P^{(n)}. \tag{2.22}$$

Isto porque é preciso ter em conta o número de argumentos dados a um programa. É que o domínio faz parte da definição de função (ver o anexo A.1, p.237). Por exemplo, as funções $f\,(x) = x$ e $g\,(x,y) = x$ são diferentes. Assim, cada programa P calcula várias funções diferentes, conforme o número de argumentos que estamos a considerar. Para explicitar a

[7]Nem todos os programas que calculam uma função têm de ser construídos com sufixos de instruções inúteis. Na resolução de problemas é comum existirem algoritmos alternativos para encontrar as soluções correctas.

aridade de uma função é costume usar-se um expoente entre parêntesis. Por exemplo $\mathsf{soma}^{(2)}$, ou $\mathsf{pred}^{(1)}$. As funções computadas pelo programa P, com a indicação da aridade são

$$\Phi_P^{(0)}, \Phi_P^{(1)}, \Phi_P^{(2)}, \Phi_P^{(3)}, \ldots$$

A relação entre programas e os seus resultados assenta na seguinte relação

Definição 2.2.4 (Equivalência de programas)
Dois programas URM são ***equivalentes*** *se, para qualquer argumento, as computações dos programas convergem ambas para o mesmo valor ou divergem ambas. Isto é, os programas URM P e Q são equivalentes, e escreve-se*

$$P \equiv Q \tag{2.23}$$

se, para cada $a \in \mathbb{N}^n$, ou

$$\exists y : P(a) \downarrow y \wedge Q(a) \downarrow y$$

ou

$$P(a) \uparrow \wedge Q(a) \uparrow .$$

Há programas que, com certos argumentos, têm computações convergentes mas, com outros argumentos, as computações divergem. Por exemplo, o programa

$$\texttt{Parcial} = [\mathtt{S}(1), \mathtt{J}(1,2,4), \mathtt{J}(1,1,1)] \tag{2.24}$$

com argumento (x, y) só converge quando $x < y$: enquanto que a computação $\texttt{Parcial}(2,4)$ converge,

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$	
2	4	0	0	0	0	$\cdots$	Estado inicial
3	4	0	0	0	0	$\cdots$	Após $\mathtt{S}(1), \mathtt{J}(1,2,4)\, \mathtt{J}(1,1,1)$
4	4	0	0	0	0	$\cdots$	Após $\mathtt{S}(1), \mathtt{J}(1,2,4)$ termina

já para a computação $\texttt{Parcial}\,(5,4)$ temos

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$	
5	4	0	0	0	0	$\cdots$	Estado inicial
6	4	0	0	0	0	$\cdots$	Após $\texttt{S}\,(1)\,,\texttt{J}\,(1,2,4)\,\texttt{J}\,(1,1,1)$
7	4	0	0	0	0	$\cdots$	Após $\texttt{S}\,(1)\,,\texttt{J}\,(1,2,4)\,\texttt{J}\,(1,1,1)$

e assim sucessivamente *ad infinitum*. Em resumo,

$$\texttt{Parcial}\,(2,4)\downarrow 4 \quad \text{mas} \quad \texttt{Parcial}\,(5,4)\uparrow .$$

Este é um exemplo onde o argumento determina uma computação convergente (finita) ou divergente (infinita).

Quando um programa tem este comportamento dizemos que as funções que calcula são **parciais**, *i.e.*, o domínio não coincide com o conjunto de partida:

Definição 2.2.5 (Função total, parcial)
*Seja $\Phi_P^{(n)} : \mathbb{N}^n \to \mathbb{N}$ a função de aridade n computada pelo programa P. Dizemos que esta função é **total** se o seu domínio coincidir com $\mathbb{N}^n$:*

$$\mathsf{dom}\left(\Phi_P^{(n)}\right) = \{a \in \mathbb{N}^n \mid P\,(a)\downarrow\} = \mathbb{N}^n.$$

*Uma função que não é total diz-se **parcial**. Para os elementos fora do domínio dizemos que a função está **indefinida** e, se $a \notin \mathsf{dom} f$, escrevemos $f\,(a) = \perp$.*

Por exemplo, $\frac{1}{0} = \perp$ e, também, $\Phi^{(2)}_{\texttt{Parcial}}\,(5,4) = \perp$. Um programa converge em qualquer argumento no domínio da função que calcula e diverge nos restantes valores. Há, assim, uma relação clara entre a convergência/divergência das computações de programas e a definição/indefinição nos valores das respectivas funções.

Predicados decidíveis

Intuitivamente, um **predicado** distingue, num certo domínio, os elementos que verificam uma certa propriedade daqueles que não a verificam (*cf.* A.2, p.246). Se φ for um predicado e a um elemento do domínio de φ, escrevemos

$$\varphi\,(a)$$

para dizer que a verifica a propriedade (representada por) φ. Um pouco incorrectamente[8], é costume escrever-se «$\varphi\,(a) =$ verdade» ou «$\varphi\,(a) = 1$» com o mesmo significado.

Definição 2.2.6 (Função característica)
Seja φ um predicado num domínio D. A ***função característica*** *de φ, é a função total*

$$\begin{array}{rccl} \mathsf{c}_\varphi: & D & \to & \{0,1\} \\ & a & \mapsto & \begin{cases} 1 & \text{se } \varphi\,(a) \\ 0 & \text{caso contrário} \end{cases} \end{array} \tag{2.25}$$

Por exemplo, o predicado «x é par?», tem a função característica

$$\mathsf{c}_{\mathsf{par}}\,(x) = \begin{cases} 1 & \text{se } x \text{ é par} \\ 0 & \text{se } x \text{ é ímpar} \end{cases}$$

Com as funções características podemos dar uma forma às noções de verdade e falso: «$\mathsf{c}_{\mathsf{par}}\,(2) = 1$» afirma «é verdade que 2 é par» enquanto $\mathsf{c}_{\mathsf{par}}\,(17) = 0$ diz «é falso que 17 é par». Claro que estes enunciados são normalmente escritos de forma mais compacta: «2 é par» e «17 é ímpar».

Na definição de função característica dissemos que «φ [é] um predicado num domínio D», sem nenhuma condição sobre o tipo de elementos que estão nesse domínio. Por enquanto a noção de computação

[8]Porque estamos a confundir a expressão de uma proposição φ com o seu valor verdade ou falso. Se confundirmos «de noite todos os gatos são pardos $= 1$» também podemos confundir «Lisboa é a capital de Portugal $= 1$» e concluir que «de noite todos os gatos são pardos = Lisboa é a capital de Portugal»...

está limitada a argumentos naturais, pelo que terá de ser $D \subseteq \mathbb{N}^n$. Na próxima secção vamos alargar o âmbito da computação a outros tipos de argumentos, mas por enquanto fica entendido que estamos a falar de predicados sobre números naturais.

É através das funções características que podemos fazer o estudo computacional da lógica (enquanto disciplina que estuda a noção de «verdade»): se a função característica de um predicado φ for computada por um programa, é-nos possível determinar computacionalmente, para cada valor a no domínio de φ, se se verifica $\varphi(a)$ ou não.

Definição 2.2.7 (Predicado decidível)
*Um predicado φ diz-se **decidível** se a sua função característica, c_φ, for computável. Caso contrário diz-se que o predicado φ é **indecidível**.*

No contexto das URM teríamos de dizer que um predicado é URM-decidível se a sua função característica for URM-computável. A definição de «predicado decidível» sugere que nem todos os predicados são decidíveis. Este assunto será tratado mais tarde, no capítulo 4.

Por enquanto podemos estudar alguns predicados decidíveis. Por exemplo, *será que «x é par?» é decidível?* Para respondermos pela positiva precisamos de encontrar um programa que compute a função $\mathsf{c}_{\mathsf{par}}$. Um programa possível é

$$\mathtt{Par} = \left[\begin{array}{l|l|l} 1 : \mathtt{J}(1,2,6) & & \\ 2 : \mathtt{S}(2) & 6 : \mathtt{Z}(1) & \\ 3 : \mathtt{J}(1,2,9) & 7 : \mathtt{S}(1) & 9 : \mathtt{Z}(1) \\ 4 : \mathtt{S}(2) & 8 : \mathtt{J}(1,1,10) & \\ 5 : \mathtt{J}(1,1,1) & & \end{array}\right]. \tag{2.26}$$

Vejamos como faz a computação do valor de $\mathsf{c}_{\mathsf{par}}(3)$:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$	
3	0	0	0	0	0	$\cdots$	Estado inicial
3	2	0	0	0	0	$\cdots$	Após $\mathtt{J}(1,2,6), \mathtt{S}(2), \mathtt{J}(1,2,9), \mathtt{S}(2)$
3	3	0	0	0	0	$\cdots$	Após $\mathtt{J}(1,1,1), \mathtt{J}(1,2,6), \mathtt{S}(2), \mathtt{J}(1,2,9)$
0	3	0	0	0	0	$\cdots$	Após $\mathtt{Z}(1)$ termina

O programa usa um contador em R_2 que, a cada passo, compara o seu valor com o argumento inicial em R_1. A cada par de incrementos, o programa volta ao início. Assim, há duas possibilidades para terminar o programa: (a) na instrução $1 : \mathtt{J}(1,2,6)$ onde, se for $r_1 = r_2$, o argumento é par, ou (b) na instrução $3 : \mathtt{J}(1,2,9)$ onde o argumento é ímpar. Cada um dos saltos desloca o controlo do programa para copiar (respectivamente) 0 ou 1 em R_1.

Parte do interesse da lógica consiste no uso das **operações lógicas**: a conjunção $\wedge$, a disjunção $\vee$ e a negação $\neg$. Do ponto de vista da computação, interessa-nos saber responder a questões do tipo: *Se φ e ψ forem predicados decidíveis, será que, por exemplo, o predicado $\varphi \wedge \psi$ também é decidível?* O próximo teorema responde a essas questões.

Teorema 2.2.8
Sejam φ e ψ predicados decidíveis no mesmo domínio. Então também são decidíveis os predicados $\varphi \wedge \psi, \varphi \vee \psi$ e $\neg\varphi$.

Demonstração. Sendo φ e ψ decidíveis, as funções c_φ e c_ψ são computáveis. Ora, como podemos definir

$$\mathsf{c}_{\varphi\wedge\psi} = \mathsf{c}_\varphi \times \mathsf{c}_\psi \tag{2.27}$$
$$\mathsf{c}_{\varphi\wedge\psi} = \mathsf{max}\{\mathsf{c}_\varphi, \mathsf{c}_\psi\} \tag{2.28}$$
$$\mathsf{c}_{\neg\varphi} = \mathsf{monus}(1, \mathsf{c}_\varphi) \tag{2.29}$$

e, como $x \times y$ (na página 70), $\mathsf{max}\{x,y\}$ (p. 70) e $\mathsf{monus}(1,x)$ (p. 69) são funções computáveis, concluímos que as funções características de $\varphi \wedge \psi, \varphi \vee \psi$ e $\neg\varphi$ também são computáveis. □

Computação noutros domínios

Até agora os argumentos das funções calculadas por programas URM são números naturais. *Como lidar com números negativos, fracções ou com tipos de dados como vectores, tuplos e listas?* Em vez de construirmos uma máquina mais complexa para processar valores de um tipo T mais

geral, basta-nos demonstrar que se podem representar valores de T com números naturais e vice-versa. Temos um exemplo deste processo quando associamos os valores 1 a «verdade» e 0 a «falso».

Porque é que isto funciona? Porque se conseguirmos associar a um valor do tipo A um número natural; se tivermos um programa URM adequado (que funcione com argumentos naturais); e se os resultados desse programa puderem ser associados a valores do tipo B, então podemos simular funções de A para B, mesmo que estes conjuntos não sejam de números naturais. Ou seja, se $f : A \to B$ for uma função entre dois conjuntos quaisquer e existirem codificações $\alpha : A \to \mathbb{N}$ e $\beta : B \to \mathbb{N}$, podemos tentar fazer um programa URM F que «simule» o cálculo $f(x) = y$, seguindo os seguintes passos:

1. usamos a codificação α para representar $x \in A$ por $n_x = \alpha(x) \in \mathbb{N}$;
2. computamos o resultado $m = F(n_x) \in \mathbb{N}$;
3. usamos a descodificação β^{-1} para encontrar $y = \beta^{-1}(m) \in B$.

Isto é,

$$f(x) = \beta^{-1}(F(\alpha(x))).$$

Este processo pode ser representado pelo diagrama

$$\begin{array}{ccc} A & \longrightarrow & B \\ \hline x & \overset{f}{\mapsto} & y \\ \alpha \downarrow & & \uparrow \beta^{-1} \\ n_x & \overset{F}{\mapsto} & m \\ \hline \mathbb{N} & \longrightarrow & \mathbb{N} \end{array}.$$

É importante termos presente que os passos α e β^{-1} de codificação e descodificação não são efectuados pelas URM: estas apenas processam um argumento natural (que é o código de um elemento de outro domínio) e, no fim da computação, o resultado obtido será ainda um número natural (de novo, o código de um elemento de outro domínio).

Porque é que a codificação é importante? Porque agora concluímos que, na definição de programa URM, basta-nos pensar em registos com números naturais, ignorando outros tipos de dados. Assim os programas URM mantêm-se simples, mas potencialmente gerais. As conclusões que retirarmos das computações com números naturais podem ser estendidas a outros domínios, desde que existam codificações apropriadas.

Mas como se processam estas codificações? Começamos por definir exactamente o que necessitamos

Definição 2.2.9 (Codificação)
Uma ***codificação*** *de um domínio T é uma função injectiva, α, com assinatura $T \to \mathbb{N}$. Para um dado $x \in T$, designa-se* ***código*** *de x ao valor $\alpha(x)$.*

Porque é que as codificações têm de ser funções injectivas? Numa função **injectiva**, f, dois valores x e y diferentes resultam em valores $f(x)$ e $f(y)$ diferentes (*cf.* A.1, p.237). Isto é necessário para que a cada valor de T seja associado a um natural que o distinga. Sem esta condição haveria ambiguidade sobre o valor representado por um dado código. Por exemplo, se $\alpha(-42) = 7$ e $\alpha(-39) = 7$, o número 7 representa -42 ou -39? Não sabemos! Mas este problema não acontece com funções injectivas. Se α fosse injectiva teríamos a garantia que $\alpha(-42) \neq \alpha(-39)$. Portanto a injectividade garante que podemos encontrar uma função inversa[9] de α, $\alpha^{-1} : \mathbb{N} \to T$ para fazer a descodificação.

Supondo que temos uma função $f : A \to B$ e codificações $\alpha : A \to \mathbb{N}$ e $\beta : B \to \mathbb{N}$. *Queremos agora definir o programa F para simular f.* Ora este programa computa uma determinada função, digamos f^*, relacionada mas distinta de f. Sabemos que f^* tem assinatura $\mathbb{N} \to \mathbb{N}$ mas precisamos de descobrir um pouco mais para definirmos o programa F. O que falta saber pode ser descoberto invertendo as codificações no

[9]Neste caso, a função inversa esquerda, *i.e.*, $\alpha^{-1} \circ \alpha = \mathsf{id}_T$

diagrama anterior:

$$\begin{array}{ccc} A & \longrightarrow & B \\ \hline\hline x & \overset{f}{\mapsto} & y \\ \alpha^{-1}\uparrow & & \downarrow\beta \\ n_x & \overset{f^*}{\mapsto} & m \\ \hline\hline \mathbb{N} & \longrightarrow & \mathbb{N} \end{array} .$$

Isto é, pretendemos que o programa F compute a função

$$f^* \;=\; \beta \circ f \circ \alpha^{-1}.$$

de assinatura $\mathbb{N} \to \mathbb{N}$.

Portanto, se a função f^* obtida por esta composição for computável, sabemos que existe um programa F que a computa. Este programa simula f. *Neste sentido técnico podemos alargar o âmbito de «função computável» a domínios além dos números naturais.*

Definição 2.2.10 (Função computável (generalização))
Sejam α e β codificações de A e B respectivamente. A função $f : A \to B$ é ***computável*** *se*

$$f^* \;=\; \beta \circ f \circ \alpha^{-1} \tag{2.30}$$

for computável.

Vejamos alguns exemplos. Para o domínio $\mathbb{Z}$ podemos usar a seguinte codificação:

$$\begin{array}{rccl} \alpha_{\mathbb{Z}}: & \mathbb{Z} & \to & \mathbb{N} \\ & x & \mapsto & \left\{ \begin{array}{ll} 2x & \text{se } x \geq 0 \\ -2x-1 & \text{se } x < 0 \end{array} \right. \end{array} \tag{2.31}$$

Nesta codificação $\alpha_{\mathbb{Z}}(-3) = 5$ e $\alpha_{\mathbb{Z}}(5) = 10$. A função $\alpha_{\mathbb{Z}}$ «distribui» alternadamente os valores positivos e negativos de $\mathbb{Z}$ por $\mathbb{N}$, associando os negativos aos ímpares e os positivos aos pares:

$$\begin{array}{ccccccccccccc} 0 & -1 & 1 & -2 & 2 & -3 & 3 & -4 & 4 & -5 & 5 & & \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \cdots & \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & & \end{array}$$

Como todos os números naturais são imagem de um número inteiro, a função $\alpha_{\mathbb{Z}}$ é também sobrejectiva (e, assim, bijectiva). A inversa, $\alpha_{\mathbb{Z}}^{-1}$, é dada por:

$$\begin{array}{rccl} \alpha_{\mathbb{Z}}^{-1}: & \mathbb{N} & \to & \mathbb{Z} \\ & n & \mapsto & \begin{cases} \frac{n}{2} & \text{se } n \text{ é par} \\ -\frac{n+1}{2} & \text{se } n \text{ é ímpar} \end{cases} \end{array} \tag{2.32}$$

Vejamos uma aplicação desta codificação. Seja $f(x) = x-1$, com $x \in \mathbb{Z}$; usando a codificação $\alpha_{\mathbb{Z}}$, fica (**exercício**: determine as composições)

$$f^*(x) = \left(\alpha_{\mathbb{Z}} \circ f \circ \alpha_{\mathbb{Z}}^{-1}\right)(x) = \begin{cases} 1 & \text{se } x = 0 \\ x-2 & \text{se } x \text{ é par positivo} \\ x+2 & \text{se } x \text{ é ímpar} \end{cases}$$

A função f^* pode ser computada por um programa apropriado. Deixa-se igualmente como exercício a construção deste programa. Podemos, assim, concluir que

$$\begin{array}{rccl} f: & \mathbb{Z} & \to & \mathbb{Z} \\ & x & \mapsto & x-1 \end{array}$$

é computável.

Além dos inteiros, também outros tipos podem ser codificados. A função seguinte codifica $\mathbb{N}^2$:

$$\begin{array}{rccl} \alpha_2: & \mathbb{N}^2 & \to & \mathbb{N} \\ & (x,y) & \mapsto & 2^{\alpha_{\mathbb{Z}}(x)} 3^{\alpha_{\mathbb{Z}}(y)} \end{array} \tag{2.33}$$

Iterando esta codificação é fácil passar a vectores de maior dimensão. Por exemplo,

$$\alpha_3(x,y,z) = \alpha_2(\alpha_2(x,y),z). \tag{2.34}$$

Apesar destes exemplos é importante ter presente que nem todos os domínios ou tipos de dados podem ser representados pela URM. Por exemplo, não existe uma codificação de $\mathbb{R}$ para $\mathbb{N}$ (ver A.3, p.248 e em A.3, p.261). Voltaremos ao assunto das codificações no decorrer do livro.

2.3 Módulos

Vamos introduzir uma notação que simplifica a escrita de programas URM. A ideia é que, *uma vez feito um programa, este possa ser usado noutro, como se fosse uma instrução.* É para isso que servem os módulos.

Definição 2.3.1 (Módulo)
Um ***módulo*** $P\,[a_1, \ldots, a_n \to b]$ *efectua a computação* $P\,(r_{a_1}, \ldots, r_{a_n})$ *e guarda o resultado no registo* R_b*. Este processo não altera qualquer registo, excepto* R_b*.*

Relembremos o programa que calcula a função binária soma,

$$\texttt{Soma} = [\texttt{J}\,(3,2,5)\,, \texttt{S}\,(1)\,, \texttt{S}\,(3)\,, \texttt{J}\,(1,1,1)]\,.$$

O módulo $\texttt{Soma}\,[a, b \to c]$ designa «computar Soma usando r_a e r_b como argumentos, colocando, após terminar a computação, o resultado em R_c». Por exemplo, temos os seguintes estados da memória, antes e depois do programa passar pelo módulo $\texttt{Soma}\,[3, 1 \to 5]$:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$	
2	3	5	4	10	1	$\cdots$	Antes de $\texttt{Soma}\,[3, 1 \to 5]$
2	3	5	4	7	1	$\cdots$	Depois de $\texttt{Soma}\,[3, 1 \to 5]$

A computação do módulo corresponde ao cálculo da soma $5 + 2$ (pois são estes os valores de r_3 e r_1) colocando o resultado, 7, no registo R_5. O valor anterior de R_5 é apagado, tal como acontece com as instruções T.

Com o uso de módulos, podemos construir programas progressivamente mais complexos, usando uma técnica de «dividir para reinar». Por exemplo, para computar a função produto $(x, y) = x \times y$, usamos o seguinte estado típico:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	y	k	kx	0	0	$\cdots$

O programa que pretendemos fazer define um ciclo onde, em cada passo,

1. actualiza-se o contador do ciclo em R_3;
2. acumula-se o valor x em R_4

Isto leva a que, quando o contador R_3 for igual a y, teremos acumulado em R_4 y cópias de x, ou seja, $x \times y$. No fim da computação teremos:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
xy	y	x	xy	0	0	$\cdots$

Podemos traduzir esta ideia para um programa que usa o módulo `Soma`:

$$\texttt{Produto} \quad = \quad \left[\begin{array}{l} \texttt{J}\,(2,3,5) \\ \texttt{Soma}\,[1,4 \to 4] \\ \texttt{S}\,(3) \\ \texttt{J}\,(1,1,1) \end{array} \middle| \; \texttt{T}\,(4,1) \right] \tag{2.35}$$

Outro exemplo. Para calcular a função

$$f\,(x) = \begin{cases} x + y & \text{se } x \text{ é par} \\ x \times y & \text{se } x \text{ é ímpar} \end{cases}$$

obtemos o programa

$$P = [\texttt{Par}\,[1 \to 3]\,, \texttt{J}\,(3,4,5)\,, \texttt{Soma}\,[1,2 \to 1]\,, \texttt{J}\,(1,1,6)\,, \texttt{Produto}\,[1,2 \to 1]]$$

Ou seja, P verifica, com o módulo `Par`, qual dos ramos da função deve ser usado. Se for o ramo «ímpar» ($r_3 = 0$) o programa usa o módulo `Produto`; senão a computação passa para o módulo `Soma`.

Como funcionam os módulos? A definição da URM só considera quatro tipos de instruções, sem contemplar módulos. Assim, é necessário explicar como é que um programa com módulos pode ser convertido num programa sem módulos, compatível com a definição URM. Será este objectivo que seguiremos no resto desta secção.

Definição 2.3.2 (Programa normalizado)
Dizemos que um programa P é ***normalizado*** *se em cada instrução* $\mathtt{J}(n,m,q)$ *é* $q \leq |P|+1$.

Por outras palavras, um programa é normalizado se as instruções de salto não enviarem o controlo do programa para além do índice a seguir à última instrução. Por exemplo, $[\mathtt{J}(1,2,3), \mathtt{S}(1)]$ é normalizado enquanto que $[\mathtt{J}(1,2,300), \mathtt{S}(1)]$ não é.

A normalização vai facilitar o uso dos módulos, mas precisamos garantir que a sua aplicação não prejudica o funcionamento do programa original.

Lema 2.3.3
Para qualquer programa P, existe um programa P^* normalizado que, para qualquer argumento $\boldsymbol{a} = (a_1, \ldots, a_n)$,

$$P(\boldsymbol{a}) \downarrow b \text{ se e só se } P^*(\boldsymbol{a}) \downarrow b$$

e

$$P(\boldsymbol{a}) \uparrow \text{ se e só se } P^*(\boldsymbol{a}) \uparrow .$$

Demonstração. Para obter P^* basta substituir todas as instruções de P no formato $\mathtt{J}(n,m,q)$ com $q > |P|+1$ por $\mathtt{J}(n,m,|P|+1)$. As restantes instruções são copiadas sem alterações para P^*.

Temos agora de assegurar que as computações feitas por P e por P^* são iguais. Como P e P^* apenas diferem nas instruções $\mathtt{J}(n,m,q)$ quando $q > |P|+1$, basta analisar o que se passa quando uma computação de P chega a uma tal instrução e, nesse estado, $r_n = r_m$. Em todas as outras circunstâncias, a computação efectuada por P e a efectuada por $P*$ coincidem. Ora, pela definição de computação, aquele é um estado terminal para P. Mas também o é para P^*. Portanto, em ambos os programas, a computação termina. Portanto $P(\boldsymbol{a}) \downarrow b$ se, e só se $P^*(\boldsymbol{a}) \downarrow b$.

Também podemos concluir que $P(\boldsymbol{a}) \uparrow$ se e só se $P^*(\boldsymbol{a}) \uparrow$ porque nestas duas computações nunca ocorre um estado terminal; em particular não ocorre o estado acima analisado. □

De acordo com este lema, para qualquer $n > 0$, $\Phi_P^{(n)} = \Phi_{P^*}^{(n)}$ *i.e.*, para qualquer n, os programas P e P^* computam a mesma função de aridade n.

A nossa estratégia para «purificar» um programa com módulos consiste em expandir cada instrução com um módulo pelas instruções que o definem. Claro que fazendo isto é preciso acertar as referências nas instruções originais, tanto do programa como do módulo. Além disso também precisamos de assegurar que as computações feitas pelo módulo não vão interferir com os registos do programa.

Portanto, queremos saber *como expandir um módulo no meio de um programa*. Podemos começar por supor que no programa há apenas um módulo, dado que expandimos um módulo de cada vez. Além disso também assumimos que tanto o programa como o módulo estão normalizados. Com esta condição não perdemos generalidade, pois, graças ao lema 2.3.3, podemos transformar, sem prejuízo das computações, um programa não normalizado num programa normalizado.

Um módulo M, sendo um programa, é uma lista finita de instruções. Isto significa que há um número máximo, $\rho(M)$, de registos usados em M. Estes vão desde R_1 até $R_{\rho(M)}$, sendo os primeiros n usados para os argumentos. Em resumo, a evocação do módulo $M[i_1, \ldots, i_n \to j]$ passa pelas seguintes fases:

1. transferir $R_{i_1}, \ldots, R_{i_n}$ para $R_1, \ldots, R_n$;
2. pôr a zero os restantes registos, *i.e.*, de R_{i_n+1} até $R_{\rho(P)}$;
3. executar M;
4. transferir o resultado de R_1 para R_j

Mas usando o módulo desta forma, apagamos a informação antes mantida nos primeiros registos, e que pode ser usada pelo programa principal, antes do módulo ser usado. É necessário salvaguardar essa informação.

Descrevemos agora um procedimento geral para a **expansão** de módulos que considera todos estes potenciais problemas. Seja P um programa com módulos. Fixamos em n o número máximo de argumentos

que P pode processar e assumimos que os módulos não contêm módulos[10] e que são programas normalizados. Seja r o maior valor entre os índices de registos usados por P e as aridades dos seus módulos.

O procedimento tem uma fase inicial de preparação:

1. *salvaguardar o argumento $a_1, \ldots, a_n$ num local seguro*, ou seja, transferir $r_1, \ldots, r_n$ para $R_{r+1}, \ldots, R_{r+n}$. Para isso adicionamos n instruções de transferência ao início do programa (por ordem inversa, porque r pode ser menor que n):

$$\begin{array}{c} \mathtt{T}\,(R_n, R_{r+n}) \\ \vdots \\ \mathtt{T}\,(R_1, R_{r+1}) \end{array}$$

2. *acertar os índices de registo no programa* porque este, agora, deve consultar os argumentos em $R_{r+1}, \ldots, R_{r+n}$ e não em $R_1, \ldots, R_n$: todas as instruções têm de ser actualizadas r registos para a direita. Também é necessário *acertar os índices de salto no programa* porque há n novas instruções `T` no início. Ou seja, precisamos de mudar

   - $\mathtt{Z}\,(a)$ para $\mathtt{Z}\,(a + r)$;
   - $\mathtt{S}\,(a)$ para $\mathtt{S}\,(a + r)$;
   - $\mathtt{T}\,(a, b)$ para $\mathtt{T}\,(a + r, b + r)$;
   - $\mathtt{J}\,(a, b, c)$ para $\mathtt{J}\,(a + r, b + r, c + n)$;
   - $M\,[i_1, i_2, \ldots, i_n \to i]$ para $M\,[i_1 + r, i_2 + r, \ldots, i_n + r \to i + r]$.

3. *transferir o resultado para o primeiro registo* para respeitar a convenção sobre a leitura dos resultado: junta-se ao fim do programa original a instrução

$$\mathtt{T}\,(r, 1)\,.$$

[10] Também aqui não há perda de generalidade pois podemos considerar que esses eventuais submódulos já terão sido expandidos por este procedimento.

Depois da fase de preparação deve-se começar por expandir o módulo que ocorre primeiro, depois o segundo e assim sucessivamente... Em cada expansão, substitui-se a instrução onde consta o módulo pelas instruções que o definem. Após esta substituição o programa «principal» ganhou mais algumas linhas, pelo que é necessário tornar a acertar as instruções J. Neste acerto é necessário considerar os saltos do módulo (porque está dentro do programa «principal») como também, no próprio programa «principal», as instruções J que refiram instruções após o módulo.

Façamos o exemplo completo de expansão do programa já conhecido

$$\texttt{Produto} = [\texttt{J}\,(2,3,5)\,,\texttt{Soma}\,[1,4\to 4]\,,\texttt{S}\,(3)\,,\texttt{J}\,(1,1,1)\,,\texttt{T}\,(4,1)]\,.$$

Neste programa está o módulo

$$\texttt{Soma} = [\texttt{J}\,(3,2,5)\,,\texttt{S}\,(1)\,,\texttt{S}\,(3)\,,\texttt{J}\,(1,1,1)]\,.$$

O espaço de trabalho deste programa é $\rho\,(\texttt{Soma}) = 3$. Isto significa que o módulo precisa de três registos para trabalhar. Por isso actualizamos o programa Produto para trabalhar para além destes três registos:

$$\left[\begin{array}{c|c|c} \begin{array}{l} 1:\texttt{T}\,(2,5) \\ 2:\texttt{T}\,(1,4) \end{array} & \begin{array}{l} 3:\texttt{J}\,(5,6,7) \\ 4:\texttt{Soma}\,[4,7\to 7] \\ 5:\texttt{S}\,(6) \\ 6:\texttt{J}\,(4,4,3) \\ 7:\texttt{T}\,(7,4) \end{array} & 8:\texttt{T}\,(4,1) \end{array}\right].$$

As duas primeiras instruções T guardam os argumentos na «zona segura» (*i.e.*, para lá da área de trabalho do módulo). A última instrução T repõe o resultado final em R_1.

Agora já é seguro executar o módulo Soma nos registos iniciais, mas a inserção das instruções do módulo $\texttt{Soma}\,[4,7\to 7]$ tem de ser preparada:

1. *transferir* os (dois) argumentos para os primeiros registos:

$$\begin{array}{c} \texttt{T}\,(4,1) \\ \texttt{T}\,(7,2) \end{array}$$

2. *anular* os restantes registos na área de trabalho:

$$\texttt{Z}\,(3)$$

3. *inserir* as instruções do programa `Soma`, com os saltos corrigidos:

$$\begin{gathered}\texttt{J}\,(\mathbf{3},\mathbf{2},\mathbf{8})\\ \texttt{S}\,(1)\\ \texttt{S}\,(3)\\ \texttt{J}\,(\mathbf{1},\mathbf{1},\mathbf{4})\end{gathered}$$

4. *transferir* o resultado para o registo indicado:

$$\texttt{T}\,(1,7)$$

A preparação do módulo $\texttt{Soma}\,[1,4\to 4]$ consiste, assim, nestas oito instruções. Os saltos `J` foram acertados devido à inclusão das três instruções de inicialização dos registos — $\texttt{T}\,(4,1)$, $\texttt{T}\,(7,2)$ e $\texttt{Z}\,(3)$ — antes das instruções originais.

Agora é necessário substituir o módulo `Soma` no programa `Produto` por este conjunto de instruções:

$$\left[\begin{array}{l|l|l}
\begin{array}{ll}1: & \texttt{T}\,(2,5)\\ 2: & \texttt{T}\,(1,4)\\ \hline 3: & \texttt{J}\,(5,6,7)\end{array} &
\begin{array}{ll}4: & \texttt{T}\,(4,1)\\ 5: & \texttt{T}\,(7,2)\\ 6: & \texttt{Z}\,(3)\\ \hline 7: & \texttt{J}\,(3,2,8)\\ 8: & \texttt{S}\,(1)\\ 9: & \texttt{S}\,(3)\\ 10: & \texttt{J}\,(1,1,4)\\ \hline 11: & \texttt{T}\,(1,7)\end{array} &
\begin{array}{ll}12: & \texttt{S}\,(6)\\ 13: & \texttt{J}\,(4,4,3)\\ 14: & \texttt{T}\,(7,4)\\ \hline 15: & \texttt{T}\,(4,1)\end{array}
\end{array}\right].$$

As instruções `J` do módulo estão agora no meio de um programa. Como há três instruções antes do módulo, somamos 3 aos saltos do módulo:

$$\left[\begin{array}{l|l|l}
\begin{array}{ll} 1: & \mathtt{T}\,(2,5) \\ 2: & \mathtt{T}\,(1,4) \\ \hline 3: & \mathtt{J}\,(5,6,7) \end{array} &
\begin{array}{ll} 4: & \mathtt{T}\,(4,1) \\ 5: & \mathtt{T}\,(7,2) \\ 6: & \mathtt{Z}\,(3) \\ \hline 7: & \mathtt{J}\,(\mathbf{3},\mathbf{2},\mathbf{11}) \\ 8: & \mathtt{S}\,(1) \\ 9: & \mathtt{S}\,(3) \\ 10: & \mathtt{J}\,(\mathbf{1},\mathbf{1},\mathbf{7}) \\ \hline 11: & \mathtt{T}\,(1,7) \end{array} &
\begin{array}{ll} 12: & \mathtt{S}\,(6) \\ 13: & \mathtt{J}\,(4,4,3) \\ 14: & \mathtt{T}\,(7,4) \\ \hline 15: & \mathtt{T}\,(4,1) \end{array}
\end{array}\right].$$

Falta só uma coisa: substituímos uma instrução por oito, portanto as instruções J do programa principal que referem saltos *para depois da expansão* têm de ser acertadas (o salto para a 7.ª instrução, antes de inserirmos as oito instruções, é agora um salto para a 15.ª instrução):

$$\left[\begin{array}{l|l|l}
\begin{array}{ll} 1: & \mathtt{T}\,(2,5) \\ 2: & \mathtt{T}\,(1,4) \\ \hline 3: & \mathtt{J}\,(\mathbf{5},\mathbf{6},\mathbf{15}) \end{array} &
\begin{array}{ll} 4: & \mathtt{T}\,(4,1) \\ 5: & \mathtt{T}\,(7,2) \\ 6: & \mathtt{Z}\,(3) \\ \hline 7: & \mathtt{J}\,(3,2,11) \\ 8: & \mathtt{S}\,(1) \\ 9: & \mathtt{S}\,(3) \\ 10: & \mathtt{J}\,(1,1,7) \\ \hline 11: & \mathtt{T}\,(1,7) \end{array} &
\begin{array}{ll} 12: & \mathtt{S}\,(6) \\ 13: & \mathtt{J}\,(4,4,3) \\ 14: & \mathtt{T}\,(7,4) \\ 15: & \mathtt{T}\,(4,1) \end{array}
\end{array}\right].$$

Por fim, obtemos um programa normalizado e sem módulos da multiplicação.

Este processo define programas com mais instruções do que seriam necessárias, por exemplo, as duas últimas instruções do programa anterior, $\mathtt{T}\,(7,4)$ e $\mathtt{T}\,(4,1)$, podiam ser substituídas por uma só, $\mathtt{T}\,(7,1)$. Este

é um pequeno preço a pagar por um processo automático, implementável num computador, e que nos proporciona, a partir de pseudo-programas com módulos, programas normalizados.

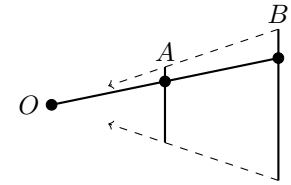
2.4 Enumeração de Programas

Nesta secção demonstramos que o conjunto dos programas URM, *i.e.*, o conjunto composto por todos os programas URM possíveis, designado por $\mathcal{P}$, tem a cardinalidade do conjunto $\mathbb{N}$, *i.e.*, $\mathcal{P}$ é um conjunto enumerável.

Esta afirmação poderá parecer estranha: como é possível existirem tantos programas URM quantos os números naturais? As combinações de instruções URM nos programas (é admissível pensarmos em programas com milhões de instruções) parecem exceder os números naturais. Mas algo semelhante ocorreu, na secção anterior, quando se codificaram domínios aparentemente maiores do que $\mathbb{N}$. Então foi-nos possível encontrar uma correspondência (bijectiva) entre os números inteiros (positivos e negativos) e os números naturais. Apesar destes conjuntos parecerem maiores, estamos a lidar com conjuntos infinitos onde o nosso senso comum nem sempre funciona[11]. Neste caso faremos uma codificação conforme o que foi descrito anteriormente em que o tipo agora a codificar é $\mathcal{P}$.

[11] Galileu foi um dos primeiros a observar as estranhas propriedades dos conjuntos infinitos. No seu trabalho final, *Duas Novas Ciências*, descreveu dois aparentes paradoxos. No primeiro mostrou que haveriam tantos quadrados perfeitos n^2 quantos números inteiros n. No segundo, afirmou a existência de igual número de pontos em segmentos de recta de tamanhos distintos.

Fixando um ponto O e projectando uma recta na direcção dos dois segmentos verticais, cada ponto do segmento pequeno (A) tem uma e uma só correspondência do segmento maior (B). Define-se, assim, uma bijecção entre os dois segmentos.



Codificação de programas

Antes de considerarmos programas URM, mostramos a tarefa mais simples de **codificar** instruções individuais. Por exemplo, para codificar instruções $\mathtt{Z}(n)$, basta usar a codificação

$$\alpha_{\mathtt{Z}}(n) \;=\; n-1 \tag{2.36}$$

que graficamente pode ser representada por

$$\begin{array}{ccccccccccc} \mathtt{Z}(1) & \mathtt{Z}(2) & \mathtt{Z}(3) & \mathtt{Z}(4) & \mathtt{Z}(5) & \mathtt{Z}(6) & \mathtt{Z}(7) & \mathtt{Z}(8) & \mathtt{Z}(9) & \cdots \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \cdots \end{array}$$

A mesma ideia pode aplicar-se ao código das instruções $\mathtt{S}$, *i.e.*:

$$\alpha_{\mathtt{S}}(n) \;=\; n-1 \tag{2.37}$$

a que corresponde

$$\begin{array}{ccccccccccc} \mathtt{S}(1) & \mathtt{S}(2) & \mathtt{S}(3) & \mathtt{S}(4) & \mathtt{S}(5) & \mathtt{S}(6) & \mathtt{S}(7) & \mathtt{S}(8) & \mathtt{S}(9) & \cdots \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \cdots \end{array}$$

Para as instruções $\mathtt{T}$ temos de passar por um passo intermédio, dado estas instruções terem dois argumentos. Para estas instruções vamos usar a função

$$\begin{array}{rccc} \Pi: & \mathbb{N}^2 & \to & \mathbb{N} \\ & (m,n) & \mapsto & 2^n(2m+1)-1 \end{array} \tag{2.38}$$

que define uma bijecção $\mathbb{N}^2 \to \mathbb{N}$, e as suas inversas

$$\begin{array}{rccc} \Pi_1: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & (x+1)_1 \end{array} \tag{2.39}$$

e

$$\begin{array}{rccc} \Pi_2: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & \frac{1}{2}\left(\frac{x+1}{2^{\Pi_1(x)}}-1\right) \end{array} \tag{2.40}$$

que definem a inversa $\Pi_1 \otimes \Pi_2 : \mathbb{N} \to \mathbb{N}^2$ (estas três funções estão explicadas no anexo A.3, p.259). Além disso também vamos precisar da função

$$\begin{array}{rccc} (\cdot)_1: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & (x)_1 \end{array} \tag{2.41}$$

que é o expoente no primeiro primo (*i.e.*, no número 2) na factorização prima de x (verificar no anexo A.3, p.252).

A função Π distribui pares de naturais por $\mathbb{N}$ e com ela definimos a codificação

$$\alpha_{\mathtt{T}}(n,m) \;=\; \Pi(n-1,m-1) \tag{2.42}$$

com distribuição

$$\begin{array}{ccccccccc} \mathtt{T}(1,1) & \mathtt{T}(2,1) & \mathtt{T}(1,2) & \mathtt{T}(3,1) & \mathtt{T}(1,3) & \mathtt{T}(2,2) & \mathtt{T}(1,4) & \mathtt{T}(4,1) & \cdots \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \cdots \end{array}$$

Para codificar as instruções J, que têm três argumentos, usamos duas vezes a função Π:

$$\alpha_{\mathtt{J}}(n,m,q) = \Pi(\Pi(n-1,m-1),q-1) \tag{2.43}$$

o que corresponde a distribuir

$$\begin{array}{cccccccc} \mathtt{J}(1,1,1) & \mathtt{J}(2,1,1) & \mathtt{J}(1,1,2) & \mathtt{J}(1,2,1) & \mathtt{J}(1,1,3) & \mathtt{J}(2,1,2) & \cdots \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ 0 & 1 & 2 & 3 & 4 & 5 & \cdots \end{array}$$

Para descodificar um número natural, digamos x, e obter a correspondente instrução de salto, calculamos $\Pi^{-1}(x) = (\Pi_1(x), \Pi_2(x)) \in \mathbb{N}^2$.

Na segunda componente temos $q - 1 = \Pi_2(x)$ e portanto $q = \Pi_2(x) + 1$. Já na primeira componente (sendo a codificação de um certo par $(n-1, m-1)$) é necessário tornar a usar Π^{-1} para obter os restantes argumentos $n - 1 = \Pi_1(\Pi_1(x))$ e $m - 1 = \Pi_2(\Pi_1(x))$. A instrução codificada pelo natural x é $\mathtt{J}(n, m, q)$.

O próximo passo é codificar uma qualquer instrução URM. Já sabemos codificar para cada tipo de instrução individualmente mas agora pretendemos juntar numa única codificação. Para tal distribuímos as instruções em intervalos de quatro (porque há quatro tipos), começando por Z:

$\mathtt{Z}(1)$				$\mathtt{Z}(2)$			$\cdots$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	
0	1	2	3	4	5	6	$\cdots$

E assim sucessivamente para os restantes tipos de instruções:

- as instruções S:

$\mathtt{Z}(1)$	$\mathtt{S}(1)$			$\mathtt{Z}(2)$	$\mathtt{S}(2)$		$\cdots$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	
0	1	2	3	4	5	6	$\cdots$

- as instruções T:

$\mathtt{Z}(1)$	$\mathtt{S}(1)$	$\mathtt{T}(1,1)$		$\mathtt{Z}(2)$	$\mathtt{S}(2)$	$\mathtt{T}(1,2)$	$\cdots$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	
0	1	2	3	4	5	6	$\cdots$

- as instruções J:

$\mathtt{Z}(1)$	$\mathtt{S}(1)$	$\mathtt{T}(1,1)$	$\mathtt{J}(1,1,1)$	$\mathtt{Z}(2)$	$\mathtt{S}(2)$	$\mathtt{T}(1,2)$	$\cdots$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	
0	1	2	3	4	5	6	$\cdots$

Ou seja, o **código** da instrução I é dado por

$$\beta(I) = \begin{cases} 4\alpha_{\mathtt{Z}}(n) & \text{se } I = \mathtt{Z}(n) \\ 4\alpha_{\mathtt{S}}(n) + 1 & \text{se } I = \mathtt{S}(n) \\ 4\alpha_{\mathtt{T}}(n,m) + 2 & \text{se } I = \mathtt{T}(n,m) \\ 4\alpha_{\mathtt{J}}(n,m,q) + 3 & \text{se } I = \mathtt{J}(n,m,q) \end{cases} \tag{2.44}$$

Por exemplo,

$$\begin{aligned} \beta(\mathtt{S}(3)) &= 4\alpha_{\mathtt{S}}(3) + 1 \\ &= 4(3-1) + 1 \\ &= 9 \end{aligned}$$

e

$$\begin{aligned} \beta(\mathtt{J}(3,1,5)) &= 4\alpha_{\mathtt{J}}(3,1,5) + 3 \\ &= 575. \end{aligned}$$

Tendo a codificação β, das instruções, falta saber como codificar um programa arbitrário $P \in \mathcal{P}$. A **codificação** de programas URM é feita pela função $\mathsf{cod} : \mathcal{P} \to \mathbb{N}$ que, ao programa URP $P = [I_1, \ldots, I_n]$, faz corresponder o número natural

$$\begin{aligned} \mathsf{cod}(P) &= 2^{\beta(I_1)} \\ &+ 2^{1+\beta(I_1)+\beta(I_2)} \\ &+ 2^{2+\beta(I_1)+\beta(I_2)+\beta(I_3)} \\ &\vdots \\ &+ 2^{(n-1)+\beta(I_1)+\cdots+\beta(I_n)} \\ &- 1 \end{aligned} \tag{2.45}$$

Esta codificação transforma o código de cada instrução numa potência de 2. A cada expoente é adicionado um número progressivamente crescente, para evitar sobreposições de duas ou mais instruções $\mathtt{Z}(1)$ consecutivas (cujo código é zero). O somatório acumulado das instruções

também tem como objectivo evitar outros tipos de sobreposição que tornariam cod uma função não injectiva. Por outro lado, a subtracção de uma unidade no final torna a função sobrejectiva e, desta forma, cada número natural é o código de um programa (um aspecto importante para a descodificação). Ao subtrair esta unidade, também o zero corresponde ao código de um programa (nomeadamente fica $0 = \mathsf{cod}([\mathtt{Z}(1)])$).

Por exemplo, para o código do programa $P = [\mathtt{S}(1), \mathtt{T}(3,1), \mathtt{Z}(2)]$, temos de encontrar os códigos de cada instrução:

$$\begin{aligned} b_1 = \beta(I_1) &= \beta(\mathtt{S}(1)) = 1 \\ b_2 = \beta(I_2) &= \beta(\mathtt{T}(3,1)) = 18 \\ b_3 = \beta(I_3) &= \beta(\mathtt{Z}(2)) = 4 \end{aligned}$$

$$\begin{aligned} \mathsf{cod}(\ [\mathtt{S}(1), \mathtt{T}(3,1), \mathtt{Z}(2)]\ ) &= 2^{b_1} + 2^{b_1+b_2+1} + 2^{b_1+b_2+b_3+2} - 1 \\ &= 2^1 + 2^{1+18+1} + 2^{1+18+4+2} - 1 \\ &= 2 + 2^{20} + 2^{25} - 1 \\ &= 34603009 \end{aligned}$$

Definição 2.4.1 *Seja P um programa. O número natural $\mathsf{cod}(P)$ é designado por **número de Gödel** de P. Denotamos por $\mathsf{dec}(m)$, ou por $\mathsf{cod}^{-1}(m)$, o programa cujo número de Gödel é m.*

Usando o exemplo anterior,

$$\mathsf{dec}(34603009) = \mathsf{cod}^{-1}(34603009) = [\mathtt{S}(1), \mathtt{T}(3,1), \mathtt{Z}(2)].$$

A função dec é a inversa de cod e associa um número natural a um programa. Veremos como se processa esta descodificação na secção seguinte. De notar ainda que estas duas funções são inversas uma da outra:

$$P = \mathsf{dec}(\mathsf{cod}(P)) \tag{2.46}$$

$$n = \mathsf{cod}(\mathsf{dec}(n)) \tag{2.47}$$

Descodificação de programas

A secção anterior mostrou como codificar um programa. *Como fazer o processo inverso?* Por exemplo, qual o programa cujo código é 99? E o programa com código 1703? A codificação cod é uma função bijectiva e possui função inversa dec. Se a cada programa corresponde um e um só número natural, a cada número corresponderá um e um só programa. Entendendo como se calcula dec entendemos como se processa esta descodificação.

A função cod codifica as várias instruções de um dado programa P num número n (que, reparando nas potências de dois na fórmula de cod, convém pensar na sua escrita binária). Para «recuperar» o programa $P = \mathsf{dec}\,(n)$, o primeiro passo é decompor $n+1$ na sua expressão binária ($n+1$ porque se descontou 1 na expressão de cod).

Esta função, para cada instrução, acumula os códigos das instruções anteriores somando-lhe um valor crescente (desde 0 até $k-1$). Para inverter este cálculo, devemos encontrar as diferenças dos expoentes subtraindo progressivamente esse valor. Por exemplo, para o número 99:

$$99 = 100 - 1 = 1100100_2 - 1 = 2^2 + 2^5 + 2^6 - 1$$

Como obtivemos três potências de 2, $\mathsf{dec}\,(99)$ é um programa com três instruções, $\mathsf{dec}\,(99) = [I_1, I_2, I_3]$. Temos o seguinte sistema de equações lineares (nas incógnitas $\beta\,(I_1)\,, \beta\,(I_2)$ e $\beta\,(I_3)$) para resolver:

$$\begin{aligned} \beta\,(I_1) &= 2 \\ \beta\,(I_1) + \beta\,(I_2) + 1 &= 5 \\ \beta\,(I_1) + \beta\,(I_2) + \beta\,(I_3) + 2 &= 6 \end{aligned}$$

Resolvendo este sistema ficamos com

$$\begin{aligned} \beta\,(I_1) &= 2 \\ \beta\,(I_2) &= 2 \\ \beta\,(I_3) &= 0 \end{aligned}$$

Só nos falta encontrar as instruções URM que têm estes códigos (já se pode concluir que a 1.ª e a 2.ª instrução são iguais). Para isso precisamos de usar a inversa de β, dada por $\beta^{-1}(n) =$

$$\begin{cases} \mathtt{Z}(q+1) & \text{se } r=0 \\ \mathtt{S}(q+1) & \text{se } r=1 \\ \mathtt{T}(\Pi_1(q)+1, \Pi_2(q)+1) & \text{se } r=2 \\ \mathtt{J}(\Pi_1(\Pi_1(q))+1, \Pi_1(\Pi_2(q))+1, \Pi_2(q)+1) & \text{se } r=3 \end{cases} \tag{2.48}$$

onde q e r são, respectivamente, o quociente e resto da divisão inteira de n por 4 (isto é, $n = 4q + r$ com $0 \leq r \leq 3$). Reparemos que dado o natural n, o resto r da divisão por quatro indica-nos o tipo de instrução e do quociente q podemos descobrir os argumentos dessa instrução. Assim, continuando o nosso exemplo:

$$\begin{aligned} \beta^{-1}(0) &= \mathtt{Z}(0+1) = \mathtt{Z}(1) \\ \beta^{-1}(2) &= \mathtt{T}(\Pi_1(0)+1, \Pi_2(0)+1) = \mathtt{T}(0+1, 0+1) = \mathtt{T}(1,1) \end{aligned}$$

Desta forma podemos concluir que $\mathsf{dec}(99) = [\mathtt{T}(1,1), \mathtt{T}(1,1), \mathtt{Z}(1)]$.

Vejamos um outro exemplo. Qual o programa $\mathsf{dec}(1703)$? Primeiro precisamos converter 1704 em base dois:

$$1703 = 1704 - 1 = 11010101000_2 - 1 = 2^3 + 2^5 + 2^7 + 2^9 + 2^{10} - 1$$

Este resultado significa que o programa contém cinco instruções, *i.e.*, $\mathsf{dec}(1703) = [I_1, I_2, I_3, I_4, I_5]$. Ficamos então com o seguinte sistema:

$$\begin{aligned} \beta(I_1) &= 3 \\ \beta(I_1) + \beta(I_2) + 1 &= 5 \\ \beta(I_1) + \beta(I_2) + \beta(I_3) + 2 &= 7 \\ \beta(I_1) + \beta(I_2) + \beta(I_3) + \beta(I_4) + 3 &= 9 \\ \beta(I_1) + \beta(I_2) + \beta(I_3) + \beta(I_4) + \beta(I_5) + 4 &= 10 \end{aligned}$$

que tem solução

$$\begin{aligned}\beta(I_1) &= 3\\ \beta(I_2) &= 1\\ \beta(I_3) &= 1\\ \beta(I_4) &= 1\\ \beta(I_5) &= 0\end{aligned}$$

No exemplo anterior já descobrimos que a instrução de código zero é $\mathtt{Z}(1)$. Falta encontrar as instruções de códigos 1 e 3:

$$\begin{aligned}\beta^{-1}(1) &= \mathtt{S}(0+1) = \mathtt{S}(1)\\ \beta^{-1}(3) &= \mathtt{J}(\Pi_1(\Pi_1(0))+1, \Pi_1(\Pi_2(0))+1, \Pi_2(0)+1)\\ &\quad \cdots\\ &= \mathtt{J}(1,1,1)\end{aligned}$$

Finalmente

$$\mathsf{dec}(1703) = [\mathtt{J}(1,1,1), \mathtt{S}(1), \mathtt{S}(1), \mathtt{S}(1), \mathtt{Z}(1)].$$

De volta ao conjunto $\mathcal{P}$

Com a codificação cod podemos afirmar o seguinte:

Teorema 2.4.2 *O conjunto $\mathcal{P}$ é enumerável.*

Demonstração. Mostrou-se que a função cod é uma bijecção entre $\mathcal{P}$ e $\mathbb{N}$. Pela definição de conjunto enumerável, concluímos que $\mathcal{P}$ é enumerável. □

Falta-nos responder, nesta secção, à seguinte questão: *A codificação de programas,* cod*, bem como a sua inversa, são funções computáveis?* Isto, segundo a definição de função computável, é o mesmo que perguntar se existem programas URM capazes de calcular cod e dec. Evidentemente que a URM não armazena instruções URM nos seus registos

(afinal, definimos estes códigos por alguma razão) mas com uma codificação apropriada, há algo na expressão destas funções que não possa ser calculado pela URM?

Lema 2.4.3
A função cod *e a sua inversa* dec *são computáveis.*

Demonstração. Será apresentado um esquema de prova (os detalhes não são relevantes neste contexto e os programas que computam as funções a seguir referidas são deixados como exercício).

A função cod é dada por uma expressão definida exclusivamente à custa de funções aritméticas ($+, \dot{-}, \times, \div, x^y$, todas computáveis) e ainda da função β. Por sua vez, a expressão de β inclui multiplicações, somas e as quatro codificações das respectivas instruções. As codificações $\alpha_{\mathtt{T}}$ e $\alpha_{\mathtt{J}}$ usam a função Π que ainda é computável. Deste modo, a função cod é computável.

Para a função inversa de cod são utilizadas as funções $(x)_1$, Π_1 e Π_2 que são computáveis (além das funções aritméticas). Do mesmo modo, também a função dec é computável. □

Definição 2.4.4 *Seja A um conjunto enumerável, pela bijecção $f : A \to \mathbb{N}$. Diz-se que A é* ***efectivamente enumerável*** *se f e f^{-1} forem ambas computáveis.*

Teorema 2.4.5 (Enumerabilidade efectiva de $\mathcal{P}$)
O conjunto $\mathcal{P}$ é efectivamente enumerável.

Demonstração. Pelo lema 2.4.3, a bijecção $\mathsf{cod} : \mathcal{P} \to \mathbb{N}$ e a sua inversa são computáveis. □

A partir da codificação cod qualquer programa tem uma representação na forma de número natural. Considerando isso, introduzimos a seguinte notação:

Definição 2.4.6 *Sejam $m, n \in \mathbb{N}$.*

- *a* ***função*** $\phi_m^{(n)}$ *é a função n-ária calculada por* $\mathsf{dec}\,(m)$. *Isto é, para cada* $\boldsymbol{x} \in \mathbb{N}^n$

$$\phi_m^{(n)}(\boldsymbol{x}) = y \;\Leftrightarrow\; \mathsf{dec}\,(m)\,(\boldsymbol{x}) \downarrow y \tag{2.49}$$

- *o* ***domínio*** *de* $\phi_m^{(n)}$ *é*

$$\mathcal{W}_m^{(n)} \;=\; \{\boldsymbol{x} \in \mathbb{N}^n \mid \mathsf{dec}\,(m)\,(\boldsymbol{x}) \downarrow\} \tag{2.50}$$

- *a* ***imagem*** *de* $\phi_m^{(n)}$ *é*

$$\mathcal{E}_m^{(n)} \;=\; \{y \in \mathbb{N} \mid \exists \boldsymbol{x} \in \mathbb{N}^n : \mathsf{dec}\,(m)\,(\boldsymbol{x}) \downarrow y\} \tag{2.51}$$

Quando $n = 1$ *escrevemos apenas* ϕ_m, $\mathcal{W}_m$ *e* $\mathcal{E}_m$ *respectivamente.*

Alguns exemplos (assumindo que as funções seguintes têm assinatura $\mathbb{N} \to \mathbb{N}$):

- O programa $[\mathtt{Z}\,(2)\,,\mathtt{S}\,(1)\,,\mathtt{Z}\,(1)\,,\mathtt{S}\,(1)\,,\mathtt{Z}\,(1)]$ tem código 1743 e, para cada n, calcula a função n-ária constante, igual a zero. Esta função é total. Assim,

$$\forall x \in \mathbb{N}\; \phi_{1703}(x) = 0, \mathcal{W}_{1703} = \mathbb{N} \text{ e } \mathcal{E}_{1703} = \{0\}\,.$$

- O programa $[\mathtt{J}\,(1,2,1)\,,\mathtt{S}\,(1)]$ tem código 10239 e calcula a função unária $x \mapsto x+1$ para valores maior que zero, divergindo para $x = 0$. Assim,

$$\begin{array}{rcl} \phi_{10239}(x) & = & \begin{cases} x+1 & \text{se } x > 0 \\ \perp & \text{c.c} \end{cases}\,, \\ \mathcal{W}_{10239} & = & \mathbb{N} \setminus \{0\} \text{ e} \\ \mathcal{E}_{10239} & = & \mathbb{N} \setminus \{0,1\}\,. \end{array}$$

- O programa $\mathtt{Omega} = [\mathtt{J}\,(1,1,1)]$ tem código 3 e calcula a função que diverge em qualquer argumento. Assim,

$$\forall x\; \phi_3(x) \uparrow, \mathcal{W}_3 = \emptyset \text{ e } \mathcal{E}_3 = \emptyset.$$

É importante recordarmo-nos que um programa $\mathsf{dec}\,(m)$ calcula diferentes funções, conforme a aridade do argumento. Por exemplo, o programa $\mathsf{dec}\,(1703)$ calcula as seguintes funções (distintas!):

- $\phi_{1703}\,(x) = 0$;
- $\phi^{(2)}_{1703}\,(x, y) = 0$;
- $\phi^{(3)}_{1703}\,(x, y, z) = 0$;
- *etc.*

Definição 2.4.7 (Conjunto das funções computáveis)
O conjunto de todas as funções computáveis é representado por $\mathcal{C}$ e o conjunto das funções n-árias computáveis por $\mathcal{C}_n$.

As funções $\mathsf{soma}, \mathsf{pred}$ e monus pertencem a $\mathcal{C}$. Em particular $\mathsf{soma} \in \mathcal{C}_2, \mathsf{pred} \in \mathcal{C}_1$ e $\mathsf{monus} \in \mathcal{C}_2$. Como a cada função computável correspondem vários programas URM, e como o conjunto $\mathcal{P}$ é enumerável podemos afirmar que tanto $\mathcal{C}$ como $\mathcal{C}_n$ são conjuntos enumeráveis.

Teorema 2.4.8
Os conjuntos $\mathcal{C}_n\,(n \geq 1)$ e $\mathcal{C}$ são enumeráveis.

Demonstração. O conjunto $\mathcal{C}_n$ é enumerável se existir uma função sobrejectiva $\mathbb{N} \to \mathcal{C}$. Ora cada função n-ária computável f tem um certo índice $f = \phi^{(n)}_i$ pelo que a função

$$\begin{array}{rccc} \phi^{(n)}: & \mathbb{N} & \to & \mathcal{C}_n \\ & i & \mapsto & \phi^{(n)}_i \end{array}$$

é sobrejectiva. Como cada $\mathcal{C}_n$ é enumerável e $\mathcal{C}$ é a união dos $\mathcal{C}_n$, podemos também afirmar que $\mathcal{C}$ é enumerável. □

Estarão todas as funções em $\mathcal{C}$? Não! É possível demonstrar que o conjunto dos reais, $\mathbb{R}$, não é enumerável (anexo A.3, p.261): há mais números reais do que naturais. Então também há mais números reais que programas em $\mathcal{P}$. A maioria das funções de assinatura $\mathbb{R} \to \mathbb{R}$

não são computáveis. Veremos ainda no capítulo 4 que há funções de assinatura $\mathbb{N} \to \mathbb{N}$ que também não são computáveis.

Serão $\mathcal{C}_n$ $(n \geq 1)$ e $\mathcal{C}$ efectivamente enumeráveis? Para o serem, a bijecção entre $\mathcal{C}_n$ e $\mathbb{N}$ (F_m na demonstração anterior) teria de ser computável. Mas veremos no capítulo 4 que o predicado «$\phi_x^{(n)} \neq \phi_y^{(n)}$» é indecidível. Por não ser possível, em geral, comparar computacionalmente funções, não existe uma bijecção computável adequada a este problema, logo $\mathcal{C}_n$ e $\mathcal{C}$ não são efectivamente enumeráveis.

2.5 Linguagens e Predicados

Uma **linguagem** é um conjunto. Este contém todas as palavras (*i.e.*, sequências de zero ou mais símbolos de um alfabeto finito Σ) dessa linguagem. Vamos considerar, no resto da secção, o caso particular de linguagens como subconjuntos[12] de $\mathbb{N}$ (e, portanto, constituídos por palavras finitas).

Definição 2.5.1 (Característica de uma linguagem)
*Seja A uma linguagem. A **função característica** de A é*

$$\begin{array}{rccl} \mathsf{c}_A: & \mathbb{N} & \to & \{0,1\} \\ & x & \mapsto & \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases} \end{array} \tag{2.52}$$

Usando a função característica de uma linguagem A, é possível determinar, para qualquer argumento $x \in \mathbb{N}$, se este pertence ou não a A.

Definição 2.5.2 (Linguagem recursiva)
*Uma linguagem é **recursiva** (ou **decidível**) se a sua função característica for computável.*

*Uma linguagem não recursiva diz-se **indecidível**.*

[12] Não há perda de generalidade pois qualquer linguagem $A \subseteq \Sigma^*$ corresponde a um subconjunto de $\mathbb{N}$ através de alguma codificação α_{Σ^*}.

Para qualquer linguagem recursiva A existe um programa URM que computa se uma palavra qualquer pertence ou não à linguagem, *i.e.*, computa c_A.

Dado um predicado φ qualquer, (por exemplo «par» ou «primo») podemos associar-lhe uma linguagem L_φ (isto é, um subconjunto) de $\mathbb{N}$ da seguinte forma:

$$\mathsf{L}_\varphi = \{x \mid \varphi(x)\} \tag{2.53}$$

que verifica a propriedade

$$x \in \mathsf{L}_\varphi \Leftrightarrow \varphi(x)\,. \tag{2.54}$$

Deste modo, um algoritmo para decidir um determinado predicado (por exemplo, par) pode ser substituído por um algoritmo para decidir uma determinada linguagem ($\mathsf{L}_{\mathsf{par}} = \mathsf{Pares}$).

Reciprocamente, também a cada linguagem corresponde um predicado. Dada uma linguagem $A \subseteq \mathbb{N}$, define-se o predicado Φ_A da seguinte forma:

$$\Phi_A(x) \Leftrightarrow x \in A.$$

Exemplo 2.5.3 *Formas equivalentes de dizer que «x é par»:*

1. $\Phi_{\mathsf{Pares}}(x)$;
2. $x \in \mathsf{Pares}$;
3. $\mathsf{par}(x)$;
4. $\mathsf{c}_{\mathsf{par}}(x) = 1$;
5. $\mathsf{c}_{\mathsf{Pares}}(x) = 1$;

Notemos que, para qualquer linguagem A e qualquer predicado φ,

$$A = \mathsf{L}_{\Phi_A} \tag{2.55}$$

$$\varphi \Leftrightarrow \Phi_{\mathsf{L}_\varphi}. \tag{2.56}$$

Teorema 2.5.4 (Operações de linguagens recursivas)
Seja A e B linguagens recursivas. As seguintes linguagens são recursivas:

1. $\overline{A}$;
2. $A \cap B$;
3. $A \cup B$;
4. $A \setminus B$.

Demonstração. Sendo A e B recursivas, as respectivas funções característica c_A e c_B são computáveis. Então também são computáveis

1. $\mathsf{c}_{\overline{A}}(x) = 1 \dot{-} \mathsf{c}_A(x)$;
2. $\mathsf{c}_{A\cap B}(x) = \mathsf{c}_A(x) \times \mathsf{c}_B(x)$;
3. $\mathsf{c}_{A\cup B}(x) = \mathsf{max}(\mathsf{c}_A(x), \mathsf{c}_B(x))$;
4. $\mathsf{c}_{A\setminus B}(x) = \mathsf{c}_A(x) \dot{-} \mathsf{c}_B(x)$.

□

Às linguagens recursivas correspondem predicados «totais», *i.e.*, as suas funções característica têm domínio $\mathbb{N}$.

Definição 2.5.5 *A linguagem A é **recursivamente enumerável** (ou **semi-decidível**) se a seguinte função for computável:*

$$\begin{array}{rccl} \mathsf{c'}_A: & \mathbb{N} & \to & \{1\} \\ & x & \mapsto & \begin{cases} 1 & \text{se } x \in A \\ \bot & \text{se } x \notin A \end{cases} \end{array} \tag{2.57}$$

*Uma linguagem que não seja recursivamente enumerável diz-se **semi-indecidível**.*

Para uma linguagem recursivamente enumerável (abrevia-se «**r.e.**») só há a garantia de ser possível determinar computacionalmente se dada palavra lhe pertence.

Fica em aberto a existência de um algoritmo capaz de identificar as palavras *fora* dessa linguagem.

Alguns exemplos de linguagens r.e.:

- qualquer linguagem recursiva
- $\{n \mid n \in \mathcal{W}_n\}$
- $\{n \mid \mathcal{W}_n \neq \emptyset\}$
- $\{n \mid \mathsf{collatz}\,(n) = 1\}$, que corresponde ao problema de Collatz descrito pelo seguinte algoritmo: enquanto $n > 1$: se n for par, $n \leftarrow n/2$ senão $n \leftarrow 3n + 1$.

Teorema 2.5.6 *Seja A uma linguagem r.e. e não recursiva. A linguagem $\overline{A}$ não é r.e.*

Demonstração. Por redução ao absurdo, se $\overline{A}$ for r.e. então a função

$$\mathsf{c'}_{\overline{A}}(x) \;=\; \begin{cases} 1 & \text{se } x \notin A \\ \perp & \text{se } x \in A \end{cases}$$

é computável. Sendo $\mathsf{c'}_A$ e $\mathsf{c'}_{\overline{A}}$ computáveis, existem programas P_1 e P_2 que as calculem. Mas então podemos criar um programa URM que corra P_1 e P_2 alternadamente, *uma instrução de cada vez*[13], obteríamos um resultado caso o argumento pertencesse a A (quando P_1 terminasse, com resultado 1) ou caso o argumento não pertencesse a A (quando P_2 terminasse, com resultado 1). Mas dessa forma, A seria uma linguagem recursiva, o que contradiz a hipótese do teorema. □

Teorema 2.5.7 *Se A e $\overline{A}$ são linguagens r.e. então A é recursiva.*

[13]Referiremos os detalhes técnicos relativos a este aspecto na secção 4.3.2. Por enquanto é suficiente a intuição de que é possível fazer um programa URM que alterne entre as instruções de P_1 e de P_2.

Demonstração. Usando o método descrito no teorema 2.5.6 é possível encontrar um programa que calcule para qualquer valor, se este pertence ou não à linguagem. Pela definição 2.5.2, sendo a função característica de A computável, A é uma linguagem recursiva. □

Voltaremos às linguagens recursivas e r.e. no capítulo 4.

2.6 Funções Algorítmicas

O termo função **algorítmica** descreve qualquer função cujo processo de cálculo é claramente descrito por um algoritmo (numa qualquer linguagem não ambígua). Por outro lado, uma função é **computável** se existe uma descrição (quer seja através de um programa URM, numa configuração da máquina de Turing ou noutra equivalente) que a calcule. Com estas noções podemos anunciar a seguinte tese:

Tese 2.6.1 (Tese de Church) *Todas as funções algorítmicas são computáveis.*

Esta tese não é um teorema dado que a noção «função algorítmica» não está formalmente definida. E isso porque o conceito geral de algoritmo não é formalmente definido mas apenas descrito intuitivamente. É esta intuição que nos permite reconhecer um algoritmo quando vemos uma descrição adequada. Por exemplo, os programas URM são algoritmos porque facilmente os reconhecemos como tal (daí a importância de serem muito simples para remover qualquer dúvida razoável). Assim, todos os programas URM calculam funções algorítmicas, *i.e.*, $\mathcal{C}$ está incluído no conjunto das funções algorítmicas.

Mas isso nada nos diz sobre se existe um certo algoritmo X que não seja computável pela URM (ou pela Máquina de Turing). Ao aceitar a Tese de Church, consideramos que tal X não existe. Um forte argumento a favor desta crença é que ninguém, entre os milhares de matemáticos e

informáticos, ao longo de quase um século, tenha encontrado um algoritmo que a invalide.

Outro facto em defesa da Tese de Church é que *todas* as definições de computabilidade clássica[14] apresentadas até hoje, são equivalentes à URM (e.g., pode-se demonstrar que a Máquina de Turing calcula as mesmas funções que a URM) e, por transitividade são equivalentes entre si. Ninguém ainda apresentou um formalismo baseado em recursos finitos (programas finitos, argumentos finitos...) capaz de calcular mais funções do que a URM. Foi este argumento que, em 1936, levou Church, Turing (e outros) a defender que o conjunto das funções algorítmicas não só inclui mas é igual a $\mathcal{C}$.

Assim, para provar que dada função é computável basta optar entre:

- descrever um programa URM que calcule essa função;
- demonstrar axiomaticamente a sua computabilidade (*cf.* capítulo 3);
- encontrar uma descrição num outro formalismo equivalente (e.g., criar uma máquina de Turing que calcule a função);
- descrever a solução de modo a que não fiquem dúvidas que se trata de um algoritmo e, de seguida, invocar a Tese de Church.

Por contraposição, se provarmos que dada função não é computável (o tema central do capítulo 4), afirmamos, através da Tese de Church, que essa função não é algorítmica.

2.7 Funções Universais

A secção 2.4 mostrou-nos como cada programa pode ser associado a um número natural e, inversamente, a partir de um número, se obtém o

[14]Nada afirmamos sobre outros tipos de computações que se encontram fora do âmbito deste livro, como por exemplo, as computações probabilística, contínua ou quântica.

respectivo programa. É assim possível definir funções de domínio natural que interpretem o seu argumento numérico como sendo o código de um certo programa URM. Uma dessas funções é definida a seguir:

Definição 2.7.1 (Função Universal)
A ***função universal*** $(n+1)$*-ária é*

$$\begin{array}{rccc} \Psi_{\mathcal{U}}^{(n)}: & \mathbb{N}^{n+1} & \to & \mathbb{N} \\ & (x, a_1, \ldots, a_n) & \mapsto & \phi_x^{(n)}(a_1, \ldots, a_n) \end{array} \tag{2.58}$$

O que significa esta definição? A função $\Psi_{\mathcal{U}}^{(n)}$ tem $n+1$ argumentos naturais. O primeiro argumento, x, representa a função n-ária computada pelo programa URM de código x, *i.e.*, a função $\phi_x^{(n)}$. O resultado final é a imagem de $(a_1, \ldots, a_n)$ por $\phi_x^{(n)}$. Como qualquer função computável pode ser descrita por um código natural, a função $\Psi_{\mathcal{U}}^{(n)}$, dando os valores apropriados, calcula o resultado de qualquer função n-ária computável! Daí ser designada por função universal.

Para cada aridade $n \geq 1$ existe uma função universal $\Psi_{\mathcal{U}}^{(n)}$. A função $\Psi_{\mathcal{U}}^{(2)}$ calcula os resultados das funções *unárias* $\phi_0, \phi_1, \phi_2, \ldots$; $\Psi_{\mathcal{U}}^{(3)}$ calcula os resultados das funções *binárias* $\phi_0^{(2)}, \phi_1^{(2)}, \phi_2^{(2)}, \ldots$ e assim sucessivamente.

Mas serão estas funções universais computáveis? A princípio pode parecer estranho que exista uma função computável que calcule qualquer outra função computável. Porquê? Porque então existe um programa URM que a computa (um **programa universal**). A ser verdade, implicaria a existência de um limite prático à «complicação» dos programas: não são necessários programas mais complicados do que um programa universal porque as computações de um programa desses podem ser «simuladas» calculando o seu código e usando-o no programa universal.

Teorema 2.7.2 *Para qualquer natural $n > 0$, a função $\Psi_{\mathcal{U}}^{(n)}$ é computável.*

Demonstração. Para provar que $\Psi_{\mathcal{U}}^{(n)}$ é computável basta mostrar que existe um programa URM que compute $\Psi_{\mathcal{U}}^{(n)}$. Mas, para evitar a complexidade de encontrar esse programa, descrevemos informalmente esse algoritmo.

Para calcular $\Psi_{\mathcal{U}}^{(n)}(x, a_1, \ldots, a_n)$:

1. obtemos o programa $\mathsf{dec}(x)$ através do processo de descodificação;
2. fazemos a computação $\mathsf{dec}(x)(a_1, \ldots, a_n)$;
3. se a computação terminar, $\Psi_{\mathcal{U}}^{(n)}(x, a_1, \ldots, a_n)$ é o valor r_1.

Esta descrição é suficientemente clara para afirmar que $\Psi_{\mathcal{U}}^{(n)}$ é uma função algorítmica. Então, pela Tese de Church, é computável. □

Sendo $\Psi_{\mathcal{U}}^{(n)}$ computável, existe um programa URM que a computa: existem programas universais. *Qual é o truque?* Um programa universal não precisa de «conter» todos os outros programas (isso, aliás, seria impossível). Basta-lhe ter um processo de descodificar o código de um qualquer programa e simular o seu funcionamento. A uma máquina capaz de executar um programa universal é costume chamar-se **computador**.

2.8 Oráculos

Nesta secção apresentamos uma extensão da URM com uma instrução extra que, quando executada, consulta uma certa função «externa», designada **oráculo**. A estas máquinas chamaremos **URMO** (*i.e.*, *URM with Oracle*).

Definição 2.8.1 *Um **oráculo** é uma função total de assinatura* $\mathbb{N} \to \mathbb{N}$.

Porquê uma função total? Consideramos que a consulta ao oráculo tem sempre um resultado definido, qualquer que seja o argumento. Não existe a possibilidade de «avariar» o oráculo com valores que não pertençam ao domínio. Porém, *não é exigido que* χ *seja computável.*

Definição 2.8.2 *A URMO executa programas URMO,* i.e.*, listas de cinco tipos de instrução:* $\mathtt{Z}, \mathtt{S}, \mathtt{T}, \mathtt{J}$ *(com o mesmo significado da URM) e ainda a instrução* $\mathtt{O}$ *associada ao oráculo* χ*. A instrução* $\mathtt{O}(n)$*, quando executada, muda o valor em* R_n *para* $\chi(r_n)$*.*

A instrução $\mathtt{O}$ funciona como uma consulta a um processo externo (o designado oráculo) que recebe o argumento, calcula o resultado e coloca-o no registo adequado na URMO.

Um exemplo de execução de $[\mathtt{S}(1), \mathtt{O}(1)]$ com oráculo $\chi(x) = x^2$:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$	
3	0	0	0	0	0	$\cdots$	Estado inicial
4	0	0	0	0	0	$\cdots$	Após $\mathtt{S}(1)$
16	0	0	0	0	0	$\cdots$	Após $\mathtt{O}(1)$ e termina

O oráculo dá-nos uma noção de computabilidade relativa. O programa P que usa o oráculo χ é também representado por P^χ, se quisermos destacar a dependência em relação a χ. Se uma função f for calculada por um programa P^χ afirmamos que f é **χ-computável**. O conjunto de todas as funções χ-computáveis representa-se por $\mathcal{C}^\chi$.

Teorema 2.8.3

1. $\chi \in \mathcal{C}^\chi$;
2. $\mathcal{C} \subseteq \mathcal{C}^\chi$;
3. $\chi \in \mathcal{C} \Rightarrow \mathcal{C} = \mathcal{C}^\chi$.

Demonstração.

1. O programa $[\mathtt{O}(1)]$ calcula a função χ;
2. Pela definição, qualquer programa URM é um programa URMO;

3. Pelo teorema anterior já sabemos que $\mathcal{C} \subseteq \mathcal{C}^\chi$, bastando verificar que $\mathcal{C}^\chi \subseteq \mathcal{C}$. Sendo a função χ computável, existe um programa P que a calcula. Assim, em cada programa URMO basta substituir as instruções $\mathtt{O}(n)$ por módulos $P[n \to n]$. Após as respectivas expansões destes módulos, obtemos um programa URM equivalente. Como $\mathcal{C}^\chi \subseteq \mathcal{C}$ e $\mathcal{C} \subseteq \mathcal{C}^\chi$ temos que a igualdade $\mathcal{C} = \mathcal{C}^\chi$.

□

A igualdade $\mathcal{C} = \mathcal{C}^\chi$ do teorema anterior diz-nos que se o oráculo utilizado for computável, a URMO só é capaz de calcular funções que sejam calculadas pela URM original.

Os programas URMO definem o conjunto $\mathcal{P}^\chi$, que é enumerável (basta adaptar a codificação $\mathtt{cod}$ para incluir instruções $\mathtt{O}$).

Linguagens e Oráculos

As linguagens podem classificar-se como recursivas ou r.e. *em relação a um oráculo.*

Definição 2.8.4 *Uma linguagem A é χ-**recursiva** se a sua função característica for χ-computável; e é χ-r.e. se a função $\mathtt{c}'_A$ for χ-computável.*

Uma linguagem A também pode ser vista como um oráculo, considerando a função característica $\mathtt{c}_A$ como a função χ. Ter acesso a $\mathtt{c}_A$ significa que se conhece por completo a linguagem A, *i.e.*, quais são e quais não são as palavras de A. Qualquer função calculada numa URMO com este oráculo diz-se $\mathtt{c}_A$-computável ou, simplesmente, A-computável. À notação das secções anteriores, neste contexto, anexa-se o sobrescrito A. Por exemplo, $\mathcal{P}^A, \varphi_x^A, \mathcal{C}^A, \ldots$

2.9 Exercícios

Exercícios Resolvidos

Programas URM sem módulos

No decorrer deste capítulo construímos, de raiz, os seguintes programas que calculam, respectivamente, as funções soma, pred e monus:

$$\texttt{Soma} = [\texttt{J}(3,2,5), \texttt{S}(1), \texttt{S}(3), \texttt{J}(1,1,1)] \tag{2.59}$$

$$\texttt{Pred} = \left[\begin{array}{l|l|l} & \texttt{J}(1,3,7) & \\ \texttt{J}(1,4,9) & \texttt{S}(2) & \texttt{T}(2,1) \\ \texttt{S}(3) & \texttt{S}(3) & \\ & \texttt{J}(1,1,3) & \end{array}\right] \tag{2.60}$$

$$\texttt{Monus} = \left[\begin{array}{l|l|l} \texttt{J}(1,4,9) & \texttt{J}(1,4,9) & \\ \texttt{J}(2,4,5) & \texttt{S}(3) & \texttt{T}(3,1) \\ \texttt{S}(4) & \texttt{S}(4) & \\ \texttt{J}(1,1,1) & \texttt{J}(1,1,5) & \end{array}\right] \tag{2.61}$$

Exercício 2.9.1 *Vamos, de seguida, construir programas que calculem as seguintes funções:*

- $\mathsf{max}(x,y) = \begin{cases} x & \text{se } x > y \\ y & \text{caso contrário} \end{cases}$ *, o maior dos dois valores;*
- $\mathsf{produto}(x,y) = x \times y$*, a multiplicação de* x *por* y*;*
- $\mathsf{rm}(x,y)$*, o resto da divisão inteira de* y *por* x *(com* $\mathsf{rm}(0,y) = y$*);*
- *A função característica do predicado «*$x = y$*».*

Um programa para max(x,y). Para determinar qual dos valores é o maior, usaremos um registo como contador (seja R_3). Este contador será comparado com ambos os valores x e y (que se encontram em R_1 e R_2). O menor destes valores será o primeiro a igualar R_3. O resultado do programa será o valor do outro argumento.

Temos, assim, o seguinte estado típico no espaço de trabalho:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	y	k	0	0	0	$\cdots$

O programa resultante:

$$\texttt{Max} = \left[\begin{array}{ll} & \texttt{J}\,(1,3,5) \quad \textit{se } k = x,\ y > x\textit{: fim com } R_1 \leftarrow y \\ & \texttt{J}\,(2,3,6) \quad \textit{se } k = y,\ x > y\textit{: fim; } R_1 \textit{ já tem } x \\ & \texttt{S}\,(3) \quad\quad \textit{senão, incrementar } k \ldots \\ & \texttt{J}\,(1,1,1) \quad \ldots \textit{ e repetir o ciclo} \\ I_5: & \texttt{T}\,(2,1) \end{array}\right] \tag{2.62}$$

Um programa para produto (x, y)**.** Para determinar o resultado do produto, efectuaremos y somas do valor x. Vamos precisar de três registos auxiliares: um para guardar quantas somas já realizamos (será R_4), um para guardar, em cada soma, quanto de x já foi somado (será R_5) e um outro registo que acumula todos estes incrementos para, no fim da computação, conter o resultado final (será R_3).

Temos, assim, o seguinte estado típico no espaço de trabalho:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	y	$kx + l$	k	l	0	$\cdots$

O programa resultante:

$$\texttt{Prod} = \left[\begin{array}{ll} & \texttt{J}\,(2,4,9) \quad \textit{se } y = k,\ x \textit{ foi somado } y \textit{ vezes} \\ & \\ & \texttt{J}\,(1,5,6) \quad \textit{já somámos um } x\textit{?} \\ & \texttt{S}\,(5) \quad\quad \textit{incrementar contador } l \\ & \texttt{S}\,(3) \quad\quad \textit{incrementar a soma global, } kx + l \\ & \texttt{J}\,(1,1,2) \quad \textit{repetir ciclo até somar outro } x \\ & \\ I_6: & \texttt{S}\,(4) \quad\quad \textit{somámos mais um } x,\ k \leftarrow k + 1 \\ & \texttt{Z}\,(5) \quad\quad \textit{reiniciar } l \textit{ para a próx}^a \textit{ soma de } x \\ & \\ & \texttt{J}\,(1,1,1) \quad \textit{recomeçar} \\ & \\ I_9: & \texttt{T}\,(3,1) \quad\quad \textit{transferir resultado de } R_3 \textit{ para } R_1 \end{array}\right] \tag{2.63}$$

Um programa para $\mathsf{rm}\,(x, y)$. Vamos novamente utilizar um contador que irá crescer de 0 a y (será R_4). Ao mesmo tempo, para cada incremento de R_4, vamos actualizando qual o resto da divisão inteira de R_4 por x (será R_3). Este cálculo é simples: ao incrementar R_4 por uma unidade, R_3 também é incrementado por uma unidade, excepto se $R_3 = x$ então R_3 volta a zero (pois estamos perante um R_4 em que x é divisor). Quando R_4 for igual a y, o valor R_3 contém o resto de y/x.

Temos, assim, o seguinte estado típico no espaço de trabalho:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	y	k/x	k	0	0	$\cdots$

O programa resultante:

$$
\mathtt{Rm} = \left[\begin{array}{lll}
 & \mathtt{J}\,(1,3,11) & \textit{caso particular, } x = 0 \\
 & & \\
I_2: & \mathtt{J}\,(1,3,7) & \textit{se } R_3 = x\textit{, } x \textit{ é divisor de } k \\
 & & \\
I_3: & \mathtt{J}\,(2,4,9) & \textit{se } y = k\textit{, o resultado está em } R_3 \\
 & \mathtt{S}\,(4) & \textit{incrementar } k \\
 & \mathtt{S}\,(3) & \textit{incrementar o resto} \\
 & \mathtt{J}\,(1,1,2) & \textit{repetir o ciclo} \\
 & & \\
I_7: & \mathtt{Z}\,(3) & \textit{reiniciar o resto: } x \textit{ é divisor de } k \\
 & \mathtt{J}\,(1,1,3) & \textit{voltar ao ciclo } I_2\text{–}I_6 \\
 & & \\
I_9: & \mathtt{T}\,(3,1) & \textit{fim: transferir } R_3 \textit{ para } R_1 \ldots \\
 & \mathtt{J}(1,1,12) & \ldots \textit{e terminar o programa} \\
 & & \\
I_{11}: & \mathtt{T}\,(2,1) & \textit{no caso particular, } \mathsf{rm}\,(0,y) = y
\end{array}\right] \tag{2.64}
$$

Um programa para «$x = y$». Para calcular este predicado obtemos o programa que calcule a respectiva função característica:

$$\mathsf{c}_{=}: \begin{array}{ccc} \mathbb{N}^2 & \to & \{0,1\} \\ (x,y) & \mapsto & \begin{cases} 1 & \text{se } x = y \\ 0 & \text{caso contrário} \end{cases} \end{array} \tag{2.65}$$

O programa resultante:

$$\mathsf{Eq} = \left[\begin{array}{lll} & \mathtt{J}(1,2,4) & \textit{se } x = y \textit{ o resultado será } 1 \\ & & \\ & \mathtt{Z}(1) & \textit{se } x \neq y,\ R_1 \leftarrow 0 \ \ldots \\ & \mathtt{J}(1,1,6) & \ldots \textit{terminar} \\ & & \\ I_4: & \mathtt{Z}(1) & \textit{se } x = y \ \ldots \\ & \mathtt{S}(1) & \ldots R_1 \leftarrow 1 \end{array}\right] \tag{2.66}$$

Programas URM com módulos

Exercício 2.9.2 *Definir programas URM para as seguintes funções:*

- $\mathsf{exp}\,(x,y) = x^y$*;*
- $\mathsf{abs}\,(x,y) = |x-y|$*;*
- $\mathsf{fact}\,(x) = x!$*;*
- $\mathsf{D}\,(x)$*, o número de divisores de* x *(com* $\mathsf{D}\,(0) = 1$*)*

Para obter estas soluções temos a opção de usar como módulos os programas definidos até agora.

Um programa para exp (x,y). Da mesma forma que a multiplicação de $x \times y$ corresponde a $x + \cdots + x$, y vezes, a potência x^y corresponde a $x \times \cdots \times x$, y vezes. Teremos um contador (em R_4) e um registo que acumula as multiplicações realizadas até ao momento (em R_3). Quando $r_4 = y$, o resultado é r_3.

Temos, assim, o seguinte estado típico no espaço de trabalho:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	y	k^x	k	0	0	$\cdots$

O programa resultante:

$$\texttt{Exp} = \left[\begin{array}{lll} & \texttt{J}\,(1,4,7) & \textit{se } x = 0,\ 0^y = 0 \\ & & \\ & \texttt{S}\,(3) & R_3 \leftarrow 1,\ \textit{el. neutro de } \times \\ & & \\ & \texttt{J}\,(2,4,7) & \textit{já fez } y \textit{ multiplicações?} \\ & \texttt{Prod}\,[1,3 \rightarrow 3] & \textit{senão, multiplicar por } x \ldots \\ & \texttt{S}\,(4) & \ldots \textit{e incrementar o contador} \\ & \texttt{J}\,(1,1,3) & \textit{repetir o ciclo } I_3\text{–}I_6 \\ & & \\ I_7: & \texttt{T}\,(3,1) & \textit{transferir } R_3 \textit{ para } R_1 \end{array}\right] \tag{2.67}$$

Um programa para abs (x, y)**.** O valor absoluto $|x - y|$ é dado pela soma de dois valores: $\mathsf{monus}\,(x, y)$ e $\mathsf{monus}\,(y, x)$. Porquê? Se $x > y$, o resultado estará em $\mathsf{monus}\,(x, y)$ e o valor de $\mathsf{monus}\,(y, x)$ será zero, em nada contribuindo para a soma final. Se $x < y$ temos o raciocínio simétrico. Quando $x = y$, ambas as expressões do monus dão zero (e $|x - y| = 0$) neste caso.

Com os programas `Soma` e `Monus`, obtemos:

$$\texttt{Abs} = [\texttt{Monus}\,[1, 2 \rightarrow 3]\,, \texttt{Monus}\,[2, 1 \rightarrow 4]\,, \texttt{Soma}\,[3, 4 \rightarrow 1]] \tag{2.68}$$

Um programa para fact (x, y)**.** O factorial de x é dado pela expressão $1 \times 2 \times \cdots \times x$. Basta usar um contador (em R_3) e um registo que acumule as multiplicação desde 1 até r_4 (em R_2).

Usamos o seguinte estado típico no espaço de trabalho:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	$k!$	k	0	0	0	$\cdots$

O programa resultante:

$$\texttt{Fact} = \left[\begin{array}{lll} & \texttt{S}\,(2) & R_2 \leftarrow 1,\ \textit{el. neutro de}\ \times \\ & & \\ & \texttt{J}\,(1,3,6) & \textit{se}\ x = r_3,\ \textit{sair do ciclo} \\ & \texttt{S}\,(3) & \textit{senão, incrementar}\ k\ \ldots \\ & \texttt{Prod}\,[2,3 \rightarrow 2] & \ldots\ \textit{e fazer a}\ \times\ \textit{seguinte} \\ & \texttt{J}\,(1,1,2) & \textit{repetir o ciclo}\ I_2\text{-}I_5 \\ & & \\ I_6: & \texttt{T}\,(2,1) & \textit{transferir}\ R_2\ \textit{para}\ R_1 \end{array}\right] \tag{2.69}$$

Um programa para $\mathsf{D}\,(x)$. Para calcular o número de divisores de x precisamos de um contador (em R_2) incrementado desde 1 até x e verificar quantos destes números divididos por x dão resto zero (guardamos o resultado parcial dos divisores encontrados em R_4). Com o módulo `Rm` o trabalho fica facilitado.

Usamos o seguinte estado típico no espaço de trabalho:

R_1	R_2	R_3	R_4	R_5	R_6	$\cdots$
x	k	k/x	$\mathsf{D}\,(k)$	0	0	$\cdots$

O programa resultante:

$$\mathtt{D} = \left[\begin{array}{lll}
 & \mathtt{J}\,(1,2,11) & \textit{caso especial: } x = 0 \\
 & & \\
 & \mathtt{S}\,(2) & \textit{incrementar o contador } k \\
 & \mathtt{Rm}\,[2,1 \to 3] & \textit{calcular o resto de } x/k \\
 & \mathtt{J}\,(3,5,7) & \textit{se deu resto zero, saltar para } I_7 \\
I_5: & \mathtt{J}\,(1,2,9) & \textit{chegou ao fim?} \\
 & \mathtt{J}\,(1,1,2) & \textit{repetir o ciclo } I_2\text{-}I_6 \\
 & & \\
I_7: & \mathtt{S}\,(4) & \textit{um divisor encontrado ...} \\
 & \mathtt{J}\,(1,1,5) & \textit{... voltar para dentro do ciclo} \\
 & & \\
I_9: & \mathtt{T}\,(4,1) & \textit{transferir o resultado final ...} \\
 & \mathtt{J}\,(1,1,12) & \textit{... e terminar} \\
 & & \\
I_{11}: & \mathtt{S}\,(1) & \textit{caso especial, } \mathsf{D}\,(0) = 1
\end{array}\right] \tag{2.70}$$

As convenções para os casos especiais ($\mathsf{D}\,(0) = 1$, $\mathsf{rm}\,(0,y) = y$) têm como objectivo definir estas funções como funções totais. Um outro motivo visa o seu uso em funções mais complexas, como veremos no capítulo seguinte.

Expansão de programas URM com módulos

Exercício 2.9.3 *Como primeiro exemplo, vamos expandir o programa* `Exp`*, que calcula* x^y*:*

$$[\mathtt{J}\,(1,4,7)\,,\mathtt{S}\,(3)\,,\mathtt{J}\,(2,4,7)\,,\mathtt{Prod}\,[1,3 \to 3]\,,\mathtt{S}\,(4)\,,\mathtt{J}\,(1,1,3)\,,\mathtt{T}\,(3,1)]$$

sabendo que o módulo `Prod` *(programa 2.63) é*

$$[\mathtt{J}\,(2,4,9)\,,\mathtt{J}\,(1,5,6)\,,\mathtt{S}\,(5)\,,\mathtt{S}\,(3)\,,\mathtt{J}\,(1,1,2)\,,\mathtt{S}\,(4)\,,\mathtt{Z}\,(5)\,,\mathtt{J}\,(1,1,1)\,,\mathtt{T}\,(3,1)]$$

Vamos seguir os passos descritos na secção 2.3, onde se explica a expansão de módulos

1. **Calcular a dimensão da área de trabalho do módulo.** A função calculada pelo programa é binária (*i.e.*, a multiplicação tem dois argumentos) e o maior índice de registo utilizado é 5, logo $\rho\,(\texttt{Prod}) = 5$. Precisamos de reservar cinco registos no início da memória para o módulo $\texttt{Prod}\,[1, 3 \to 3]$.

2. **Ajustar o programa.** Alteramos `Exp` para trabalhar cinco registos mais à frente (e acertamos as instruções `J` de forma adequada)

$$\left[\begin{array}{l|l|l} & \texttt{J}\,(6,9,9) & \\ & \texttt{S}\,(8) & \\ & 5:\ \texttt{J}\,(7,9,9) & \\ \texttt{T}\,(\mathbf{2},\mathbf{7}) & \texttt{Prod}\,[6,8 \to 8] & \texttt{T}\,(\mathbf{6},\mathbf{1}) \\ \texttt{T}\,(\mathbf{1},\mathbf{6}) & \texttt{S}\,(9) & \\ & \texttt{J}\,(6,6,5) & \\ & 9:\ \texttt{T}\,(8,6) & \end{array}\right]$$

   incluindo a transferência dos argumentos iniciais para uma zona segura (neste caso, R_6 e R_7) e a transferência do resultado final para R_1.

3. **Expandir o módulo $\texttt{Prod}\,[6, 8 \to 8]$.** É necessário mudarmos `Prod` de forma a inicializar os argumentos e limpar a restante área de trabalho (as cinco primeiras instruções), antes de o inserirmos em `Exp`. Também precisamos de fazer a transferência final do resultado para o registo apropriado (neste caso, R_8) e, como sempre, acertamos os saltos das instruções `J`.

$$\left[\begin{array}{l|l|l} & 6:\ \texttt{J}\,(2,4,14) & \\ & 7:\ \texttt{J}\,(1,5,11) & \\ \texttt{T}\,(\mathbf{6},\mathbf{1}) & \texttt{S}\,(5) & \\ \texttt{T}\,(\mathbf{8},\mathbf{2}) & \texttt{S}\,(3) & \\ \texttt{Z}\,(\mathbf{3}) & \texttt{J}\,(1,1,7) & \texttt{T}\,(\mathbf{1},\mathbf{8}) \\ \texttt{Z}\,(\mathbf{4}) & 11:\ \texttt{S}\,(4) & \\ \texttt{Z}\,(\mathbf{5}) & \texttt{Z}\,(5) & \\ & \texttt{J}\,(1,1,6) & \\ & 14:\ \texttt{T}\,(3,1) & \end{array}\right]$$

4. **Substituir a instrução do módulo pelo programa expandido** e acertar os saltos das instruções `J`. Para isso, é necessário considerar os saltos de dentro do módulo (cinco instruções antes da chamada do módulo) e os saltos no programa principal para instruções além do módulo (substituído por quinze instruções):

$$\left[\begin{array}{l|l|l}
\begin{array}{ll} & \mathtt{T}\,(2,7) \\ & \mathtt{T}\,(1,6) \\ & \mathtt{J}\,(\mathbf{6},\mathbf{9},\mathbf{23}) \\ & \mathtt{S}\,(8) \\ 5: & \mathtt{J}\,(\mathbf{7},\mathbf{9},\mathbf{23}) \\ \hline & \mathtt{T}\,(6,1) \\ & \mathtt{T}\,(8,2) \\ & \mathtt{Z}\,(3) \\ & \mathtt{Z}\,(4) \\ & \mathtt{Z}\,(5) \end{array} &
\begin{array}{ll} 11: & \mathtt{J}\,(\mathbf{2},\mathbf{4},\mathbf{19}) \\ 12: & \mathtt{J}\,(\mathbf{1},\mathbf{5},\mathbf{16}) \\ & \mathtt{S}\,(5) \\ & \mathtt{S}\,(3) \\ & \mathtt{J}\,(\mathbf{1},\mathbf{1},\mathbf{12}) \\ 16: & \mathtt{S}\,(4) \\ & \mathtt{Z}\,(5) \\ & \mathtt{J}\,(\mathbf{1},\mathbf{1},\mathbf{11}) \\ 19: & \mathtt{T}\,(3,1) \end{array} &
\begin{array}{ll} & \\ & \\ & \\ & \mathtt{T}\,(1,8) \\ \hline & \mathtt{S}\,(9) \\ & \mathtt{J}\,(6,6,5) \\ & \mathtt{T}\,(8,6) \\ 23: & \mathtt{T}\,(6,1) \end{array}
\end{array}\right]$$

No exemplo seguinte, vamos expandir o programa 2.68, que calcula $|x-y|$:

$$\mathtt{Abs} = [\mathtt{Monus}\,[1,2 \to 3]\,, \mathtt{Monus}\,[2,1 \to 4]\,, \mathtt{Soma}\,[3,4 \to 1]]$$

expandindo os módulos dos programas 2.59 e 2.61:

$$\mathtt{Soma} = [\mathtt{J}\,(3,2,5)\,, \mathtt{S}\,(1)\,, \mathtt{S}\,(3)\,, \mathtt{J}\,(1,1,1)]$$

e

$$\mathtt{Monus} = \left[\begin{array}{l|l|l}
\mathtt{J}\,(1,4,9) & \mathtt{J}\,(1,4,9) & \\
\mathtt{J}\,(2,4,5) & \mathtt{S}\,(3) & \\
\mathtt{S}\,(4) & \mathtt{S}\,(4) & \mathtt{T}\,(3,1) \\
\mathtt{J}\,(1,1,1) & \mathtt{J}\,(1,1,5) &
\end{array}\right]$$

Seguimos o procedimento de expansão:

1. **Calcular a dimensão da área de trabalho dos módulos.** O programa `Soma` necessita de três registos e `Monus` de quatro. Logo, a área de trabalho dos módulos é de quatro registos (de R_1 a R_4).

2. **Ajustar o programa.**

$$\left[\begin{array}{c|c|c} \begin{array}{l}\mathtt{T}\,(2,6)\\ \mathtt{T}\,(1,5)\end{array} & \begin{array}{l}\mathtt{Monus}\,[5,6\to 7]\\ \mathtt{Monus}\,[6,5\to 8]\\ \mathtt{Soma}\,[7,8\to 5]\end{array} & \mathtt{T}\,(5,1) \end{array}\right]$$

3. **Expandir os módulos.**

   - `Monus` $[5,6\to 7]$ expande para:

$$\mathtt{Monus567} = \left[\begin{array}{c|c|c} \begin{array}{l}\\ \\ \mathtt{T}\,(\mathbf{5},\mathbf{1})\\ \mathtt{T}\,(\mathbf{6},\mathbf{2})\\ \mathtt{Z}\,(\mathbf{3})\\ \mathtt{Z}\,(\mathbf{4})\end{array} & \begin{array}{l}\mathtt{J}\,(1,4,13)\\ \mathtt{J}\,(2,4,9)\\ \mathtt{S}\,(4)\\ \mathtt{J}\,(1,1,5)\\ \mathtt{J}\,(1,4,13)\\ \mathtt{S}\,(3)\\ \mathtt{S}\,(4)\\ \mathtt{J}\,(1,1,9)\\ \mathtt{T}\,(3,1)\end{array} & \begin{array}{l}\\ \\ \\ \\ \mathtt{T}\,(\mathbf{1},\mathbf{7})\end{array} \end{array}\right]$$

   - enquanto `Monus` $[6,5\to 8]$ expande para:

$$\mathtt{Monus658} = \left[\begin{array}{c|c|c} \begin{array}{l}\\ \\ \mathtt{T}\,(\mathbf{6},\mathbf{1})\\ \mathtt{T}\,(\mathbf{5},\mathbf{2})\\ \mathtt{Z}\,(\mathbf{3})\\ \mathtt{Z}\,(\mathbf{4})\end{array} & \begin{array}{l}\mathtt{J}\,(1,4,13)\\ \mathtt{J}\,(2,4,9)\\ \mathtt{S}\,(4)\\ \mathtt{J}\,(1,1,5)\\ \mathtt{J}\,(1,4,13)\\ \mathtt{S}\,(3)\\ \mathtt{S}\,(4)\\ \mathtt{J}\,(1,1,9)\\ \mathtt{T}\,(3,1)\end{array} & \begin{array}{l}\\ \\ \\ \\ \mathtt{T}\,(\mathbf{1},\mathbf{8})\end{array} \end{array}\right]$$

   - e o módulo `Soma` $[7,8\to 5]$ expande para:

$$\mathtt{Soma785} = \left[\begin{array}{c|c|c} \begin{array}{l}\mathtt{T}\,(\mathbf{7},\mathbf{1})\\ \mathtt{T}\,(\mathbf{8},\mathbf{2})\\ \mathtt{Z}\,(\mathbf{3})\end{array} & \begin{array}{l}\mathtt{J}\,(3,2,8)\\ \mathtt{S}\,(1)\\ \mathtt{S}\,(3)\\ \mathtt{J}\,(1,1,4)\end{array} & \mathtt{T}\,(\mathbf{1},\mathbf{5}) \end{array}\right]$$

4. **Substituir as instruções dos módulos pelos programas expandidos**, com acerto dos saltos nos módulos:

   (a) inserir `Monus567`:

$$\left[\begin{array}{c|l|l|l}
 & \mathtt{T}\,(5,1) & & \\
 & \mathtt{T}\,(6,2) & & \\
 & \mathtt{Z}\,(3) & & \\
 & \mathtt{Z}\,(4) & & \\
 & \hline
 & \mathtt{J}\,(\mathbf{1},\mathbf{4},\mathbf{15}) & & \\
 & \mathtt{J}\,(\mathbf{2},\mathbf{4},\mathbf{11}) & & \\
\mathtt{T}\,(2,6) & \mathtt{S}\,(4) & \mathtt{Monus}\,[6,5 \rightarrow 8] & \\
\mathtt{T}\,(1,5) & \mathtt{J}\,(\mathbf{1},\mathbf{1},\mathbf{7}) & \mathtt{Soma}\,[7,8 \rightarrow 5] & \mathtt{T}\,(5,1) \\
 & \mathtt{J}\,(\mathbf{1},\mathbf{4},\mathbf{15}) & & \\
 & \mathtt{S}\,(3) & & \\
 & \mathtt{S}\,(4) & & \\
 & \mathtt{J}\,(\mathbf{2},\mathbf{4},\mathbf{11}) & & \\
 & \mathtt{T}\,(3,1) & & \\
 & \hline
 & \mathtt{T}\,(1,7) & &
\end{array}\right]$$

   (b) inserir `Monus658`:

$$\left[\begin{array}{c|l|l|l}
 & \mathtt{T}\,(5,1) & \mathtt{T}\,(6,1) & \\
 & \mathtt{T}\,(6,2) & \mathtt{T}\,(5,2) & \\
 & \mathtt{Z}\,(3) & \mathtt{Z}\,(3) & \\
 & \mathtt{Z}\,(4) & \mathtt{Z}\,(4) & \\
 & \hline
 & \mathtt{J}\,(1,4,15) & \mathtt{J}\,(\mathbf{1},\mathbf{4},\mathbf{29}) & \\
 & \mathtt{J}\,(2,4,11) & \mathtt{J}\,(\mathbf{2},\mathbf{4},\mathbf{25}) & \mathtt{Soma}\,[7,8 \rightarrow 5] \\
\mathtt{T}\,(2,6) & \mathtt{S}\,(4) & \mathtt{S}\,(4) & \\
\mathtt{T}\,(1,5) & \mathtt{J}\,(1,1,7) & \mathtt{J}\,(\mathbf{1},\mathbf{1},\mathbf{21}) & \hline \\
 & \mathtt{J}\,(1,4,15) & \mathtt{J}\,(\mathbf{1},\mathbf{4},\mathbf{29}) & \mathtt{T}\,(5,1) \\
 & \mathtt{S}\,(3) & \mathtt{S}\,(3) & \\
 & \mathtt{S}\,(4) & \mathtt{S}\,(4) & \\
 & \mathtt{J}\,(2,4,11) & \mathtt{J}\,(\mathbf{1},\mathbf{1},\mathbf{25}) & \\
 & \mathtt{T}\,(3,1) & \mathtt{T}\,(3,1) & \\
 & \hline
 & \mathtt{T}\,(1,7) & \mathtt{T}\,(1,8) &
\end{array}\right]$$

(c) inserir `Soma785`:

$$\left[\begin{array}{c|c|c|c}
 & \mathtt{T}(5,1) & \mathtt{T}(6,1) & \\
 & \mathtt{T}(6,2) & \mathtt{T}(5,2) & \mathtt{T}(7,1) \\
 & \mathtt{Z}(3) & \mathtt{Z}(3) & \mathtt{T}(8,2) \\
 & \mathtt{Z}(4) & \mathtt{Z}(4) & \mathtt{Z}(3) \\
 & \hline \mathtt{J}(1,4,15) & \mathtt{J}(1,4,29) & \hline \mathtt{J}(\mathbf{3},\mathbf{2},\mathbf{40}) \\
 & \mathtt{J}(2,4,11) & \mathtt{J}(2,4,25) & \mathtt{S}(1) \\
\mathtt{T}(2,6) & \mathtt{S}(4) & \mathtt{S}(4) & \mathtt{S}(3) \\
\mathtt{T}(1,5) & \mathtt{J}(1,1,7) & \mathtt{J}(1,1,21) & \mathtt{J}(\mathbf{1},\mathbf{1},\mathbf{36}) \\
 & \mathtt{J}(1,4,15) & \mathtt{J}(1,4,29) & \hline \mathtt{T}(1,5) \\
 & \mathtt{S}(3) & \mathtt{S}(3) & \\
 & \mathtt{S}(4) & \mathtt{S}(4) & \hline \\
 & \mathtt{J}(2,4,11) & \mathtt{J}(1,1,25) & \mathtt{T}(5,1) \\
 & \mathtt{T}(3,1) & \mathtt{T}(3,1) & \\
 & \hline \mathtt{T}(1,7) & \mathtt{T}(1,8) &
\end{array}\right]$$

5. Finalmente (!), no **ajuste dos saltos nas instruções** nada mais há a mudar pois os saltos nos módulos inseridos já foram acertados e o «programa principal» não tem saltos.

Codificação de programas

Exercício 2.9.4 *Vamos calcular o código de Gödel do programa 2.66, que computa a característica do predicado «$x = y$».*

Temos então

$$\mathtt{Eq} = [\mathtt{J}(1,2,4), \mathtt{Z}(1), \mathtt{J}(1,1,6), \mathtt{Z}(1), \mathtt{S}(1)].$$

Para calcular $\mathtt{cod}(\mathtt{Eq})$ necessitamos primeiro de calcular os códigos das respectivas instruções:

- $\beta(\mathtt{J}(1,2,4)) = 4\alpha_{\mathtt{J}}(1,2,4) + 3 = 4\Pi(\Pi(0,1),3) + 3 = 4\Pi(1,3) + 3 = 95$;

- $\beta\left(\mathtt{Z}\left(1\right)\right) = 4\alpha_{\mathtt{Z}}\left(1\right) = 4\left(1-1\right) = 0$;
- $\beta\left(\mathtt{J}\left(1,1,6\right)\right) = 4\alpha_{\mathtt{J}}\left(1,1,6\right) + 3 = 4\Pi\left(\Pi\left(0,0\right),5\right) + 3 = 4\Pi\left(0,5\right) + 3 = 127$;
- $\beta\left(\mathtt{S}\left(1\right)\right) = 4\alpha_{\mathtt{S}}\left(1\right) + 1 = 4(1-1) + 1 = 1$.

Com estes códigos, o código de Gödel de $\mathtt{Eq}$, $\mathsf{cod}\left(\mathtt{Eq}\right)$, é:

$$\begin{aligned} & 2^{95} + 2^{95+0+1} + 2^{95+0+127+2} + 2^{95+0+127+0+3} + 2^{95+0+127+0+1+4} - 1 \\ = \; & 2^{95} + 2^{96} + 2^{224} + 2^{225} + 2^{227} - 1 \\ = \; & 29655941333865703774133716595721593741012743089171769379853002866687 9 \end{aligned}$$

Um número relativamente grande... mas finito.

Descodificação de programas

Exercício 2.9.5 *Qual o programa de código* 1000*,* i.e.*,* $\mathsf{dec}\left(1000\right)$*?*

Seguimos o procedimento descrito na secção 2.4:

1. Encontrar o numeral de 1001 na base 2. Usando o algoritmo A.3, p.256, 1001 em notação binária é 1111101001. Assim,

$$1000 = 2^0 + 2^3 + 2^5 + 2^6 + 2^7 + 2^8 + 2^9 - 1$$

Como a soma contém sete potências, o programa terá sete instruções,

$$\left[I_1, \ldots, I_7\right].$$

2. Determinar o código de cada I_i, resolvendo o sistema de equações

$$\begin{aligned}
\beta(I_1) &= 0 \\
\beta(I_1) + \beta(I_2) + 1 &= 3 \\
\beta(I_1) + \beta(I_2) + \beta(I_3) + 2 &= 5 \\
\beta(I_1) + \beta(I_2) + \beta(I_3) + \beta(I_4) + 3 &= 6 \\
\beta(I_1) + \beta(I_2) + \beta(I_3) + \beta(I_4) + \beta(I_5) + 4 &= 7 \\
\beta(I_1) + \beta(I_2) + \beta(I_3) + \beta(I_4) + \beta(I_5) + \beta(I_6) + 5 &= 8 \\
\beta(I_1) + \beta(I_2) + \beta(I_3) + \beta(I_4) + \beta(I_5) + \beta(I_6) + \beta(I_7) + 6 &= 9
\end{aligned}$$

Resolvendo as equações obtemos:

$$\begin{aligned}
\beta(I_1) &= 0 \\
\beta(I_2) &= 2 \\
\beta(I_3) &= 1 \\
\beta(I_4) &= 0 \\
\beta(I_5) &= 0 \\
\beta(I_6) &= 0 \\
\beta(I_7) &= 0
\end{aligned}$$

3. Descodificar os códigos das instruções. Existem apenas três instruções diferentes (valores $0, 1$ e 2):

$$\begin{aligned}
\beta^{-1}(0) &= \mathtt{Z}(1) \\
\beta^{-1}(1) &= \mathtt{S}(1) \\
\beta^{-1}(2) &= \mathtt{T}(1,1)
\end{aligned}$$

4. Apresentar o programa final. Sabendo as instruções, o programa final é

$$\mathsf{dec}(1000) = [\mathtt{Z}(1), \mathtt{T}(1,1), \mathtt{S}(1), \mathtt{Z}(1), \mathtt{Z}(1), \mathtt{Z}(1), \mathtt{Z}(1)].$$

Exercícios por Resolver

Exercício 2.9.6 *Construa programas URM sem módulos que calculem as seguintes funções:*

- $\mathsf{quatro}\,(x) = 4$*;*
- $\mathsf{dobro}\,(x) = 2x$*;*
- $\mathsf{sg}\,(x)$*, vale* 0 *se* $x > 0$*,* 1 *no caso contrário;*
- $\mathsf{half}\,(x)$*, metade de* x *(arredondando por defeito);*
- $\mathsf{half'}\,(x)$*, metade de* x *se* x *for par, diverge se* x *for ímpar;*
- $\mathsf{min}\,(x, y)$*, o menor valor entre* x *e* y*;*
- $\mathsf{qt}\,(x, y)$*, quociente da divisão inteira de* y *por* x *(com* $\mathsf{qt}\,(0, y) = 0$*);*
- $\mathsf{raiz}\,(x)$*, devolve a raiz quadrada se* x *é quadrado perfeito, diverge caso contrário;*
- *A função característica do predicado «*$x > y$*»;*
- *A função característica de «*x *é quadrado perfeito».*

Exercício 2.9.7 *Podendo usar os programas obtidos até agora como módulos, construa os programas URM que calculem as seguintes funções:*

- $f\,(x, y, z) = x + y + z$*;*
- $f\,(x, y, z) = \mathsf{max}\,(x, z) + y$*;*
- $\mathsf{p}\,(x)$*, o* x*-ésimo primo (com* $p\,(0) = 0$*);*
- $(x)_y$*, o* y*-ésimo expoente da expansão prima de* x *(*cf. *anexo A.3, p.252);*
- *A função característica de «*$x \neq 2$*»;*
- *A função característica de «*$x \neq y$*»;*
- *A função característica de «*$x|y$*»,* i.e.*, «*x *é divisor de* y*»;*
- *A função característica de «*x *é primo»;*
- *A função característica de «*x *é cubo perfeito»,*

Exercício 2.9.8 *Construa programas URM para calcular as funções com assinatura* $\mathbb{Z} \to \mathbb{Z}$*:*

- $f(x) = -x$
- $f(x) = -2x$;
- $f(x, y) = x - y$;
- *A função característica de «$x < 0$»;*
- *A função característica de «x é ímpar».*

Exercício 2.9.9 *Determine os códigos das seguintes instruções:*

$$\begin{array}{llll} \mathtt{Z}(1), & \mathtt{S}(1), & \mathtt{Z}(3), & \mathtt{S}(3), \\ \mathtt{T}(1,4) & \mathtt{T}(2,4), & \mathtt{T}(3,4), & \mathtt{J}(1,1,3), \\ \mathtt{J}(1,2,3), & \mathtt{J}(3,2,5), & \mathtt{J}(4,2,1) & \end{array}$$

Exercício 2.9.10 *Codifique os seguintes programas URM:*

- $[\mathtt{Z}(1)]$;
- $[\mathtt{J}(1,2,3)]$;
- $[\mathtt{J}(1,2,3), \mathtt{S}(1)]$;
- $[\mathtt{J}(1,2,3), \mathtt{Z}(1), \mathtt{S}(1)]$;
- $[\mathtt{Z}(1), \mathtt{J}(1,2,3), \mathtt{S}(1)]$;
- $[\mathtt{T}(1,4), \mathtt{S}(4), \mathtt{S}(3), \mathtt{J}(4,2,1)]$;
- $\mathtt{Soma} = [\mathtt{J}(3,2,5), \mathtt{S}(1), \mathtt{S}(3), \mathtt{J}(1,1,1)]$.

Exercício 2.9.11 *Encontre os respectivos programas:*

$$\begin{array}{lll} \mathsf{dec}(0), & \mathsf{dec}(1), & \mathsf{dec}(64) \\ \mathsf{dec}(128), & \mathsf{dec}(256), & \mathsf{dec}(255) \\ \mathsf{dec}(225), & \mathsf{dec}(522), & \mathsf{dec}(1025) \end{array}$$

Entre Capítulos

Apresentámos a URM, um modelo de computação baseado numa abordagem operacional, *i.e.*, a computação vista como uma sequência de passos simples e bem definidos por programas. Esta abordagem é a mesma da Máquina de Turing. Enquanto nesta é a estrutura de controlo que

define o programa a executar, a URM pode executar programas diferentes sem modificar a sua estrutura interna. Ambas as máquinas utilizam um espaço de memória ilimitado mas sempre finito (lembremo-nos que uma computação usa apenas recursos finitos). Pode-se provar que as duas máquinas são equivalentes, ou seja, o conjunto de funções que ambas calculam é o mesmo (porém, não demonstraremos esta equivalência).

Após definirmos o modelo URM explorámos algumas consequências deste sistema como modelo da computação. Falámos de funções (totais e parciais) computáveis, de predicados decidíveis e semi-decidíveis, bem como de linguagens recursivas e recursivamente enumeráveis. Mostrámos que o conjunto de programas possíveis é enumerável, *i.e.*, existem tantos programas quantos os números naturais. Também explorámos a ligação entre linguagens e predicados e como uma linguagem pode ser um oráculo da URMO, uma variante da URM. Falámos da existência e da computabilidade de funções e programas universais, capazes de simular qualquer outra função computável. E talvez o mais relevante, discutimos a Tese de Church, que defende que algoritmia e computabilidade (definida nos termos destas máquinas) coincidem.

No próximo capítulo apresentamos uma abordagem diferente à computação. Esta será analisada do ponto de vista axiomático, *i.e.*, uma função é computável se for um teorema de um certo sistema que explicita um conjunto de axiomas elementares e um conjunto de regras de construção. Veremos que esta definição de computabilidade é equivalente à URM, reforçando, assim, a verosimilhança da Tese de Church.

Capítulo 3

Computação com Funções

Quando se discute computação, a principal interpretação dada ao termo é «o resultado de um processo mecânico e sequencial de execução de instruções», com vista à obtenção de um resultado final. Assim, uma prova da computabilidade de uma função é feita apresentando-se um programa que a calcule. Esta *abordagem operacional* foi discutida no capítulo 2 onde se apresentou a URM. Existem, porém, outras abordagens. Neste capítulo falaremos de uma *abordagem axiomática* às funções computáveis.

3.1 Geração de Funções Computáveis

Um **sistema axiomático** define um conjunto de palavras a partir de:

1. um conjunto de princípios elementares, designados por **axiomas**, e

2. um conjunto de **regras de construção** que produzem **teoremas** a partir de verdades conhecidas (sejam axiomas ou teoremas).

Um sistema axiomático baseia-se num processo indutivo, onde os axiomas são a «base» e as regras definem os «passos» da indução. Esta noção de processo indutivo é também aplicada na definição de uma linguagem a

partir de uma gramática. Existem «símbolos terminais» que fazem parte da linguagem (os axiomas) e «regras de construção» de novas palavras a partir de palavras na linguagem. Na gramática

$$\begin{aligned} A &= BA|\varepsilon \\ B &= a|b|c \end{aligned}$$

existem três símbolos terminais (a, b e c) e as regras definem novas palavras da linguagem à custa de outras palavras, previamente aceites (ε é o símbolo da palavra vazia). Neste exemplo a gramática dada define a linguagem $(a|b|c)*$, *i.e.*, o conjunto de palavras formadas por zero ou mais símbolos a, b ou c. Cada palavra (ou teorema) dessa linguagem pode ser deduzida por aplicação das regras enunciadas.

O primeiro objectivo deste capítulo é apresentar um sistema axiomático capaz de gerar todas as funções computáveis (e mais nenhuma), tal como as regras da gramática anterior definem todas as palavras de uma certa linguagem.

Quais são os ***axiomas*** *desse sistema?*

Definição 3.1.1 (Axiomas das funções primitivas recursivas) *As seguintes funções são* ***primitivas recursivas****:*

- *As constantes naturais*

$$0, 1, 2, \ldots \tag{3.1}$$

- *A função Zero*

$$\begin{array}{rccc} \mathsf{zero}: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & 0 \end{array} \tag{3.2}$$

- *A função Sucessor*

$$\begin{array}{rccc} \mathsf{succ}: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & x+1 \end{array} \tag{3.3}$$

- *As funções Projecção*

$$\begin{array}{rccc} \mathsf{proj}_i^n : & \mathbb{N}^n & \to & \mathbb{N} \\ & (x_1, \ldots, x_n) & \mapsto & x_i \end{array} \tag{3.4}$$

As constantes são **funções nulárias**, *i.e*, com aridade zero. Veremos que, juntamente com a função zero, podem definir-se à custa da constante 0, mas incluí-mo-las como axiomas para simplificar os cálculos.

Vamos agora garantir que estes axiomas são computáveis, no sentido operacional, mostrando programas URM que os computem.

Lema 3.1.2 (Computabilidade dos axiomas) *As seguintes funções são computáveis:*

1. *cada constante* $n \geq 0$*;*
2. *a função* zero*;*
3. *a função* succ*;*
4. *cada projecção* proj_i^n*.*

Demonstração. Para demonstrar que uma função é computável, mostramos um programa URM que a calcule. Assim temos:

1. $\mathtt{C_n} = [\mathtt{Z}(1), \mathtt{S}(1), \ldots, \mathtt{S}(1)]$, com n repetições de $\mathtt{S}(1)$;
2. $[\mathtt{Z}(1)]$;
3. $[\mathtt{S}(1)]$;
4. $[\mathtt{T}(i, 1)]$.

□

No resto do capítulo, denotamos estas funções especiais por n, Z, S e P_i^n. O termo P_i^n, mais que uma função, representa um conjunto de funções em que n e i são parâmetros inteiros positivos. Por exemplo, $\mathsf{P}_1^3(x, y, z) = x$, $\mathsf{P}_2^3(x, y, z) = y$, $\mathsf{P}_3^3(x, y, z) = z$.

Quais são as ***regras de construção****?* A primeira que veremos designa-se por **composição** ou **substituição**:

Teorema 3.1.3 (Composição) *Sejam $f(y_1, \ldots, y_k)$ e $g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x})$ funções computáveis (com $\boldsymbol{x} = x_1, \ldots, x_n$). Então a função*

$$h(\boldsymbol{x}) = f(g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x}))$$

também é computável. Neste caso escrevemos

$$h = \mathsf{C}[f; g_1, \ldots, g_k]. \tag{3.5}$$

Demonstração. Sejam F e $G_1, \ldots, G_k$ os programas URM que calculam, respectivamente, f e $g_1, \ldots, g_k$ (sabemos que existem porque partimos do pressuposto que estas funções são computáveis). Para provar que a função h é computável, é necessário encontrar um programa, H, que a calcule.

Este programa deve começar por executar $G_1, \ldots, G_k$ e armazenar os resultados para a computação de F. Isto pode ser feito usando módulos URM:

$$H = \left[\begin{array}{l} G_1[1, \ldots, n \to n+1] \\ \vdots \\ G_k[1, \ldots, n \to n+k] \\ F[n+1, \ldots, n+k \to 1] \end{array}\right] \tag{3.6}$$

Expandindo H (cf. a secção 2.3) obtemos um programa URM que computa a função h. □

Devemos notar que a função h é total se todas as funções $g_1, \ldots, g_k$ forem totais e $f(g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x}))$ estiver definida para qualquer $\boldsymbol{x}$ — não é necessário que f seja total, bastando estar definida nas imagens $g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x})$.

Supondo que as funções soma e monus são computáveis (o que iremos confirmar adiante), vejamos alguns exemplos que usam a regra da composição:

- Aplicando o teorema 3.1.3, podemos concluir que a função $h(x, y, z) = z + 1$ é computável. Como? Sejam $f = \mathsf{S}$ e $g_1 = \mathsf{P}^3_3$. Obtém-se h pela composição $f \circ g_1$:

$$h = \mathsf{C}\left[\mathsf{S}; \mathsf{P}^3_3\right].$$

- A função $h(x,y,z) = x+y+z$ é computável porque

$$h = \mathsf{C}\left[\mathsf{soma}; \mathsf{C}\left[\mathsf{soma}; \mathsf{P}_1^3, \mathsf{P}_2^3\right], \mathsf{P}_3^3\right].$$

  Isto equivale a dizer que $x+y+z = (x+y)+z$. (**exercício:** Como seria expressa a igualdade $x+y+z = x+(y+z)$?)

- A função $\mathsf{abs}(x,y) = |x-y|$ também é computável porque

$$|x-y| = \mathsf{monus}(x,y) + \mathsf{monus}(y,x).$$

  Portanto

$$\mathsf{abs} \;=\; \mathsf{C}\left[\mathsf{soma}; \mathsf{C}\left[\mathsf{monus}; \mathsf{P}_1^2, \mathsf{P}_2^2\right], \mathsf{C}\left[\mathsf{monus}; \mathsf{P}_2^2, \mathsf{P}_1^2\right]\right] \quad (3.7)$$

- Uma função definida por casos é computável se os seus ramos e os respectivos predicados forem computáveis e mutuamente exclusivos (*i.e.*, para cada argumento da função principal, exactamente um dos predicados é verdadeiro). Esta conclusão resulta de podermos definir a função por ramos

$$h(x) = \begin{cases} f_1(x) & \text{se } \phi_1(x) \\ f_2(x) & \text{se } \phi_2(x) \\ \cdots & \\ f_n(x) & \text{se } \phi_n(x) \end{cases}$$

  pela igualdade (sem ramos)

$$h(x) \;=\; \mathsf{c}_{\phi_1}(x) \times f_1(x) + \cdots + \mathsf{c}_{\phi_n}(x) \times f_n(x). \quad (3.8)$$

  Se as funções f_i e os predicados ϕ_i forem computáveis esta expressão define uma função computável através da composição de somas e produtos de funções computáveis (lembrar que c_{ϕ_i} é a função característica do predicado ϕ_i).

Outra propriedade útil da composição é que permite obter funções por recombinação dos argumentos. É isso que nos informa o seguinte corolário:

Corolário 3.1.4 (Recombinação dos argumentos) *Supondo que a função $f(y_1, \ldots, y_n)$ é computável e que $x_{i1}, \ldots, x_{in}$ é uma sequência composta por elementos de $x_1, \ldots, x_k$ (eventualmente repetidos), então a função*

$$h(x_1, \ldots, x_k) = f(x_{i1}, \ldots, x_{in})$$

é computável.

Demonstração. Basta observar que podemos definir

$$h = \mathsf{C}\left[f; \mathsf{P}^k_{x_1}, \ldots, \mathsf{P}^k_{x_n}\right].$$

□

Vejamos alguns exemplos deste corolário. Supondo que f é computável,

Rearranjo: $h(x, y) = f(y, x)$ é computável porque $h = \mathsf{C}[f; \mathsf{P}^2_2, \mathsf{P}^2_1]$
Identificação: $h(x) = f(x, x)$ é computável porque $h = \mathsf{C}[f; \mathsf{P}^1_1, \mathsf{P}^1_1]$
Adição: $h(x, y, z) = f(x, y)$ é computável porque $h = \mathsf{C}[f; \mathsf{P}^3_1, \mathsf{P}^3_2]$

A próxima regra de construção é designada por **recursão** (ou **recorrência**):

Teorema 3.1.5 (Recursão) *Sejam $f : \mathbb{N}^n \to \mathbb{N}$ e $g : \mathbb{N}^{n+2} \to \mathbb{N}$ duas funções computáveis. Então a função $h : \mathbb{N}^{n+1} \to \mathbb{N}$ definida para cada $(\boldsymbol{x}, y)$ (com $\boldsymbol{x} = x_1, \ldots, x_n$) por:*

$$h(\boldsymbol{x}, 0) = f(\boldsymbol{x}) \tag{3.9}$$
$$h(\boldsymbol{x}, y+1) = g(\boldsymbol{x}, y, h(\boldsymbol{x}, y)) \tag{3.10}$$

é computável.

Se h for definida por recorrência sobre f e g escrevemos

$$h = \mathsf{R}[f; g]. \tag{3.11}$$

Demonstração. Sejam F e G os programas URM que calculam f e g. Para provar que a função h é computável basta encontrar um programa H que a calcule.

Para calcular o resultado de H com argumentos $x_1, \ldots, x_n, y$ é necessário considerar o valor de y. Se $y = 0$ então $F(\boldsymbol{x})$ dá-nos o resultado final (equação 3.9). Se for $y > 0$ temos de construir uma «torre» de valores intermédios. Na base está $h(\boldsymbol{x}, 0) = f(\boldsymbol{x})$ e cada andar é construído sobre o anterior usando a função g. Para isso executa-se G com argumentos $x_1, \ldots, x_n$ e o resultado intermédio anterior (equação 3.10), até que a «torre» de valores intermédios tenha a altura desejada (y).

Assim, o funcionamento de H assemelha-se a um ciclo (com uma variável de progresso i) que executa o módulo apropriado (na primeira vez F e, nas vezes seguintes, G) até que $y = i$. Em pseudo-código:

```
valor ← f(x)                  # calcula o resultado de F(x)
i ← 0                         # inicializa a variável de progresso
enquanto y ≠ i :
    valor ← g(x, i, valor)    # o próximo resultado depende do anterior
    i ← i + 1                 # incrementa a variável de progresso
retorna valor
```

Por exemplo, para calcular $h(\boldsymbol{x}, 3)$, começamos por calcular $h(\boldsymbol{x}, 0) = f(\boldsymbol{x})$, $h(\boldsymbol{x}, 1) = g(\boldsymbol{x}, 1, h(\boldsymbol{x}, 0))$ e $h(\boldsymbol{x}, 2) = g(\boldsymbol{x}, 2, h(\boldsymbol{x}, 1))$.

Resta-nos descrever um programa adequado para H. Como o argumento inicial ocupa os registos $R_1, \ldots, R_{n+1}$, colocamos a variável de progresso em R_{n+2} e o resultado intermédio em R_{n+3}, no seguinte estado típico:

R_1	$\cdots$	R_n	R_{n+1}	R_{n+2}	R_{n+3}	$\cdots$
x_1	$\cdots$	x_n	y	i	$valor$	$\cdots$

Deste modo, o programa é dado por

$$H = \left[\begin{array}{l} F\,[1,\ldots,n \to n+3] \\ \hline \mathtt{J}\,(n+1,n+2,6) \\ G\,[1,\ldots,n,n+2,n+3 \to n+3] \\ \mathtt{S}\,(n+2) \\ \mathtt{J}\,(1,1,2) \\ \hline \mathtt{T}\,(n+3,1) \end{array}\right]$$

□

A demonstração anterior permite-nos concluir também que se as funções f e g forem totais, a função obtida por recursão a partir de f e g, $\mathsf{R}[f;g]$, também é total.

Com a recursão é possível definir uma variedade de funções muito conhecidas. Vejamos alguns exemplos:

Soma A função $\mathsf{soma}\,(x,y) = x+y$ é computável por recursão. *Como verificar esta afirmação?* Se aplicarmos a recorrência sobre o argumento $y \geq 0$ obtemos

$$\left\{\begin{array}{rcl} \mathsf{soma}\,(x,0) & = & x \\ \mathsf{soma}\,(x,y+1) & = & \mathsf{soma}\,(x,y)+1 \end{array}\right. \tag{3.12}$$

Isto significa que a função soma será computável se as funções

$$\begin{array}{rcl} f\,(x) & = & x \\ g\,(x,y,z) & = & z+1 \end{array}$$

forem computáveis. Porquê $g\,(x,y,z) = z+1$? A função g parece ter demasiados argumentos, mas é importante pensar que o método da recursão deve ter a maior generalidade possível. Assim, o passo g, *no caso mais geral*, pode usar os argumentos originais (x e y) bem como o valor anterior de h — que corresponde à iteração anterior no pseudo-código na demonstração do teorema 3.1.5, sobre a recursão.

Como se verifica em 3.12, a função apenas necessita do valor anterior da soma (ou seja, o terceiro argumento) para lhe somar 1. Para vermos como funciona, observemos o seguinte exemplo, do cálculo recursivo de $\mathsf{soma}\,(4,3)$:

$$\begin{aligned}
\mathsf{soma}\,(4,3) &= \mathsf{soma}\,(4,2)+1 \\
&= (\mathsf{soma}\,(4,1)+1)+1 \\
&= ((\mathsf{soma}\,(4,0)+1)+1)+1 \\
&= ((4+1)+1)+1 \\
&= 7
\end{aligned}$$

As funções f e g são computáveis porque

$$\begin{aligned}
f &= \mathsf{P}_1^1 \\
g &= \mathsf{C}\left[\mathsf{S};\mathsf{P}_3^3\right]
\end{aligned}$$

portanto, pelo teorema da recursão aplicado a f e g, a soma é computável:

$$\mathsf{soma} = \mathsf{R}\left[\mathsf{P}_1^1;\mathsf{C}\left[\mathsf{S};\mathsf{P}_3^3\right]\right]. \tag{3.13}$$

Produto A função $\mathsf{produto}\,(x,y) = x \times y$ também é computável por recursão. Aplicando a recorrência em y:

$$\begin{aligned}
\mathsf{produto}\,(x,0) &= 0 \\
\mathsf{produto}\,(x,y+1) &= \mathsf{produto}\,(x,y)+x
\end{aligned}$$

Temos assim que mostrar que as seguintes funções são computáveis:

$$\begin{aligned}
f\,(x) &= 0 \\
g\,(x,y,z) &= z+x
\end{aligned}$$

Antes disso observemos, uma vez, mais, a recursão em acção:

$$\begin{aligned}
\mathsf{produto}\,(4,3) &= \mathsf{produto}\,(4,2)+4\\
&= (\mathsf{produto}\,(4,1)+4)+4\\
&= ((\mathsf{produto}\,(4,0)+4)+4)+4\\
&= ((0+4)+4)+4\\
&= 12
\end{aligned}$$

Como podemos escrever

$$\begin{aligned}
f &= \mathsf{Z}\\
g &= \mathsf{C}\left[\mathsf{soma};\mathsf{P}^3_3,\mathsf{P}^3_1\right]
\end{aligned}$$

concluímos que f e g são computáveis. Pelo teorema da recursão, a função $\mathsf{produto}$ é computável:

$$\mathsf{produto} = \mathsf{R}\left[\mathsf{Z};\mathsf{C}\left[\mathsf{soma};\mathsf{P}^3_3,\mathsf{P}^3_1\right]\right]. \tag{3.14}$$

Factorial A função factorial $\mathsf{fact}\,(x) = x!$ é computável por recursão. Aplicando a recorrência a x:

$$\begin{aligned}
\mathsf{fact}\,(0) &= 1\\
\mathsf{fact}\,(x+1) &= (x+1)\times\mathsf{fact}\,(x)
\end{aligned}$$

Vejamos a recursão:

$$\begin{aligned}
\mathsf{fact}\,(4) &= 4\times\mathsf{fact}\,(3)\\
&= 4\times(3\times\mathsf{fact}\,(2))\\
&= 4\times(3\times(2\times\mathsf{fact}\,(1)))\\
&= 4\times(3\times(2\times(1\times\mathsf{fact}\,(0))))\\
&= 4\times(3\times(2\times(1\times 1)))\\
&= 24
\end{aligned}$$

Temos assim que mostrar que são computáveis:

$$\begin{aligned} f &= 1 \\ g(x,z) &= x \times z \end{aligned}$$

Neste caso, f é a função nulária, ou constante, 1 (qualquer número natural é trivialmente computável). Já a função g é computável porque é composição dos seus dois argumentos sobre a função produto, *i.e.*, $g = \mathsf{C}[\mathsf{produto}; \mathsf{P}^2_1, \mathsf{P}^2_2]$. Logo, o factorial é computável:

$$\mathsf{fact} = \mathsf{R}\left[1; \mathsf{C}\left[\mathsf{produto}; \mathsf{P}^2_1, \mathsf{P}^2_2\right]\right]. \tag{3.15}$$

Predecessor A função predecessor, **pred**, calcula o número anterior ao argumento dado. Esta função pode também ser definida por recursão da seguinte forma: supondo que queremos calcular o predecessor de x, se soubermos o predecessor de $x-1$, basta somar-lhe 1 para encontrar o valor que procuramos. Por exemplo, se quisermos encontrar $\mathsf{pred}\,(8)$ e já soubermos que $\mathsf{pred}\,(7) = 6$, então $\mathsf{pred}\,(8) = \mathsf{pred}\,(7) + 1 = 7$. Isto dá-nos uma pista para a definição recursiva. A base da recursão será o predecessor do valor mais pequeno, $\mathsf{pred}\,(0)$, que é, por convenção, 0 (dado não admitirmos valores negativos e queremos manter as nossas funções como funções totais). Assim:

$$\begin{aligned} \mathsf{pred}\,(0) &= 0 \\ \mathsf{pred}\,(x+1) &= \mathsf{pred}\,(x) + 1 = x \end{aligned}$$

e temos de mostrar que são computáveis:

$$\begin{aligned} f &= 0 \\ g(x,z) &= x \end{aligned}$$

Ora, a função f é um número natural e g é a projecção P^2_1. Deste modo, o predecessor é computável:

$$\mathsf{pred} = \mathsf{R}\left[0; \mathsf{P}^2_1\right]. \tag{3.16}$$

Monus A função monus, como referido no capítulo anterior, é a subtracção natural, ou seja, funciona como a subtracção excepto quando a subtracção normal é negativa, onde monus vale zero.

$$\mathsf{monus}\,(x,y) \;=\; \begin{cases} x-y & \text{se } x>y \\ 0 & \text{caso contrário} \end{cases}$$

É comum usar-se o símbolo «$\dot{-}$» para representar a subtracção natural. Também usaremos «$x \dot{-} 1$» para representar o predecessor e «$x \dot{-} y$» para a função monus.

Esta função pode ser igualmente definida por recursão. Para isso, aplicamos a base e o passo da recursão ao segundo argumento, y:

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= (x \dot{-} y) \dot{-} 1 \end{aligned}$$

Assim:

$$\begin{aligned} f\,(x) &= x \\ g\,(x,y,z) &= z \dot{-} 1 \end{aligned}$$

A base f é a projecção P^1_1 e o passo g é dado pela expressão $\mathsf{C}[\mathsf{pred};\mathsf{P}^3_3]$, logo a função monus é computável:

$$\mathsf{monus} \;=\; \mathsf{R}\left[\mathsf{P}^1_1;\mathsf{C}\left[\mathsf{pred};\mathsf{P}^3_3\right]\right]. \tag{3.17}$$

Sinal A função **sg** (sinal)

$$\mathsf{sg}\,(x) \;=\; \begin{cases} 0 & \text{se } x=0 \\ 1 & \text{caso contrário} \end{cases}$$

é computável por recursão. Aplicando a recorrência a x obtemos

$$\begin{aligned} \mathsf{sg}\,(0) &= 0 \\ \mathsf{sg}\,(x+1) &= 1 \end{aligned}$$

Temos assim de mostrar que são computáveis:

$$\begin{aligned} f &= 0 \\ g(x,z) &= 1 \end{aligned}$$

A função f é um natural enquanto que podemos definir g, por exemplo, por $g = \mathsf{C}[\mathsf{C}[\mathsf{S};\mathsf{Z}];\mathsf{P}^2_1]$. A função sinal é computável porque

$$\mathsf{sg} = \mathsf{R}[0;g]\,. \tag{3.18}$$

Sinal Inverso A função $\overline{\mathsf{sg}}$

$$\overline{\mathsf{sg}}(x) = \begin{cases} 1 & \text{se } x = 0 \\ 0 & \text{caso contrário} \end{cases}$$

é computável porque

$$\overline{\mathsf{sg}} = \mathsf{C}\left[\mathsf{monus};\mathsf{C}[\mathsf{S};\mathsf{Z}],\mathsf{C}\left[\mathsf{sg};\mathsf{P}^1_1\right]\right]. \tag{3.19}$$

Relações e operadores lógicos As funções de sinal, sg e $\overline{\mathsf{sg}}$ podem ser usadas, conjuntamente com as funções pred e monus, para exprimir relações entre naturais assumindo a convenção (usada nas funções características) que o valor lógico verdadeiro é representado pelo número 1, e falso pelo 0.

Por exemplo, para exprimir «$x > y$» observamos que

$$x > y \;\Leftrightarrow\; x \ominus y > 0 \;\Leftrightarrow\; \mathsf{sg}(x \ominus y) = 1 \tag{3.20}$$

Outras relações também podem ser descritas:

$$x \geq y \;\Leftrightarrow\; x > y \ominus 1 \tag{3.21}$$
$$x = y \;\Leftrightarrow\; x \geq y \wedge y \geq x \tag{3.22}$$
$$x \neq y \;\Leftrightarrow\; x > y \vee y > x \tag{3.23}$$

Já os operadores lógicos da conjunção ($\wedge$) e disjunção ($\vee$) usam a soma e o produto:

$$x \wedge y \Leftrightarrow \mathsf{sg}(x) \times \mathsf{sg}(y) \tag{3.24}$$
$$x \vee y \Leftrightarrow \mathsf{sg}(\mathsf{sg}(x) + \mathsf{sg}(y)) \tag{3.25}$$

Também a negação ($\neg$) fica

$$\neg x \Leftrightarrow \overline{\mathsf{sg}}(x) \tag{3.26}$$

Desta forma podemos usar os operadores lógicos como abreviaturas das respectivas operações computáveis entre naturais.

Vejamos agora um outro resultado que pode ser obtido com o uso da recursão.

Corolário 3.1.6 (Somas e Produtos limitados) *Seja $h(\boldsymbol{x}, z)$ uma função computável (com $\boldsymbol{x} = x_1, \ldots, x_n$). Para cada $y \in \mathbb{N}$, as funções* ***soma limitada*** *e* ***produto limitado*** *dadas, respectivamente, pelas expressões*

$$\begin{aligned} s(\boldsymbol{x}, y) &= h(\boldsymbol{x}, 0) + h(\boldsymbol{x}, 1) + \ldots + h(\boldsymbol{x}, y-1) \\ p(\boldsymbol{x}, y) &= h(\boldsymbol{x}, 0) \times h(\boldsymbol{x}, 1) \cdot \ldots \times h(\boldsymbol{x}, y-1) \end{aligned}$$

são computáveis. Usamos a seguinte notação

$$\Sigma_{z<y} h(\boldsymbol{x}, z) = h(\boldsymbol{x}, 0) + h(\boldsymbol{x}, 1) + \ldots + h(\boldsymbol{x}, y-1) \tag{3.27}$$
$$\Pi_{z<y} h(\boldsymbol{x}, z) = h(\boldsymbol{x}, 0) \times h(\boldsymbol{x}, 1) \cdot \ldots \times h(\boldsymbol{x}, y-1) \tag{3.28}$$

Demonstração. Mostramos, por recursão, que as funções s e p são computáveis. Para s, aplicando a recorrência a y:

$$\begin{aligned} s(\boldsymbol{x}, 0) &= 0 \\ s(\boldsymbol{x}, y+1) &= h(\boldsymbol{x}, y) + s(\boldsymbol{x}, y) \end{aligned}$$

o que resulta nas funções

$$\begin{aligned} f_1(\boldsymbol{x}) &= 0 \\ g_1(\boldsymbol{x}, y, z) &= h(\boldsymbol{x}, y) + z \end{aligned}$$

que são computáveis.

Para o produto, aplicando a recorrência igualmente a y:

$$\begin{aligned} p(\boldsymbol{x}, 0) &= 1 \\ p(\boldsymbol{x}, y+1) &= h(\boldsymbol{x}, y) \times p(\boldsymbol{x}, y) \end{aligned}$$

o que resulta nas funções

$$\begin{aligned} f_2(\boldsymbol{x}) &= 1 \\ g_2(\boldsymbol{x}, y, z) &= h(\boldsymbol{x}, y) \times z \end{aligned}$$

que também são computáveis. □

Podemos usar este corolário para nos ajudar a obter funções mais complexas. Por exemplo, seja $\mathsf{D}(x)$ o número de divisores de x (por convenção, $\mathsf{D}(0) = 1$). Assim, $\mathsf{D}(1) = 1, \mathsf{D}(2) = 2, \mathsf{D}(3) = 2, \mathsf{D}(4) = 3 \ldots$ Para calcular este valor, verifica-se para cada natural i (desde 1 até x) se é, ou não, divisor de x, *i.e.*, se $\mathsf{rm}(i, x) = 0$. Podemos agora calcular $\mathsf{D}(x)$ com a soma limitada, contando 1 para cada divisor e 0 para cada não divisor; se aplicarmos o resultado de rm (cf. 3.4, p.123) à função $\overline{\mathsf{sg}}$ obtemos

$$\mathsf{D}(x) = \Sigma_{i<x+1} \overline{\mathsf{sg}}(\mathsf{rm}(i, x)). \tag{3.29}$$

Por exemplo, $\mathsf{D}(4)$ resulta do somatório:

$$\begin{aligned} \mathsf{D}(4) &= \Sigma_{i<5} \overline{\mathsf{sg}}(\mathsf{rm}(i, 4)) \\ &= \overline{\mathsf{sg}}(\mathsf{rm}(0, 4)) + \overline{\mathsf{sg}}(\mathsf{rm}(1, 4)) + \overline{\mathsf{sg}}(\mathsf{rm}(2, 4)) + \\ &\quad \overline{\mathsf{sg}}(\mathsf{rm}(3, 4)) + \overline{\mathsf{sg}}(\mathsf{rm}(4, 4)) \\ &= \overline{\mathsf{sg}}(4) + \overline{\mathsf{sg}}(0) + \overline{\mathsf{sg}}(0) + \overline{\mathsf{sg}}(1) + \overline{\mathsf{sg}}(0) \\ &= 0 + 1 + 1 + 0 + 1 \\ &= 3 \end{aligned}$$

Os divisores de 4 são $1, 2$ e 4, portanto confirma-se que 4 tem três divisores.

Com D podemos definir o predicado «primo» que verifica se um número é primo (ou seja, se tem somente dois divisores, o 1 e o próprio número),

$$\mathsf{primo}\,(x) \quad = \quad \overline{\mathsf{sg}}\,(|\mathsf{D}\,(x) - 2|)\,. \tag{3.30}$$

Outra operação útil na criação de novas funções é a *minimização limitada*. Esta operação determina o menor valor que anula uma dada função, até um limite máximo, digamos y. Se a função dada não tiver um zero em $0, \ldots, y-1$, o resultado da operação é y.

Corolário 3.1.7 (Minimização Limitada) *Seja $h\,(\boldsymbol{x}, z)$ uma função computável (com $\boldsymbol{x} = x_1, \ldots, x_n$). Para cada $y \in \mathbb{N}$, a* ***minimização limitada*** *de h, dada pela expressão*

$$\mu_{z<y}\,[h(\boldsymbol{x}, z)] \quad = \quad \begin{cases} \mathsf{min}\{z | h(\boldsymbol{x}, z) = 0\} & \exists z < y : h(\boldsymbol{x}, z) = 0 \\ y & \text{c.c.} \end{cases} \tag{3.31}$$

é computável.

Demonstração. Sejam

$$z_0 = \mu_{z<y}\,[h\,(\boldsymbol{x}, z)]$$

e

$$j\,(\boldsymbol{x}, v) = \Pi_{u \leq v} \mathsf{sg}\,(h\,(\boldsymbol{x}, u))\,,$$

computável (pelo corolário 3.1.6 e porque sg também é computável).

A função j é um produto de 0s e 1s, consoante o valor de h é nulo ou positivo, respectivamente. Isto é, a função j vale 1 até ao primeiro valor que anula h (que é z_0); daí em diante j vale 0.

Assim, z_0 é igual ao número de vs menores que y tais que $j\,(\boldsymbol{x}, v) = 1$:

$$z_0 = \Sigma_{v<y} j\,(\boldsymbol{x}, v)\,.$$

Em resumo, podemos escrever

$$\mu_{z<y}\left[h\left(\boldsymbol{x},z\right)\right] \;=\; \Sigma_{v<y}\left(\Pi_{u\leq v}\mathsf{sg}\left(h\left(\boldsymbol{x},u\right)\right)\right). \tag{3.32}$$

Esta igualdade mostra-nos que a função $(\boldsymbol{x},y)\mapsto\mu_{z<y}\left[h\left(\boldsymbol{x},z\right)\right]$ é computável, usando a composição de funções computáveis e o corolário 3.1.6. □

É possível aplicar a composição para generalizar o limite y na soma, produto e minimização limitadas, substituindo y por uma função computável. Assim, dadas as funções computáveis $h\left(\boldsymbol{x},z\right)$ e $f\left(\boldsymbol{x},w\right)$, também são computáveis:

$$\begin{gathered}\Sigma_{z<f(\boldsymbol{x},w)}h\left(\boldsymbol{x},z\right)\\ \Pi_{z<f(\boldsymbol{x},w)}h\left(\boldsymbol{x},z\right)\\ \mu_{z<f(\boldsymbol{x},w)}\left[h\left(\boldsymbol{x},z\right)\right]\end{gathered}$$

Analisemos agora uma função definida por minimização limitada. Seja p a função que calcula o x-ésimo primo, convencionando que $\mathsf{p}\left(0\right)=0$ (conforme A.3, p.252, temos $\mathsf{p}\left(n\right)=p_n$ para $n>0$). Assim, $\mathsf{p}\left(1\right)=2,\mathsf{p}\left(2\right)=3,\mathsf{p}\left(3\right)=5,\mathsf{p}\left(4\right)=7,\ldots$ Esta função é computável e pode ser definida por minimização limitada e recursão:

$$\begin{aligned}\mathsf{p}\left(0\right) &= 0\\ \mathsf{p}\left(x+1\right) &= \mu_{i<p(x)!+2}\left[\overline{\mathsf{sg}}\left(\mathsf{primo}\left(i\right)\wedge\left(i>\mathsf{p}\left(x\right)\right)\right)\right]\end{aligned}$$

Como funciona esta função? Para encontrar o $(x+1)$-ésimo primo, procuramos o primeiro número primo i que seja maior que o x-ésimo primo. Esta condição é expressa por «$\mathsf{primo}\left(i\right)\wedge\left(i>\mathsf{p}\left(x\right)\right)$» e invertemos o sinal com a função «$\overline{\mathsf{sg}}$» dado que a minimização limitada «procura» o valor 0, neste caso da condição dada. O limite da minimização é dado por $\mathsf{p}\left(x\right)!+2$, valor suficientemente grande para majorar $\mathsf{p}\left(x+1\right)$. Temos assim que mostrar que são computáveis:

$$\begin{aligned}f &= 0\\ g\left(x,z\right) &= \mu_{i<z!+2}\left[\overline{\mathsf{sg}}\left(\mathsf{primo}\left(i\right)\wedge\left(i>\mathsf{p}\left(x\right)\right)\right)\right]\end{aligned}$$

Já sabemos que f é computável e basta constatar que g resulta da minimização limitada de uma função definida por composições de funções computáveis. Também o limite $z!+2$ da minimização pode ser obtido por composição de funções computáveis.

Outro exemplo. Como referido no anexo A.3, p.253, a expansão prima de um número n é a sua decomposição num produto de primos (que é única para cada n). Por exemplo, $3024 = 2^4 \times 3^3 \times 7^1$. Seja $(x)_y$ o expoente do primo $\mathsf{p}(y)$ na expansão prima de x. No caso do exemplo dado, $(3024)_1 = 4$, $(3024)_2 = 3$, $(3024)_3 = 0$, $(3024)_4 = 1$, $(3024)_5 = 0\ldots$ (por convenção, se $x = 0$ ou $y = 0$, $(x)_y = 0$).

Esta função é computável pela expressão

$$(x)_y \quad = \quad \mu_{z<x}\left[\overline{\mathsf{sg}}\left(\mathsf{rm}\left(\mathsf{p}(y)^{z+1}, x\right)\right)\right] \tag{3.33}$$

Ou seja, o resultado da expressão é o menor expoente z tal que $\mathsf{p}(y)^z$ divide x mas $\mathsf{p}(y)^{z+1}$ já não divide x. Se $\mathsf{p}(y)$ não dividir x então $(x)_y = 0$, que é o valor correcto para esses casos.

Esta função é útil para a seguinte codificação de tuplos n-ários[1]

$$\alpha_n(x_1, \ldots, x_n) = \mathsf{p}(1)^{x_1+1} \times \mathsf{p}(2)^{x_2+1} \times \cdots \times \mathsf{p}(n)^{x_n+1} \tag{3.34}$$

sendo uma generalização do código 2.33, p.38.

Dado um número m neste código, obtém-se a aridade do respectivo tuplo pela expressão $\mu_{z<m}\left[(m)_{z+1}\right]$ e o argumento x_i é dado por $\mathsf{monus}\left((m)_i, 1\right)$.

Ainda outro exemplo. Seja φ um qualquer predicado decidível. A função

$$\mu_{z<y}\left[\varphi(\boldsymbol{x}, z)\right] = \begin{cases} \mathsf{min}\left\{z \mid \varphi(\boldsymbol{x}, z)\right\} & \text{se } \exists z < y : \varphi(\boldsymbol{x}, z) \\ y & \text{caso contrário} \end{cases}$$

é computável porque podemos escrever

$$\mu_{z<y}\left[\varphi(\boldsymbol{x}, z)\right] = \mu_{z<y}\left[\overline{\mathsf{sg}}\left(\mathsf{c}_\varphi(\boldsymbol{x}, z)\right)\right] \tag{3.35}$$

[1]Esta codificação é uma alternativa ao método usado pela função cod na codificação de programas.

Como funciona? Para o primeiro valor z que satisfaça o predicado φ temos, na função característica, $\mathsf{c}_{\varphi}(z) = 1$. A minimização limitada calcula o primeiro z que anula $\overline{\mathsf{sg}}(\mathsf{c}_{\varphi}(z))$. Como $\overline{\mathsf{sg}}$ transforma 1s em 0s e vice versa, esse z é o primeiro número que satisfaz φ, *i.e.*, o valor desejado.

Este último exemplo é útil para entender a **quantificação limitada** que é uma outra forma de construir predicados. Temos a quantificação universal limitada, «$\forall_{z<y}$», e a existencial limitada, «$\exists_{z<y}$»:

$$\forall_{z<y}\varphi(z) = \varphi(0) \wedge \varphi(1) \wedge \cdots \wedge \varphi(y-1) \tag{3.36}$$
$$\exists_{z<y}\varphi(z) = \varphi(0) \vee \varphi(1) \vee \cdots \vee \varphi(y-1) \tag{3.37}$$

Estas duas operações estão para os predicados como o produto e a soma limitada estão para as funções. Na quantificação universal limitada, o predicado resultado, $\forall_{z<y}\varphi(z)$, só é verdadeiro se φ for verdadeiro para todos os valores de 0 até $y-1$. Na quantificação existencial limitada, $\exists_{z<y}\varphi(z)$, este predicado é verdadeiro desde que φ seja verdadeiro para algum valor entre 0 e $y-1$.

Corolário 3.1.8 *Seja $\varphi(\boldsymbol{x}, y)$ um predicado decidível. Os seguintes predicados são decidíveis:*

1. $\mu_{z<y}[\varphi(\boldsymbol{x}, z)]$;
2. $\forall_{z<y}\varphi(\boldsymbol{x}, z)$;
3. $\exists_{z<y}\varphi(\boldsymbol{x}, z)$.

Demonstração. Estes predicados são decidíveis porque as suas funções características são dadas pelas expressões

1. $\mu_{z<y}[\overline{\mathsf{sg}}(\mathsf{c}_{\varphi}(\boldsymbol{x}, z))]$;
2. $\prod_{z<y}\mathsf{c}_{\varphi}(\boldsymbol{x}, z)$;
3. $1 \dot{-} \prod_{z<y}(1 \dot{-} \mathsf{c}_{\varphi}(\boldsymbol{x}, z))$

que estão definidas usando apenas regras e axiomas computáveis. □

Funções primitivas recursivas

Tanto a regra da composição como da recursão, se aplicadas a funções totais, produzem funções totais. Como os axiomas $n, \mathsf{Z}, \mathsf{S}$ e P_i^n são funções totais, com estas regras e axiomas só é possível definir funções totais. Designamos as funções definidas deste modo por primitivas recursivas.

Definição 3.1.9 *O conjunto das* ***funções primitivas recursivas****,* $\mathcal{PR}$*, é o mais pequeno conjunto que contém os axiomas* $n, \mathsf{Z}, \mathsf{S}$ *e* P_i^n *e que está fechado para as operações de composição e recursão.*

Qualquer função definida por uma sequência de composições e recursões a partir dos axiomas está incluída em $\mathcal{PR}$. Pelos teoremas 3.1.3 e 3.1.5 mostra-se que todas estas funções são computáveis.

Um predicado φ diz-se **primitivo recursivo** se a sua função característica pertencer a $\mathcal{PR}$. Por exemplo, a relação de igualdade entre naturais, «=», é primitiva recursiva dado que

$$\mathsf{c}_{=}(a,b) = \overline{\mathsf{sg}}\left(\mathsf{monus}(a,b) + \mathsf{monus}(b,a)\right).$$

Como cada predicado define um conjunto[2] (pela igualdade $\mathsf{L}_\varphi = \{n \mid \varphi(n)\}$), $\mathcal{PR}$ é fechado para a união, intersecção e complementos (das linguagens) de predicados primitivos recursivos (*cf.* teorema 2.2.8, p.34). Estas operações sobre conjuntos correspondem, respectivamente, à disjunção, conjunção e negação dos respectivos predicados. Resumindo, dados dois predicados φ, ψ, temos:

- $\mathsf{L}_\varphi \cup \mathsf{L}_\psi = \{x \mid \varphi(x) \vee \psi(x)\}$
- $\mathsf{L}_\varphi \cap \mathsf{L}_\psi = \{x \mid \varphi(x) \wedge \psi(x)\}$
- $\overline{\mathsf{L}_\varphi} = \{x \mid \neg\varphi(x)\}$

[2]tecnicamente, de acordo com A.2, p.246, um predicado *é* um conjunto e a notação «$\varphi(x)$» apenas abrevia «$x \in \varphi$».

Por exemplo, como «=» e «>» são predicados em $\mathcal{PR}$, também $\geq \in \mathcal{PR}$ pois $\mathsf{L}_{\geq} = \mathsf{L}_{=} \cup \mathsf{L}_{>}$.

Será que $\mathcal{PR}$ contém todas as funções computáveis? Ou seja, $\mathcal{PR} = \mathcal{C}$? Não! O conjunto $\mathcal{PR}$ não contém, por exemplo, funções parciais que sabemos ser computáveis (ω, ou a função parcial definida na página 30). Assim, temos apenas $\mathcal{PR} \subset \mathcal{C}$ mas $\mathcal{PR} \neq \mathcal{C}$.

Pior, $\mathcal{PR}$ nem sequer contém todas as funções *totais* computáveis. Este resultado não é trivial mas é possível encontrarem-se funções totais computáveis que se mostra não serem primitivas recursivas. Um exemplo é a **função de Ackermann**:

$$\mathsf{ack}\,(x,y) \;=\; \begin{cases} y+1 & \text{se } x = 0 \\ \mathsf{ack}\,(x-1,1) & \text{se } x > 0 \wedge y = 0 \\ \mathsf{ack}\,(x-1,\mathsf{ack}\,(x,y-1)) & \text{se } x > 0 \wedge y > 0 \end{cases}$$

Esta função cresce mais rapidamente que qualquer função primitiva recursiva (a demonstração desta afirmação fica fora do âmbito deste livro) mas é total e a definição acima é claramente algorítmica. Logo, pela Tese de Church, é computável.

Vejamos outro exemplo de uma função total que não pertence a $\mathcal{PR}$. Seja δ_0 a função soma e consideremos a seguinte sequência de funções, onde δ_{k+1} é definida por recursão a partir de δ_k:

$$\begin{aligned} \delta_{k+1}\,(x,0) &= x \\ \delta_{k+1}\,(x,y+1) &= \delta_k\,(x,\delta_{k+1}\,(x,y)) \end{aligned}$$

Por exemplo:

$$\begin{aligned} \delta_1\,(3,2) &= \delta_0\,(3,\delta_1\,(3,1)) \\ &= \delta_0\,(3,\delta_0\,(3,\delta_1\,(3,0))) \\ &= \delta_0\,(3,\delta_0\,(3,3)) \\ &= \delta_0\,(3,3+3) \\ &= \delta_0\,(3,6) \\ &= 3+6=9 \end{aligned}$$

Estas funções generalizam os somatórios:

$$\delta_1(x, y+1) = \underbrace{\delta_0(\cdots \delta_0(x, x)\cdots)}_{y \text{ vezes}}$$
$$\delta_2(x, y+1) = \underbrace{\delta_1(\cdots \delta_1(x, x)\cdots)}_{y \text{ vezes}}$$
$$\delta_3(x, y+1) = \underbrace{\delta_2(\cdots \delta_2(x, x)\cdots)}_{y \text{ vezes}}$$

Como $\delta_0 \in \mathcal{PR}$ e se obtêm as restantes δ_k por sucessivas aplicações da recursão, cada $\delta_k \in \mathcal{PR}$. É possível mostrar-se que δ_k cresce tão ou mais depressa que qualquer função primitiva recursiva definida por k recursões.

Agora consideremos a função total

$$\delta(x, y+1) = \underbrace{\delta_y(\cdots \delta_y(x, x)\cdots)}_{y \text{ vezes}}.$$

Esta função não só faz y repetições, como a função repetida, δ_y, depende do argumento. Fixando x, esta função cresce mais depressa que qualquer δ_k a partir de um certo valor y finito, *i.e.*, $\forall k \geq 0 : \delta_k \in O(\delta)$.

Como δ cresce mais depressa que qualquer função primitiva recursiva, esta função não pode pertencer a $\mathcal{PR}$.

Minimização

Acabámos de constatar que se pretendemos descrever axiomaticamente todas as funções computáveis (por exemplo, URM-computável ou Turing-computável) é necessário considerar outras vias além da composição e a recursão. Apresentamos, em seguida, a terceira (e última) regra de construção, designada **minimização**, com expressividade suficiente para definir as restantes funções computáveis (porém $\mathcal{PR}$ não está fechado para esta operação).

Teorema 3.1.10 (Minimização) *Seja $f(\boldsymbol{x}, y)$ uma função computável (com $\boldsymbol{x} = x_1, \ldots, x_n$). A função*

$$\mu_y [f(\boldsymbol{x}, z)] \quad = \quad \mathsf{min} \{y \mid f(\boldsymbol{x}, y) = 0 \wedge \forall z < y, f(\boldsymbol{x}, z) \downarrow\} \quad (3.38)$$

é computável.

Demonstração. Seja F o programa que computa $f(\boldsymbol{x}, y)$. O programa para calcular o resultado da minimização deve executar repetidamente F com valores y a partir de 0 até que o resultado de F seja 0. Em pseudo-código:

```
y ← 0                          # inicia a variável de progresso
enquanto f(x, y) ≠ 0 :
    y ← y + 1                  # actualiza a variável de progresso
retorna y
```

Façamos, como nos casos da composição e da recursão, um esquema de programa URM que calcule este valor. Na URM, os argumentos encontram-se nos primeiros n registos (que armazenam $x_1, \ldots, x_n$). O registo R_{n+1} irá conter a variável de progresso y e R_{n+3} o resultado da execução de F. O registo R_{n+2} manterá o valor zero para comparar com o resultado da execução de F:

$$H = \left[\begin{array}{l} F[1, \ldots, n+1 \rightarrow n+3] \\ \mathtt{J}(n+2, n+3, 5) \\ \mathtt{S}(n+1) \\ \mathtt{J}(1, 1, 1) \\ \hline \mathtt{T}(n+1, 1) \end{array}\right]$$

Se a computação de F divergir, o programa H diverge. Porém, este é o comportamento esperado para o respectivo valor da função. Como existe um programa que a calcula, a função definida pela regra da minimização é computável. □

A minimização determina o menor y que anula a função dada. Se este valor não existir, a função não está definida. Isto também pode acontecer se a função a anular não estiver definida em algum valor antes de y. Resumindo, $\mu_y [f]$ não está definida se

1. não existe y tal que $f(y) = 0$ ou
2. $f(y) = 0$ mas existe $w < y$ tal que $f(w) \uparrow$.

Esta propriedade distingue a minimização da composição e da recursão, pois permite definir funções parciais.

Um exemplo. A função $h(x) = \mu_y [|x - y^2|]$ calcula, para cada x, o primeiro y que anula $|x - y^2|$. Se x for um quadrado perfeito (digamos $|16 - y^2|$), então esta minimização calcula a raiz quadrada de x ($y = 4$). Caso contrário (por exemplo $|17 - y^2|$), não existe um natural y que anule a expressão, *i.e.*, a função é indefinida:

$$h(x) = \mu_y \left[\left|x - y^2\right|\right] \quad = \quad \begin{cases} \sqrt{x} & \text{se } x \text{ é quadrado perfeito} \\ \perp & \text{caso contrário} \end{cases}$$

Outro exemplo. Seja f uma função computável e injectiva (ver mais em A.1, p.237). Podemos usar a minimização para mostrar que a função inversa direita, g, é computável. Nesse caso o valor $x = g(y)$ é tal que $f(x) = y$. Dado um y, para encontrar computacionalmente o x que lhe corresponde por g basta testar progressivamente, para cada $x \geq 0$, se o valor $f(x)$ é igual a y. *Porquê?* Se $f(x) = y$ e $f(x') = y$ então $x = x'$ porque f é injectiva: se $f(x) = y$ então $g(y) = x$. Esta implicação garante que podemos definir

$$g(y) \quad = \quad \mu_x [|y - f(x)|] .$$

Se f não for sobrejectiva então a inversa direita, g, é parcial pois alguns y não são imagem por f: nenhum x é solução de $|y - f(x)| = 0$. Para esses y a minimização não fica definida.

A minimização permite, assim, definir mais funções computáveis do que as definidas apenas por composição e recursão. Não só funções parciais mas também algumas funções totais fora do alcance das duas regras anteriores, como a função de Ackermann e a função δ descritas na página 107.

3.2 Provas Axiomáticas

Nesta secção apresentamos um método de demonstração da computabilidade de funções. Este parte de «verdades conhecidas» (axiomas ou teoremas já provados) e, dando «passos seguros» (as regras), percorre um caminho até um dado «destino» (a função que se pretende provar que é computável).

Uma **prova axiomática** da função f consiste numa sequência de constantes ou funções $f_1, \ldots, f_m$ onde $f = f_m$. Cada função intermédia f_i é um axioma ou resulta da aplicação da composição, recursão ou minimização de funções anteriores (ou ainda porque é uma hipótese do teorema a demonstrar). Assim cada função intermédia é computável. A função f final diz-se um **teorema**.

Fixemos alguma notação que vamos usar nesta secção e seguintes:

$0, 1, 2 \ldots$	constantes
Z	o axioma «zero»
S	o axioma «sucessor»
P_i^n	os axiomas «projecção»
$\mathsf{C}[j; i_1, \ldots, i_k]$	regra da composição $f_j\,(f_{i_1}, \ldots, f_{i_k})$
$\mathsf{R}[i; j]$	regra da recursão com base f_i e passo f_j
$\mathsf{M}[y; i]$	regra da minimização de f_i na variável y
H	*hipótese* (função assumida como computável)
T	*teorema* (função computável)

No formato de prova que usaremos cada função da sequência $f_1, \ldots, f_m$ ocupa uma linha. A i-ésima linha corresponde à descrição da função f_i e tem quatro colunas:

1. o identificador da linha (i);
2. os argumentos (por exemplo, x ou x, z);
3. a expressão da função (por exemplo, $\mathsf{dobro}\,(x)$ ou $|z - x|$);
4. o axioma ou a regra de construção que define f_i;

Exemplos: A seguinte prova axiomática, de cinco linhas, demonstra que a função soma é computável (o texto em itálico são apenas comentários e não contribuem para a correcção da prova):

1.	x	$x+1$	S	
2.	x	x	P_1^1	
3.	x, y, z	z	P_3^3	*as funções f_1, f_2 e f_3 são axiomas*
4.	x, y, z	$z+1$	$\mathsf{C}[1;3]$	$f_4 = f_1 \circ f_3$
5.	x, y	$x+y$	$\mathsf{R}[2;4]$	*recursão de base f_2 e passo f_4 (pág. 94)*

Na última linha desta prova temos um teorema, a função $f_5 : (x, y) \mapsto x+y$, isto é, a soma. Esta pode agora contribuir para a demonstração de outros teoremas. Por exemplo, é útil para demonstrar a computabilidade do produto:

1.	x	0	Z	
2.	x, y, z	z	P_3^3	
3.	x, y, z	x	P_1^3	
4.	x, y	$x+y$	T	*usamos o teorema anterior*
5.	x, y, z	$z+x$	$\mathsf{C}[4;2,3]$	
6.	x, y	$x \times y$	$\mathsf{R}[1;5]$	

A função pred (*cf.* páginas 25 e 97) é computável por recursão, com base a constante zero e passo $x, z \mapsto x$:

1.		0	0	*f_1 é a constante zero, não o axioma* Z
2.	x, z	z	P_2^2	
3.	x	$\mathsf{pred}\,(x)$	$\mathsf{R}[1;2]$	

Também $\mathsf{monus}\,(x, y)$ (*cf.* páginas 26 e 98) é computável por recursão. Se aplicarmos a recorrência sobre o argumento y obtemos as igualdades

$$\begin{aligned} \mathsf{monus}\,(x, 0) &= x \\ \mathsf{monus}\,(x, y+1) &= \mathsf{pred}\,(\mathsf{monus}\,(x, y))\,. \end{aligned}$$

Portanto monus será computável se

$$\begin{aligned} f\,(x) &= x \\ g\,(x, y, z) &= \mathsf{pred}\,(z) \end{aligned}$$

forem computáveis. Resume-se este argumento na seguinte prova axiomática:

1.	x	x	P_1^1
2.	x	$x \dot{-} 1$	$\mathsf{T} : \mathsf{pred}$
3.	x, y, z	z	P_3^3
4.	x, y, z	$z \dot{-} 1$	$\mathsf{C}[2; 3]$
5.	x, y	$x \dot{-} y$	$\mathsf{R}[1; 4]$

A seguir temos a prova da computabilidade da função sinal, sg (*cf.* página 98):

1.		0	0
2.	x	0	Z
3.	x	$x + 1$	S
4.	x	1	$\mathsf{C}[3; 2]$
5.	x, y	x	P_1^2
6.	x, z	1	$\mathsf{C}[4; 5]$
7.	x	$\mathsf{sg}\,(x)$	$\mathsf{R}[1; 6]$

Segue a prova axiomática da soma limitada (*cf.* página 100) $\Sigma_{z<y} h\,(\boldsymbol{x}, z)$, sendo h uma função computável e $\boldsymbol{x} = x_1, \ldots, x_n$:

1.	x	0	T	*exercício*
2.	x, y	$x + y$	$\mathsf{T} : \mathsf{soma}$	
3.	$\boldsymbol{x}, y$	$h\,(\boldsymbol{x}, y)$	$\mathsf{H}\,(h)$	*h é computável*
4.1	$\boldsymbol{x}, y, z$	x_1	P_1^{n+2}	
$\vdots$				
4.n	$\boldsymbol{x}, y, z$	x_n	P_n^{n+2}	
5.	$\boldsymbol{x}, y, z$	y	P_{n+1}^{n+2}	
6.	$\boldsymbol{x}, y, z$	$h\,(\boldsymbol{x}, y)$	$\mathsf{C}[3; 4.1, \ldots, 4.n, 5]$	
7.	$\boldsymbol{x}, y, z$	z	P_{n+2}^{n+2}	
8.	$\boldsymbol{x}, y, z$	$h\,(\boldsymbol{x}, y) + z$	$\mathsf{C}[2; 6, 7]$	
9.	$\boldsymbol{x}, y$	$\sum_{z<y} h\,(\boldsymbol{x}, z)$	$\mathsf{R}[1; 8]$	

Compilação de Provas Axiomáticas

Uma prova axiomática é, assim, uma sequência de funções computáveis onde se explicita sem ambiguidade, linha a linha, a justificação da sua computabilidade. Com esta informação vamos definir um algoritmo que, dada uma prova axiomática de uma função f, produz um programa URM que computa f. Ao processo de transformar uma prova axiomática num programa URM chamamos **compilação**.

Como fazer uma compilação? Se a função for um axioma, a demonstração do lema 3.1.2 dá-nos os programas que devemos usar. Da mesma forma, se a função resultar das regras da composição, recursão ou minimização, as respectivas demonstrações dos teoremas (3.1.3, 3.1.5 e 3.1.10 respectivamente) fornecem esquemas de programas adequados.

Seja $f_1, \ldots, f_n$ a prova axiomática de $f = f_n$. A compilação desta prova axiomática passa pela construção progressiva dos respectivos programas $P_1, \ldots, P_n$, que computam as funções dessa sequência; no fim deste processo P_n computa f. Para cada função f_i verificamos a i-ésima linha da prova axiomática, para determinar a construção da sua computabilidade. De seguida escolhemos o esquema de programa correspondente e preenchemos os respectivos módulos desse esquema com os programas das funções referidas na linha i.

Por exemplo, se a justificação de uma determinada linha for $\mathsf{C}[2;3]$ devemos usar o esquema da composição, preenchendo os módulos desse esquema com os programa determinados nas linhas 2 e 3. Resumidamente, os esquemas de compilação são:

para os axiomas (*cf.* 3.1.2):

- constantes, n: $\mathtt{C_n}$;
- zero, Z: $[\mathtt{Z}\,(1)]$;
- sucessor, S: $[\mathtt{S}\,(1)]$;
- projecções, P_i^n: $[\mathtt{T}\,(i,1)]$.

para as regras de construção (*cf.* 3.1.3, 3.1.5 e 3.1.10):

- composição, $\mathsf{C}[f; g_1, \ldots, g_k]$:

$$\left[\begin{array}{l} G_1\,[1, \ldots, n \to n+1] \\ \vdots \\ G_k\,[1, \ldots, n \to n+k] \\ F\,[n+1, \ldots, n+k \to 1] \end{array}\right]$$

- recursão, $\mathsf{R}[f; g]$:

$$\left[\begin{array}{l} F\,[1, \ldots, n \to n+3] \\ \hline \mathtt{J}\,(n+1, n+2, 6) \\ G\,[1, \ldots, n, n+2, n+3 \to n+3] \\ \mathtt{S}\,(n+2) \\ \mathtt{J}\,(1, 1, 2) \\ \hline \mathtt{T}\,(n+3, 1) \end{array}\right]$$

- minimização, $\mu_y\,[f]$:

$$\left[\begin{array}{l} F\,[1, \ldots, n+1 \to n+3] \\ \mathtt{J}\,(n+2, n+3, 5) \\ \mathtt{S}\,(n+1) \\ \mathtt{J}\,(1, 1, 1) \\ \hline \mathtt{T}\,(n+1, 1) \end{array}\right]$$

Vejamos um exemplo completo, compilando a prova axiomática da função $\mathsf{pred}\,(x)$:

1.		0	0	f_1
2.	x, z	z	P_2^2	f_2
3.	x	$\mathsf{pred}\,(x)$	$\mathsf{R}[1; 2]$	f_3

A primeira linha é a constante 0, que pode ser calculada pelo programa $[\mathtt{Z}\,(1)]$. Assim, $P_1 = [\mathtt{Z}\,(1)]$. Na segunda linha ocorre o axioma P_2^2, ou seja, $P_2 = [\mathtt{T}\,(2, 1)]$. Na terceira linha, f_3 é computável pela expressão $\mathsf{R}[1; 2]$. Vamos usar o esquema da recursão, que usa como módulos F e G, os programas P_1 e P_2, respectivamente. Inserindo estes programas no

esquema da recursão (e como a aridade de pred é um, o valor n referido no esquema é igual a zero):

$$P_3 = \left[\begin{array}{l} \dfrac{P_1\,[\rightarrow 3]}{\mathtt{J}\,(1,2,6)} \\ P_2\,[2,3 \rightarrow 3] \\ \mathtt{S}\,(2) \\ \dfrac{\mathtt{J}\,(1,1,2)}{\mathtt{T}\,(3,1)} \end{array}\right]$$

Para obtermos o programa P_3 necessitamos de expandir os módulos P_1 e P_2. Sendo $\rho\,(P_1) = 1$ e $\rho\,(P_2) = 2$ necessitamos de dois registos no início para a execução dos módulos. Assim, desviamos todos os registos duas posições para a frente:

$$\left[\begin{array}{l|l|l} \mathtt{T}\,(1,3) & \begin{array}{l} \dfrac{P_1\,[\rightarrow 5]}{\mathtt{J}\,(3,4,7)} \\ P_2\,[4,5 \rightarrow 5] \\ \mathtt{S}\,(4) \\ \dfrac{\mathtt{J}\,(1,1,3)}{\mathtt{T}\,(5,3)} \end{array} & \mathtt{T}\,(3,1) \end{array}\right]$$

Precisamos de expandir cada módulo:

$$\begin{array}{c|c} P_1 & P_2 \\ \left[\begin{array}{l} \dfrac{\mathtt{Z}\,(1)}{\mathtt{Z}\,(1)} \\ \dfrac{}{\mathtt{T}\,(1,5)} \end{array}\right] & \left[\begin{array}{l} \mathtt{T}\,(4,1) \\ \dfrac{\mathtt{T}\,(5,2)}{\mathtt{T}\,(1,1)} \\ \dfrac{}{\mathtt{T}\,(1,5)} \end{array}\right] \end{array}$$

e inserir essas expansões no programa P_3:

$$\left[\begin{array}{l|l|l|l|l|l} \mathtt{T}\,(1,3) & \begin{array}{l} \dfrac{\mathtt{Z}\,(1)}{\mathtt{Z}\,(1)} \\ \dfrac{}{\mathtt{T}\,(1,5)} \end{array} & \mathtt{J}\,(3,4,7) & \begin{array}{l} \mathtt{T}\,(4,1) \\ \dfrac{\mathtt{T}\,(5,2)}{\mathtt{T}\,(1,1)} \\ \dfrac{}{\mathtt{T}\,(1,5)} \end{array} & \begin{array}{l} \mathtt{S}\,(4) \\ \dfrac{\mathtt{J}\,(1,1,3)}{\mathtt{T}\,(5,3)} \end{array} & \mathtt{T}\,(3,1) \end{array}\right]$$

Finalmente, é necessário actualizar as instruções de salto nos módulos (neste caso, não têm) e as instruções de salto do programa principal:

$$\left[\begin{array}{c|c|c|c|c|c|c} \mathtt{T}(1,3) & \begin{array}{c}\mathtt{Z}(1)\\ \hline \mathtt{Z}(1)\\ \hline \mathtt{T}(1,5)\end{array} & \mathtt{J}(3,4,\mathbf{12}) & \begin{array}{c}\mathtt{T}(4,1)\\ \mathtt{T}(5,2)\\ \hline \mathtt{T}(1,1)\\ \hline \mathtt{T}(1,5)\end{array} & \begin{array}{c}\mathtt{S}(4)\\ \mathtt{J}(1,1,\mathbf{5})\\ \hline \mathtt{T}(5,3)\end{array} & \mathtt{T}(3,1) \end{array}\right]$$

Este processo de compilação tende a criar programas com muitas instruções, normalmente com mais instruções do que uma solução construída de raiz por uma pessoa (para a função pred, o programa da página 25 tem apenas sete instruções). O interessante desta abordagem é a existência de um *procedimento automático* de construção de programas a partir do formalismo das provas axiomáticas.

3.3 Funções Parciais Recursivas

Como verificámos antes, as funções primitivas recursivas são definidas à custa dos axiomas e da aplicação finita de composições e recursões. Agora, se permitimos o uso da regra da minimização obtemos um conjunto novo, maior, de funções computáveis (onde se inclui a função de Ackermann, funções parciais...).

Definição 3.3.1 (Funções parciais recursivas)
O conjunto das ***funções parciais recursivas****, $\mathcal{R}$, é o mais pequeno conjunto que contém os axiomas n,* Z*,* S *e* P_i^n *e que é fechado para as operações de composição, recursão e minimização.*

Pretendemos relacionar os conjuntos $\mathcal{R}$ e $\mathcal{C}$ mas antes de prosseguir é necessário definir algumas funções auxiliares para analisar os códigos de programas, memória, *etc.*.

Definição 3.3.2 (Funções p.r. auxiliares) *Seja P um programa URM e fixemos um estado da memória $\boldsymbol{r} = (r_1, \ldots, r_n)$ (onde n é escolhido de forma que, se $j > n$, então $r_j = 0$) e uma instrução actual I_i; fixemos também $c = \mathsf{cod}(P)$ (e, portanto, $P = \mathsf{dec}(c)$).*

- *o* ***código da memória*** *é*

$$m = \mathsf{codmem}\,(\boldsymbol{r}) \quad = \quad 2^{r_1} \times 3^{r_2} \times \cdots \times p_n^{r_n} \tag{3.39}$$

onde p_i *é o* i*-ésimo número primo;*

- *o* ***código da configuração*** *da computação depende do programa* P*, da memória* $\boldsymbol{r}$ *e da instrução actual* I_i*:*

$$s = \mathsf{config}\,(c, i, m) \quad = \quad \Pi\,(\Pi\,(c, i)\,, m) \tag{3.40}$$

- *após o processamento da instrução* I_i *do programa* P *com a configuração de memória* $\boldsymbol{r}$*, obtemos:*
  - *(****exercício:*** *mostre que é p.r.) o índice da* ***próxima instrução****:*

$$i' \quad = \quad \mathsf{instr}\,(s) \tag{3.41}$$

  convencionando que, quando s *for terminal,* $\mathsf{instr}\,(s) = 0$*;*
  - *(****exercício:*** *mostre que é p.r.) o código do* ***novo estado da memória****:*

$$m' \quad = \quad \mathsf{mem}\,(s)\,; \tag{3.42}$$

  - *o código da* ***nova configuração****:*

$$\mathsf{passo}\,(s) \quad = \quad \mathsf{config}\,(c, i', m')\,; \tag{3.43}$$

- *a sucessão dos códigos das configuração de uma computação pode agora ser definida por*

$$\begin{aligned} s_0 &= \mathsf{config}\,(c, 1, m)\,; \\ s_1 &= \mathsf{passo}\,(s_0)\,; \\ s_2 &= \mathsf{passo}\,(s_1)\,; \\ &\vdots \end{aligned}$$

Esta sucessão fica definida por recorrência pelas equações

$$\begin{cases} \mathsf{comput}\,(s,0) & = & s \\ \mathsf{comput}\,(s,n+1) & = & \mathsf{passo}\,(\mathsf{comput}\,(s,n)) \end{cases} . \tag{3.44}$$

Dados dois números s e n, $\mathsf{comput}\,(s,n)$ *codifica a configuração que se obtém a partir da configuração inicia de código s, após terem sido executados n passos elementares. A sucessão de configurações, isto é, a* ***computação****, pode ser re-escrita como*

$$\begin{aligned} s_0 &= \mathsf{comput}\,(s_0,0)\,; \\ s_1 &= \mathsf{comput}\,(s_0,1)\,; \\ s_2 &= \mathsf{comput}\,(s_0,2)\,; \\ &\vdots \end{aligned}$$

- *por minimização obtemos o número mínimo de passos necessários para que, com o estado inicial de memória* $\boldsymbol{r}$*, o programa P termine a computação, isto é, o* ***tempo*** *da computação:*

$$\mathsf{tempo}(c,m) \;=\; \mu_z\,[\mathsf{instr}(\mathsf{comput}(\mathsf{config}(c,1,m),z))] \tag{3.45}$$

Esta minimização resulta pois antes definimos instr *de modo que* $\mathsf{instr}\,(s)=0$ *quando s for um estado terminal;*

- *o resultado dessa computação é o valor que está no primeiro registo, o que nos permite definir a seguinte função* ***universal****:*

$$\mathsf{u}(c,m) \;=\; (\Pi_2(\mathsf{comput}(\mathsf{config}(c,1,m),\mathsf{tempo}(c,m))))_1 \tag{3.46}$$

Em particular, u *permite re-definir as funções universais da definição 2.7.1:*

$$\begin{aligned} \Psi_{\mathcal{U}}^{(1)}\,(c,x) &= \mathsf{u}\,(c,2^x) \\ \Psi_{\mathcal{U}}^{(2)}\,(c,x,y) &= \mathsf{u}\,(c,2^x3^y) \\ \Psi_{\mathcal{U}}^{(3)}\,(c,x_1,x_2,x_3) &= \mathsf{u}\,(c,2^{x_1}3^{x_2}5^{x_3}) \\ &\vdots \end{aligned}$$

Todas estas funções são computáveis (basta observar como estão definidas) e, com a excepção das duas últimas, tempo e u, são mesmo *primitivas recursivas* (e, portanto, *totais*). No entanto tempo e u usam o operador de minimização ilimitada, mas *apenas uma única vez.* O primeiro argumento de u, c, é o código (um número natural) de um programa URM P e o segundo argumento, m, (também um número natural) codifica a configuração inicial, $\boldsymbol{r}$, dos registos de P. O resultado da computação de P a partir da memória $\boldsymbol{r}$ (*i.e.* o valor do primeiro registo) é então

$$\mathsf{u}\left(c,m\right)=\mathsf{u}\left(\mathsf{cod}\left(P\right),\mathsf{codmem}\left(\boldsymbol{r}\right)\right) \tag{3.47}$$

que, no caso da computação não terminar, fica

$$\mathsf{u}\left(c,m\right)\uparrow .$$

Resumimos estas conclusões no

Lema 3.3.3 *As funções* codmem, config, instr, mem, passo *e* comput *são primitivas recursivas; As funções* tempo *e* u *são parciais recursivas.*

Estamos agora em condições de perguntar *qual é a relação entre* $\mathcal{R}$ *e* $\mathcal{C}$*, o conjunto das funções URM-computáveis?* O teorema seguinte mostra que estes conjuntos são iguais.

Teorema 3.3.4

$$\mathcal{R}=\mathcal{C}.$$

Demonstração. Pelo lema 3.1.2 e pelos teoremas 3.1.3, 3.1.5 e 3.1.10, obtemos $\mathcal{R}\subseteq\mathcal{C}$.

Para mostrarmos que $\mathcal{C}\subseteq\mathcal{R}$ precisamos mostrar que cada função (URM-)computável também é parcial recursiva.

Seja então $f\in\mathcal{C}$. Isso significa que existe um certo programa F que computa f:

$$\forall x,y\in\mathbb{N},f\left(x\right)=y\Leftrightarrow F\left(x\right)\downarrow y$$

Usando a função parcial recursiva u:

$$F(x) \downarrow y \Leftrightarrow \mathsf{u}(\mathsf{cod}(F), 2^x) = y$$

Fazendo $c = \mathsf{cod}(F)$, podemos concluir que

$$\forall x, y \in \mathbb{N}, f(x) = y \Leftrightarrow \mathsf{u}(c, 2^x) = y$$

Isto é:

$$f = \mathsf{C}[\mathsf{u}; c, \mathsf{exp2}]$$

o que mostra que f é primitiva recursiva, isto é, $f \in \mathcal{R}$ (a demonstração de que $\mathsf{exp2}(x) = 2^x$ é p.r. está nos exercícios). Temos então que

$$\forall f, f \in \mathcal{C} \Rightarrow f \in \mathcal{R}$$

o que nos permite concluir que

$$\mathcal{C} \subseteq \mathcal{R}$$

Em conclusão, tendo $\mathcal{C} \supseteq \mathcal{R}$ e $\mathcal{R} \supseteq \mathcal{C}$, obtemos $\mathcal{R} = \mathcal{C}$ □

A demonstração desta igualdade passou por definir uma função universal. Esta descrição pode ser apresentada como uma prova axiomática que, por sua vez, pode ser compilada. A compilação desta prova axiomática (que não faremos por ser demasiado extensa) resulta num programa URM universal!

Pela definição, $\mathcal{R} \supset \mathcal{PR}$, isto é, as funções parciais recursivas incluem as primitivas recursivas. É costume definir o conjunto das funções recursivas, $\mathcal{R}_0$, como a união do subconjunto de $\mathcal{R}$ das funções primitivas recursivas com as funções totais definidas por minimização. Devido à existência de funções totais não primitivas recursivas, como a função de Ackermann, $\mathcal{R}_0 \supset \mathcal{PR}$. E qual a relação entre $\mathcal{R}_0$ e o conjunto de todas as funções totais recursivas?

Teorema 3.3.5 *O conjunto $\mathcal{R}_0$ contém todas as funções totais recursivas.*

Demonstração. Seja uma função f total recursiva, assim, $f \in \mathcal{R}$. Pelo teorema anterior existe um programa URM $\mathtt{P}$ que calcula f e converge em qualquer argumento x (pois f é total). Assim, a função $\mathtt{P}, x \mapsto \mathsf{u}(\mathsf{cod}(\mathtt{P}), 2^x)$ é total recursiva. Como esta função é definida pela aplicação de composições, recursões e uma minimização que produz uma função total, por definição $f \in \mathcal{R}_0$. □

3.4 Exercícios

Exercícios Resolvidos

Consideramos como teoremas as funções já resolvidas no capítulo, nomeadamente, a soma, o produto, o predecessor, $\dot{-}$, a função sg e a soma limitada, se vierem a ser necessárias nas provas axiomáticas seguintes.

Módulo da Diferença. Em certos casos, é mais fácil expressar uma função pela composição de outras funções mais simples. É este o caso da nossa função binária que pode ser definida à custa da função monus.

O módulo da diferença $|x - y|$ é um valor não-negativo que pode ser obtido de uma de duas formas. Se $x > y$ então o resultado é dado pela subtracção $x \dot{-} y$, se $x < y$ então o resultado é dado pela subtracção $y \dot{-} x$. Quando $x = y$ ambas as expressões calculam o valor correcto, isto é, $|x - y| = 0$.

Como combinar estas duas expressões numa só? Como em cada caso, a outra expressão é igual a zero, basta somar as duas, ou seja:

$$|x - y| = (x \dot{-} y) + (y \dot{-} x)$$

Com esta expressão, a estrutura da prova axiomática fica facilitada:

1.	x, y	$x \ominus y$	$\mathsf{T} : \mathsf{monus}$	*precisamos da função* monus
2.	x, y	$x + y$	$\mathsf{T} : \mathsf{soma}$	*precisamos da função* soma
3.	x, y	x	P_1^2	
4.	x, y	y	P_2^2	
5.	x, y	$y \ominus x$	$\mathsf{C}[1; 4, 3]$	
6.	x, y	$\lvert x - y \rvert$	$\mathsf{C}[2; 1, 5]$	*fica* $\lvert x - y \rvert = (x \ominus y) + (y \ominus x)$

Máximo. Para calcular o máximo entre dois números podemos usar o mesmo esquema do exercício anterior. Reformular a expressão do máximo usando funções já nossas conhecidas.

O maior número entre x e y pode ser visto como a soma de um deles (seja x) e a sua diferença natural em relação ao outro ($y \ominus x$), ou seja,

$$\mathsf{max}\,(x, y) = x + (y \ominus x)$$

Porque é que isto funciona? Vejamos os dois casos. Se x for realmente o maior, então $y \ominus x$ dará zero, não influenciado o resultado da soma (x). Se y for o maior dos dois números, então estamos a somar o menor com a diferença entre os dois, o que resulta inevitavelmente no maior ($x + (y \ominus x) = x + (y - x) = x + y - x = y$).

A prova axiomática correspondente a este raciocínio é muito semelhante à anterior:

1.	x, y	$x \ominus y$	$\mathsf{T} : \mathsf{monus}$	*precisamos da função* monus
2.	x, y	$x + y$	$\mathsf{T} : \mathsf{soma}$	*precisamos da função* soma
3.	x, y	x	P_1^2	
4.	x, y	y	P_2^2	
5.	x, y	$y \ominus x$	$\mathsf{C}[1; 4, 3]$	
6.	x, y	$\mathsf{max}\,(x, y)$	$\mathsf{C}[2; 3, 5]$	*fica* $\mathsf{max}\,(x, y) = x + (y \ominus x)$

Resto da Divisão Inteira. A função $\mathsf{rm}\,(x, y)$ calcula o resto da divisão inteira de y por x. Por exemplo, $\mathsf{rm}\,(2, 4) = 0$ (4 a dividir por 2 dá resto zero) e $\mathsf{rm}\,(2, 5) = 1$. Por convenção, estabelece-se que $\mathsf{rm}\,(0, y) = y$ para a função ser total e também para que rm possa ser directamente

calculada por recursão, tendo-se a base $\mathsf{rm}\,(x,0)=0$ e o passo

$$\mathsf{rm}\,(x,y+1)=\begin{cases} 0 & \text{se } \mathsf{rm}\,(x,y)+1=x \\ \mathsf{rm}\,(x,y)+1 & \text{caso contrário} \end{cases}.$$

Temos que verificar se as funções

$$\begin{aligned} f\,(x) &= 0 \\ g\,(x,y,z) &= (z+1)\times \mathsf{sg}\,(|(z+1)-x|) \end{aligned}$$

são computáveis. A expressão da função g deriva do facto de estar envolvida uma condição $(z+1=x)$. O valor calculado por g é $z+1$ se e só se $z+1\neq x$, ou seja se $|(z+1)-x|\neq 0$, o que pode ser expresso com a ajuda da função sg. Em particular, se $z+1=x$ então $|(z+1)-x|=0$ logo $\mathsf{sg}\,(|(z+1)-x|)=0$ e $g\,(x,y,z)=0$. Mas se $z+1\neq x$ então $|(z+1)-x|\neq 0$ logo $\mathsf{sg}\,(|(z+1)-x|)=1$ e $g\,(x,y,z)=z+1$.

A base f é o axioma Z e o passo g é dado pela composição de diversas funções computáveis.

1.	x	0	Z	*a base f*
2.	x	$x+1$	S	
3.	x	$\mathsf{sg}\,(x)$	$\mathsf{T}:\mathsf{sg}$	
4.	x,y	$\lvert x-y\rvert$	$\mathsf{T}:\lvert x-y\rvert$	
5.	x,y	$x\times y$	$\mathsf{T}:\mathsf{produto}$	
6.	x,y,z	x	P_1^3	*destacar x...*
7.	x,y,z	z	P_3^3	*... e z*
8.	x,y,z	$z+1$	$\mathsf{C}[2;7]$	
9.	x,y,z	$\lvert(z+1)-x\rvert$	$\mathsf{C}[4;8,6]$	
10.	x,y,z	$\mathsf{sg}\,(((z+1)-x))$	$\mathsf{C}[3;9]$	
11.	x,y,z	$(z+1)\times \mathsf{sg}\,(\lvert(z+1)-x\rvert)$	$\mathsf{C}[5;8,10]$	*o passo g*
12.	x,y	$\mathsf{rm}\,(x,y)$	$\mathsf{R}[1;11]$	*recursão*

Predicado par. Este exercício refere-se a um predicado; a prova axiomática será da sua função característica:

$$\mathsf{c}_{\mathsf{par}}\,(x)=\begin{cases} 1 & \text{se } x \text{ for par} \\ 0 & \text{caso contrário} \end{cases}$$

Para determinar se um número n é par, podemos verificar se o resto da divisão inteira de n por 2 é zero. Se x for par então $\overline{\mathsf{sg}}\,(\mathsf{rm}\,(2,x))$ será 1, senão será 0. Ora, são estes os resultados que pretendemos para a nossa função característica, assim:

$$\mathsf{c_{par}}\,(x) \quad = \quad \overline{\mathsf{sg}}\,(\mathsf{rm}\,(2,x)) \tag{3.48}$$

Agora podemos fazer a prova axiomática:

1.	x, y	$\mathsf{rm}\,(x,y)$	$\mathsf{T} : \mathsf{rm}$	
2.	x	$\overline{\mathsf{sg}}\,(x)$	$\mathsf{T} : \overline{\mathsf{sg}}$	
3.	x	2	T	*exercício*
4.	x	x	P_1^1	
5.	x	$\mathsf{rm}\,(2,x)$	$\mathsf{C}[1;3,4]$	
6.	x	$\mathsf{c_{par}}\,(x)$	$\mathsf{C}[2;5]$	*fica* $\mathsf{c_{par}}\,(x) = \overline{\mathsf{sg}}\,(\mathsf{rm}\,(2,x))$

Raiz Quadrada Inteira. Pretendemos calcular a parte inteira da raiz quadrada de um dado número natural. Por exemplo, $\mathsf{raiz}\,(16) = 4$ e $\mathsf{raiz}\,(17) = 4$.

Esta função pode ser calculada testando sucessivamente $i = 0, 1, 2, 3, ...$ até que o respectivo quadrado ultrapasse o número x dado para calcular a raiz, isto é, até que $i^2 > x$. Por exemplo, se $x = 17$, os quadrados testados seriam $0, 1, 4, 9, 16, 25$ parando a progressão em $i^2 = 25$ porque teria ultrapassado o valor de x. O resultado que pretendemos será o penúltimo valor de i, 4. Isto pode ser efectuado pela minimização limitada (não precisamos da minimização porque podemos calcular um limite com x):

$$\mathsf{raiz}\,(x) = \mu_{z<x+2}\left[z^2 > x\right] \dot{-} 1$$

O limite «$x + 2$» resulta da necessidade de ultrapassar $x = 0$ ou $x = 1$. O cálculo do predecessor corresponde a querermos o valor *anterior* ao que verifica a condição $z^2 > x$. Ou seja, não pretendemos o menor z tal que z^2 que ultrapassa x, mas sim o maior z tal que z^2 *não* ultrapassa x. Falta-nos agora substituir a relação «>» pela sua característica (*cf.* página 99):

$$\mathsf{raiz}\,(x) \quad = \quad \mu_{z<x+2}\left[\overline{\mathsf{sg}}\left(z^2 \dot{-} x\right)\right] \dot{-} 1 \tag{3.49}$$

A prova axiomática fica:

1.	x	$\overline{\mathsf{sg}}\,(x)$	$\mathsf{T} : \overline{\mathsf{sg}}$	
2.	x	$x \dot{-} 1$	$\mathsf{T} : \mathsf{pred}$	
3.	x, y	$x \dot{-} y$	$\mathsf{T} : \mathsf{monus}$	
4.	x, y	x	P_1^2	
5.	x, y	y^2	T	*exercício*
6.	x, y	$x + 2$	T	*exercício*
7.	x, z	$z^2 \dot{-} x$	$\mathsf{C}[3; 5, 4]$	
8.	x, z	$\overline{\mathsf{sg}}\,(z^2 \dot{-} x)$	$\mathsf{C}[1; 7]$	
9.	x	$\mu_{z<x+2}\,[\overline{\mathsf{sg}}\,(z^2 \dot{-} x)]$	$\mathsf{T} : \mu$	*minimização limitada*
10.	x	$\mathsf{raiz}\,(x)$	$\mathsf{C}[2; 9]$	

Exercícios por Resolver

Exercício 3.4.1 *Prove, pela via axiomática, a computabilidade das seguintes funções:*

- $\mathsf{quatro}\,(x) = 4$*;*
- $\mathsf{dobro}\,(x) = 2x$*,*
- $\mathsf{triplo}\,(x) = 3x$*,*
- $\overline{\mathsf{sg}}\,(x)$*, vale* 1 *se* $x > 0$ *e vale* 0 *se* $x = 0$*;*
- $\mathsf{exp2}\,(x) = 2^x$*;*
- $\mathsf{exp}\,(x, y) = x^y$*;*
- $\mathsf{min}\,(x, y)$*, o menor valor entre* x *e* y*;*
- $\mathsf{qt}\,(x, y)$*, o quociente da divisão inteira de* y *por* x *(com* $\mathsf{qt}\,(0, y) = 0$*);*
- $\mathsf{mmc}\,(x, y)$*, o menor múltiplo comum entre* x *e* y*;*
- $\mathsf{D}\,(x)$*, o número de divisores de* x *(com* $\mathsf{D}\,(0) = 1$*);*
- $\mathsf{p}\,(x)$*, o* x*-ésimo primo (com* $\mathsf{p}\,(0) = 0$*);*
- $(x)_y$*, o* y*-ésimo expoente na expansão prima de* x *(*cf. *A.3, p.252);*
- $\mathsf{mdc}\,(x, y)$*, o maior divisor comum entre* x *e* y*;*

- $\mathsf{euler}\,(x)$, *quantos números entre 2 e* $x-1$ *são primos com* x *(e.g.,* $\mathsf{euler}\,(6)=1$, $\mathsf{euler}\,(9)=5$*);*
- $\mathsf{perfeito}\,(x)$, *se* x *é número perfeito,* i.e., *se a soma de todos os divisores de* x *(excepto o próprio) é igual a* x*;*
- *A função característica de «*$x \neq 2$*»;*
- *A função característica de «*$x \neq y$*»;*
- *A função característica de «*$x > y$*»;*
- *A função característica de «*x *é quadrado perfeito»;*
- *A função característica de «*$x|y$*»,* i.e., *«*x *é divisor de* y*»;*
- *A função característica de «*x *é primo».*

Exercício 3.4.2 *Faça a compilação das provas axiomáticas de:*

- $\mathsf{soma}\,(x,y)=x+y$*;*
- $f\,(x,y,z)=x+y$*;*
- $\mathsf{monus}\,(x,y)$*;*
- $\mathsf{sg}\,(x)$ *(*assuma o teorema $f\,(x)=1$ com programa $[\mathtt{Z}\,(1)\,,\mathtt{S}\,(1)]$*);*
- $\mathsf{min}\,(x,y)$ *(*assuma o teorema $\mathsf{monus}\,(x,y)$ com o programa da página 26*).*

Entre Capítulos

Introduzimos uma noção de computação baseada num sistema axiomático que produz funções computáveis a partir de outras funções computáveis. Verificou-se que esta abordagem axiomática da computação é equivalente à abordagem operacional da máquina de registos (e igualmente da máquina de Turing, apesar de não termos apresentado este resultado).

Se nestes capítulos falámos do que é computável e das suas propriedades, falaremos nos seguintes das suas limitações. O próximo capítulo apresentará os limites da computação, ou seja, será demonstrado que existem problemas concretos que não podem ser resolvidos por qualquer

máquina ou por outro qualquer formalismo equivalente de computação. Este resultado, talvez contra o senso comum, diz-nos que certas limitações dos nossos computadores não dependem da tecnologia mas sim da própria estrutura matemática que lhes está subjacente.

Capítulo 4

Indecidibilidade

Neste capítulo estudaremos os limites teóricos da computação: veremos que nem todas as questões podem ser resolvidas computacionalmente e aprenderemos algumas técnicas para determinar se uma dada linguagem pode ser, ou não, *decidida* por algum algoritmo.

A noção de decidibilidade algorítmica assenta na seguinte ideia intuitiva: uma linguagem A pode ser *decidida* ser existir algum algoritmo que responda (correctamente) «sim» ou «não» à questão «$x \in A$?». Nesse caso dizemos que A é uma **linguagem decidível**. Por exemplo, o conjunto $\{7, 9\}$ é decidível porque existe um algoritmo que, dado um inteiro n, responde «sim» se $n = 7$ ou $n = 9$ e responde «não» caso contrário (por exemplo, se $n = 4$).

Com um pouco mais de generalidade definem-se também as linguagens semi-decidíveis: A é uma **linguagem semi-decidível** se existir um algoritmo que responda «sim» se, e só se, $x \in A$. Mas da resposta que deve ser dada no caso $x \notin A$ só sabemos que não é «sim», podendo até o algoritmo não terminar — ao contrário do que acontece com as linguagens decidíveis, onde o algoritmo «tem» de responder «não». Intuitivamente, suspeitamos, desde já, que qualquer linguagem decidível é, também, semi-decidível — se insistirmos que o algoritmo não termine no caso «$x \notin A$» basta adaptar um programa que decida a linguagem, substituindo o «ramo» em que esse programa responde «não» por um

ciclo infinito, para se obter um programa que «semi-decide» essa linguagem. A definição de linguagem semi-decidível é menos restritiva do que a de linguagem decidível. No último caso é exigida uma resposta em *todos* os casos enquanto que no primeiro só é exigida uma resposta nos casos positivos.

Embora as linguagens semi-decidíveis possam parecer, neste momento, um pouco artificiais, veremos que reflectem uma situação importante na computabilidade, pois permitem formalizar situações do seguinte tipo:

> Queremos usar um computador para saber se *existem quarenta '7' consecutivos na expansão decimal de* π. Fazemos um programa (que vá calculando os dígitos de π e observando se ocorrem quarenta '7' consecutivos) e deixamos o computador a «correr» esse programa... Esperamos uma hora, e o programa não terminou... duas horas e nada ainda... passa-se um dia, uma semana, e nada...
>
> *O que ficámos a saber passado esse tempo?* Nada descobrimos sobre a existência dos quarenta '7' consecutivos...
>
> Devemos abandonar a pesquisa, interrompendo o programa? Mas assim a questão fica por responder. Mais perverso é que, supondo que a pesquisa percorreu já um milhão de dígitos, até pode acontecer que os quarenta '7' consecutivos surjam com o próximo dígito — é como falhar a lotaria por uma semana.
>
> Por outro lado, se optarmos por deixar o programa a correr, também pode acontecer que π não tenha os quarenta '7' consecutivos, e só estamos a perder tempo e gastar energia explorando mais e mais dígitos.

Estes dilemas, sem solução, podem efectivamente ocorrer nas aplicações da teoria da computação. Assim, é importante estarmos munidos de ferramentas que nos permitam identificar e, dentro do possível, lidar com situações deste tipo.

Tais situações, que envolvem computações infinitas, são formalizadas pelas linguagens que *não* são decidíveis ou que *não* são semi-decidíveis. Isto é, as linguagens que não podem ser definidas pela *terminação de algoritmos* e que, portanto, transcendem o que pode ser feito por qualquer computador, actual ou que venha a ser construído no futuro, independentemente do progresso da tecnologia. É uma limitação *teórica*, que não pode ser resolvida por via de avanços *práticos* ou *tecnológicos*.

Os temas principais deste capítulo são, primeiro, investigar se existem tais linguagens, e, segundo, sabendo que existem, como poderão ser caracterizadas e também o que mais poderemos descobrir sobre elas.

Para esse estudo desenvolveremos um pequeno conjunto de técnicas: o **teorema s-m-n**, a **intercalação** e a **redução**. Empregaremos frequentemente uma determinada função (chamada «**resultado**») reminescente do **predicado-T de Kleene** que, intuitivamente, é verdadeiro se, e só se, a computação de um dado programa c, com argumento x, termina em n (ou menos) passos elementares.

4.1 Revisões

Linguagens, predicados e funções características. Procurando alguma economia de conceitos, optamos por não trabalhar com «**predicados decidíveis**». Mas esta opção só é possível porque a decidibilidade dos predicados pode ser substituída (sem prejuízo, graças às equivalências 2.53 e 2.54, na página 60) pela decidibilidade de linguagens — que será desenvolvida aqui.

Redução de aridade. Outra simplificação importante neste capítulo tem a ver com a aridade das funções: usaremos frequentemente funções unárias para «representar» outras funções de maior aridade. *Como?* Com a bijecção $\Pi : \mathbb{N}^2 \to \mathbb{N}$ (definida no anexo A.3, p.259) e que já usámos para calcular o código de Gödel dos programas URM.

Vejamos exactamente como se «reduz» a aridade de funções, come-

çando pelas funções binárias. Seja

$$\begin{array}{rccc} f: & \mathbb{N}^2 & \to & \mathbb{N} \\ & (x,y) & \mapsto & f(x,y) \end{array}$$

uma função binária. Usando a bijecção Π e as inversas parciais Π_1 e Π_2, podemos definir a função unária

$$\begin{array}{rccc} \overline{f}: & \mathbb{N} & \to & \mathbb{N} \\ & z & \mapsto & f(\Pi_1(z), \Pi_2(z)) \end{array}$$

que respeita a equação (**exercício**: verifique esta igualdade)

$$f(x,y) = \overline{f}(\Pi(x,y)). \tag{4.1}$$

Reciprocamente, sendo g uma função unária, podemos definir uma função binária

$$\begin{array}{rccc} \widetilde{g}: & \mathbb{N}^2 & \to & \mathbb{N} \\ & (x,y) & \mapsto & g(\Pi(x,y)) \end{array}$$

que respeita a equação

$$g(z) = \widetilde{g}(\Pi_1(z), \Pi_2(z)). \tag{4.2}$$

Compondo estas duas transformações de funções (de binária para unária e vice-versa), resulta, das propriedades de Π, Π_1 e Π_2, e das equações 4.1 e 4.2, que

$$\widetilde{\overline{f}} = f \tag{4.3}$$
$$\overline{\widetilde{g}} = g. \tag{4.4}$$

sendo f uma função binária e g unária. Ou seja, *a* ***redução de aridade*** *é uma bijecção entre as funções binárias e unárias.*

Exemplo 4.1.1 *A função binária*

$$\begin{array}{rccc} \mathsf{soma}: & \mathbb{N}^2 & \to & \mathbb{N} \\ & (x,y) & \mapsto & x+y \end{array}$$

fica reduzida pela função unária

$$\begin{array}{rccl} \overline{\mathsf{soma}}: & \mathbb{N} & \to & \mathbb{N} \\ & z & \mapsto & \Pi_1(z) + \Pi_2(z) = (z+1)_1 + \frac{1}{2}\left(\frac{z+1}{2^{(z+1)_1}} - 1\right) \end{array}.$$

Observemos um caso concreto:

$$\begin{aligned} 5 = 3+2 &= \mathsf{soma}(3,2) \\ &= \overline{\mathsf{soma}}(\Pi(3,2)) \\ &= \overline{\mathsf{soma}}(39) \\ &= (39+1)_1 + \frac{1}{2}\left(\frac{39+1}{2^{(39+1)_1}} - 1\right) \\ &= \cdots \\ &= 3+2 = 5. \end{aligned}$$

Este caso ilustra que o cálculo binário de $\mathsf{soma}(3,2)$ *pode ser efectuado em dois passos: primeiro, codificando o argumento* $(3,2)$ *no inteiro* $39 = \Pi(3,2)$ *e segundo, fazendo o cálculo unário* $\overline{\mathsf{soma}}(39)$*. Daqui resulta que o estudo da função binária* soma *pode ser reduzido ao estudo da função unária* $\overline{\mathsf{soma}}$*.*

Com a redução de aridade *podemos estudar as funções binárias por via das funções unárias.* Além disso, as funções Π, Π_1 e Π_2, usadas para fazer esta redução, são primitivas recursivas. Portanto,

Lema 4.1.2 *A função binária* $f : \mathbb{N}^2 \to \mathbb{N}$ *é primitiva (*resp. *parcial) recursiva se, e só se, a função unária* $\overline{f} : \mathbb{N} \to \mathbb{N}$ *for primitiva (*resp. *parcial) recursiva.*

Também temos que

Lema 4.1.3 *A função unária* $g : \mathbb{N} \to \mathbb{N}$ *é primitiva (*resp. *parcial) recursiva se, e só se, a função binária* $\widetilde{g} : \mathbb{N}^2 \to \mathbb{N}$ *for primitiva (*resp. *parcial) recursiva.*

Vejamos agora como reduzir aridades superiores a 2. Precisamos de generalizar a bijecção $\Pi : \mathbb{N}^2 \to \mathbb{N}$ para essas dimensões. Ora, a partir da bijecção $\Pi : \mathbb{N}^2 \to \mathbb{N}$, iterativamente, também se definem bijecções $\mathbb{N}^3 \to \mathbb{N}, \mathbb{N}^4 \to \mathbb{N}, \ldots$ da forma seguinte:

$$\begin{array}{lll}
\Pi^{(3)}: & \mathbb{N}^3 \to \mathbb{N}, & (x,y,z) \mapsto \Pi\left(\Pi\left(x,y\right),z\right) \\
\Pi^{(4)}: & \mathbb{N}^4 \to \mathbb{N}, & (x,y,z,w) \mapsto \Pi\left(\Pi^{(3)}\left(x,y,z\right),w\right) \\
\vdots & & \\
\Pi^{(k)}: & \mathbb{N}^k \to \mathbb{N}, & (x_1,\ldots,x_k) \mapsto \Pi\left(\Pi^{(k-1)}\left(x_1,\ldots,x_{k-1}\right),x_k\right) \\
\vdots & &
\end{array}$$

Exemplo 4.1.4

$$\begin{aligned}
\Pi^{(4)}\left(1,0,3,7\right) &= \Pi\left(\Pi^{(3)}\left(1,0,3\right),7\right) \\
&= \Pi\left(\Pi\left(\Pi\left(1,0\right),3\right),7\right) \\
&= \Pi\left(\Pi\left(1,3\right),7\right) \\
&= \Pi\left(13,7\right) \\
&= 122879
\end{aligned}$$

Tal como a bijecção $\Pi : \mathbb{N}^2 \to \mathbb{N}$ tem duas «inversas parciais», Π_1 e Π_2, ambas $\mathbb{N} \to \mathbb{N}$, também cada bijecção $\Pi^{(k)} : \mathbb{N}^k \to \mathbb{N}$ tem k «inversas parciais», $\Pi_1^{(k)}, \ldots, \Pi_k^{(k)}$, todas $\mathbb{N} \to \mathbb{N}$, de forma que, para cada $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{N}^k$,

$$\begin{aligned}
x_1 &= \Pi_1^{(k)}\left(\Pi^{(k)}\left(\boldsymbol{x}\right)\right) \\
x_2 &= \Pi_2^{(k)}\left(\Pi^{(k)}\left(\boldsymbol{x}\right)\right) \\
&\vdots \\
x_k &= \Pi_k^{(k)}\left(\Pi^{(k)}\left(\boldsymbol{x}\right)\right)
\end{aligned} \tag{4.5}$$

e, para qualquer $z \in \mathbb{N}$,

$$\Pi^{(k)}\left(\Pi_1^{(k)}\left(z\right),\ldots,\Pi_k^{(k)}\left(z\right)\right) = z. \tag{4.6}$$

Supondo agora que $f : \mathbb{N}^k \to \mathbb{N}$ é uma função computável, de aridade k, podemos definir uma função $\overline{f}$, *unária*, tal que, para qualquer $\boldsymbol{x} \in \mathbb{N}^k$ e $z \in \mathbb{N}$,

$$f(\boldsymbol{x}) = \overline{f}\left(\Pi^{(k)}(\boldsymbol{x})\right) \tag{4.7}$$

$$\overline{f}(z) = f\left(\Pi_1^{(k)}(z), \ldots, \Pi_k^{(k)}(z)\right). \tag{4.8}$$

Dado que as bijeccções $\Pi, \Pi^{(k)}$ e as suas inversas são primitivas recursivas, pelo lema 4.1.2, f é primitiva recursiva (parcial recursiva) se, e só se, $\overline{f}$ é primitiva recursiva (*resp.* parcial recursiva).

Como a notação $f\left(\Pi^{(m)}(\boldsymbol{x})\right)$ tende a tornar os enunciados ilegíveis, usamos, em sua substituição, $\overline{\boldsymbol{x}}$ para abreviar $\Pi^{(m)}(\boldsymbol{x})$ (ignorando a dimensão m do vector $\boldsymbol{x} \in \mathbb{N}^m$). Também convencionamos que, se $\boldsymbol{y} \in \mathbb{N}^n$, a notação $\overline{\boldsymbol{x}, \boldsymbol{y}}$ abrevia $\overline{\overline{\boldsymbol{x}}, \overline{\boldsymbol{y}}}$. No caso particular em que $x \in \mathbb{N}$, fica $\overline{x} = x$.

Exemplo 4.1.5

$$\begin{aligned}
\overline{56} &= 56 \\
\overline{3,4} &= \Pi(3,4) = 71 \\
\overline{1,(2,0)} &= \overline{\overline{1},\overline{2,0}} \\
&= \overline{1,\Pi(2,0)} \\
&= \overline{1,3} \\
&= \Pi(1,3) = 13 \\
\overline{(1,2),(0,3)} &= \overline{\overline{1,2},\overline{0,3}} \\
&= \overline{9,6} = 1215
\end{aligned}$$

enquanto que é errado calcular, por exemplo,

$$\overline{(1,2),(0,3)} = \overline{\overline{1,2,0},3} = \overline{\overline{\overline{1,2},0},3} = \overline{\overline{11,0},3} = \overline{22,3} = 359.$$

Assim, quando escrevermos $f(\overline{x,y})$ estaremos a pensar, conforme for mais conveniente, em f como uma função unária ou na função $(m+n)$-ária

$$x, y \mapsto f\left(\Pi\left(\Pi^{(m)}(x), \Pi^{(n)}(y)\right)\right).$$

Exemplo 4.1.6 *Voltando ao caso da* soma*, quando escrevermos*

$$\mathsf{soma}\left(\overline{3,2}\right) = 5$$

tanto podemos estar a fazer a afirmação $\overline{\mathsf{soma}}\,(39) = 5$*, sobre a função unária* $\overline{\mathsf{soma}}$*, como também podemos estar a afirmar que* $\mathsf{soma}\,(3,2) = 5$ *sobre a função binária* soma*. As equações 4.1, 4.2 e os lemas 4.1.2, 4.1.3 anteriores, dizem-nos que estas duas afirmações são equivalentes.*

Representação de predicados n-ários por linguagens em $\mathbb{N}$. Vejamos como podemos combinar a representação de predicados por linguagens com a redução de aridade.

Exemplo 4.1.7 *Ao predicado binário*

$$\mathsf{maior}\,(x,y) \Leftrightarrow x > y$$

associamos a linguagem (em $\mathbb{N}^2$*)*

$$\mathsf{Maior} = \{(x,y) \mid x > y\} = \{(1,0),(2,1),(2,0),(3,2),(3,1),(3,0),\ldots\}.$$

A função característica desta linguagem é a função binária

$$\mathsf{c}_{\mathsf{Maior}}\,(x,y) = \begin{cases} 1 & \text{se } x > y \\ 0 & \text{caso contrário} \end{cases}$$

Mas a função unária

$$\overline{\mathsf{c}_{\mathsf{Maior}}} : z \mapsto \mathsf{c}_{\mathsf{Maior}}\left(\Pi_1(z), \Pi_2(z)\right)$$

verifica a igualdade

$$\overline{\mathsf{c}_{\mathsf{Maior}}}\,(\overline{x,y}) = \mathsf{c}_{\mathsf{Maior}}\,(x,y).$$

Ou seja, temos

$$x > y \Leftrightarrow \overline{\mathsf{c}_{\mathsf{Maior}}}\,(\overline{x,y}) = 1.$$

Conclusão: a função unária $\overline{\mathsf{c}_{\mathsf{Maior}}}$ *pode ser usada para estudar o predicado* binário maior.

Em particular, será que $\mathsf{maior}\,(3,1)$*? Ora*

$$\begin{aligned}\mathsf{maior}\,(3,1) &\Leftrightarrow \mathsf{c}_{\mathsf{Maior}}\,(3,1)=1\\ &\Leftrightarrow \overline{\mathsf{c}_{\mathsf{Maior}}}\left(\overline{3,1}\right)=1\\ &\Leftrightarrow \overline{\mathsf{c}_{\mathsf{Maior}}}\left(\Pi\,(3,1)\right)=1\\ &\Leftrightarrow \overline{\mathsf{c}_{\mathsf{Maior}}}\,(23)=1.\end{aligned}$$

Este exemplo mostra como encadear duas equivalências importantes

1. As questões sobre predicados podem ser resolvidas com linguagens (e *vice-versa*);
2. As questões sobre linguagens «n-árias» podem ser resolvidas com linguagens unárias (e *vice-versa*);

para estudar predicados n-ários apenas com linguagens unárias. Resumindo,

Teorema 4.1.8 *Podemos substituir o estudo dos predicados sobre números naturais pelo estudo de linguagens em* $\mathbb{N}$.

Códigos. Ainda em relação aos capítulos anteriores, lembremos que **função computável** é qualquer função parcial recursiva e **função p.r.** qualquer função primitiva recursiva.

Estamos interessados em associar **códigos** — números naturais — a certos aspectos das computações. Fazendo isso, podemos conceber programas que processem esses números, e obter *algoritmicamente* informação sobre as próprias computações. Isto é, associando as computações a números podemos conceber *programas que estudam as computações de outros programas*.

Retomando as funções da definição 3.3.2, definimos agora a função computável unária

$$\begin{array}{rccl}\mathsf{univ}: & \mathbb{N} & \to & \mathbb{N}\\ & z & \mapsto & \mathsf{u}\left(\Pi_1\,(z)\,,2^{\Pi_2(z)}\right)\end{array} \tag{4.9}$$

que verifica a igualdade

$$\mathsf{univ}\left(\overline{c,x}\right) = \mathsf{u}\left(c, 2^x\right) \tag{4.10}$$

Para que serve esta função? Consideremos um programa URM P e uma configuração inicial da memória $\boldsymbol{r} = (x, 0, 0, \ldots)$. Há duas questões importantes: (1) *Qual é o resultado da computação que P faz com configuração inicial $\boldsymbol{r}$?* e (2) *Esse resultado pode ser obtido por uma função unária?* Comecemos por codificar $c = \mathsf{cod}\left(P\right)$, $m = \mathsf{codmem}\left(\boldsymbol{r}\right) = 2^x$ e $z = \Pi\left(c, m\right)$. Agora basta calcular $\mathsf{univ}\left(z\right)$, que é o resultado da computação (*i.e.* o valor que está no primeiro registo quando esta termina). E sendo univ uma função unária, podemos responder afirmativamente a estas duas questões.

Composição. No anexo B.1, p.267 mostra-se que existe uma função primitiva recursiva unária «**comp**», tal que $\mathsf{comp}\left(z\right)$ é o código do programa obtido pela composição dos programas de códigos x e y. Isto é, *se* $z = \mathsf{comp}\left(\Pi\left(x, y\right)\right)$*, a computação que o programa* $\mathsf{dec}\left(z\right)$ *efectua a partir da configuração inicial M dos registos termina com o mesmo resultado que a computação do programa*

$$\left[\mathsf{dec}\left(x\right)\left[1, \ldots, n \to 1\right], \mathsf{dec}\left(y\right)\left[1 \to 1\right]\right]$$

a partir de M. Se algum dos programas $\mathsf{dec}\left(x\right)$ ou $\mathsf{dec}\left(y\right)$ não terminarem as suas computações, também $\mathsf{dec}\left(z\right)$ não termina. Por outras palavras, o programa URM $\mathsf{dec}\left(z\right)$ é a **composição** dos programas $\mathsf{dec}\left(x\right)$ com $\mathsf{dec}\left(y\right)$.

4.2 Enumeração das Funções Computáveis.

A codificação dos programas URM (no capítulo 2.4) define uma enumeração das funções computáveis, conforme ficou demonstrado pelo teorema 2.4.2.

Lembremos que se f for uma função computável existe uma prova axiomática de f e a compilação dessa prova produz um certo programa

F, que computa f. Ao código de Gödel desse programa chamamos também um **código** (ou **índice**) de f. Agora, recapitulando a definição 2.4.6, sendo f uma função computável com índice i, representamos f por ϕ_i; o domínio de f por $\mathcal{W}_i$; e o conjunto-imagem por $\mathcal{E}_i$: $\mathcal{W}_i = \mathsf{dom}\,(\phi_i)$ e $\mathcal{E}_i = \mathsf{im}\,(\phi_i)$. Isto é, $\mathcal{W}_i$ é o domínio da função que é computada pelo programa de código i e $\mathcal{E}_i$ é o conjunto imagem da mesma função.

Esta enumeração das funções computáveis não é a única possível: outras enumerações poderiam ser obtidas se, em vez das URM, tivéssemos escolhido as máquinas de Turing ou qualquer outro modelo de computação — digamos uma linguagem de programação como o *C* ou o *Python* — para descrever a classe das funções computáveis. Além da escolha da «linguagem de programação», também a própria função cod, para codificar os programas URM, poderia ter sido definida de forma diferente, de que resultaria ainda uma outra enumeração das funções computáveis.

Definição 4.2.1 (Enumeração das funções computáveis unárias) *A **enumeração das funções computáveis unárias** que se obtém pela sequência*

função computável → prova axiomática → programa URM → código de Gödel

é representada por

$$\Phi = \{\phi_i \mid i \in \mathbb{N}\}\,. \tag{4.11}$$

*Se ϕ_i for uma função computável então i é (um) seu **índice**; além disso, $\mathsf{dec}\,(i)$ é um programa URM que computa ϕ_i.*

Lembremos que uma função computável tem infinitas provas axiomáticas, portanto cada função computável tem infinitos índices.

Para uniformizar a notação representaremos a função universal unária, univ, por ϕ_{u} e a «composição», comp, por ϕ_{c}:

$$\phi_{\mathsf{u}} = \mathsf{univ} \tag{4.12}$$

$$\phi_{\mathsf{c}} = \mathsf{comp} \tag{4.13}$$

A enumeração das funções computáveis, juntamente com a função ϕ_{u}, tem uma consequência importante, e um pouco surpreendente: *define funções não computáveis*. Vejamos como:

Teorema 4.2.2 (T. Fundamental da Computabilidade) *A função*

$$\begin{array}{rccc} d: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & \phi_{\mathsf{u}}\left(\overline{x,x}\right)+1 \end{array} \tag{4.14}$$

não é computável.

Demonstração. Demonstraremos por absurdo: supondo que a função d é computável, seja i um seu índice, $d=\phi_i$. Por definição da função universal, para qualquer $x\in\mathbb{N}$, tem-se

$$d\left(x\right)=\phi_{\mathsf{u}}\left(\overline{i,x}\right).$$

Em particular, para $x=i$, fica

$$d\left(i\right)=\phi_{\mathsf{u}}\left(\overline{i,i}\right).$$

Mas, pela definição de d, tem de ser

$$d\left(i\right)=\phi_{\mathsf{u}}\left(\overline{i,i}\right)+1.$$

Estas duas últimas igualdades implicam $0=1$, um absurdo que resulta de supormos que d seria uma função computável. Portanto d não é computável. □

Embora esta demonstração assinale um facto importante, não proporciona ferramentas para aprofundar o estudo da computação. O próximo resultado sobre a enumeração é conhecido como o **teorema s-m-n** ou **teorema da parametrização** e tem consequências importantes, de aplicação técnica e teórica.

Teorema 4.2.3 (O teorema s-m-n ou da parametrização)
Existe uma função computável total $s:\mathbb{N}^2\to\mathbb{N}$ *tal que, para quaisquer* $i\in\mathbb{N}$, $\boldsymbol{x}\in\mathbb{N}^m$, $\boldsymbol{y}\in\mathbb{N}^n$, *se tem*

$$\phi_{s(i,\overline{\boldsymbol{x}})}\left(\overline{\boldsymbol{y}}\right)=\phi_i\left(\overline{\boldsymbol{x},\boldsymbol{y}}\right). \tag{4.15}$$

Demonstração. Comecemos por mostrar que existe uma função $h : \mathbb{N} \to \mathbb{N}$ que verifica a igualdade

$$\phi_{h(n)}(z) = \overline{n, z}$$

e que é primitiva recursiva. Sejam

$$\begin{array}{rcl} f(z) & = & \overline{0, z} \\ g(\overline{n, z}) & = & \overline{n+1, z}. \end{array}$$

Como f e g são funções p.r. (pois podem ser definidas por composição de outras funções p.r.), existem índices j e k para as funções f e g, respectivamente: $f = \phi_j$ e $g = \phi_k$.

Define-se agora, por recursão, a função $h : \mathbb{N} \to \mathbb{N}$

$$\left\{ \begin{array}{rl} h(0) & = j \\ h(n+1) & = \phi_{\mathsf{c}}\left(\overline{k, h(n)}\right) \end{array} \right. .$$

A função h é primitiva recursiva pois ϕ_{c} e Π são funções primitivas recursivas e a definição de h está de acordo com o esquema da recursão (3.1.5).

(**exercício**:) Por indução em n, tem-se que, para cada $n, z \in \mathbb{N}$,

$$\phi_{h(n)}(z) = \overline{n, z}.$$

Agora, para cada $i, \overline{\boldsymbol{x}}, \overline{\boldsymbol{y}} \in \mathbb{N}$,

$$\begin{array}{rcl} \phi_i(\overline{\boldsymbol{x}, \boldsymbol{y}}) & = & \phi_i\left(\phi_{h(\overline{\boldsymbol{x}})}(\overline{\boldsymbol{y}})\right) \\ & = & \left(\phi_i \circ \phi_{h(\overline{\boldsymbol{x}})}\right)(\overline{\boldsymbol{y}}) \\ & = & \phi_s(\overline{\boldsymbol{y}}) \end{array}$$

em que

$$s = \phi_{\mathsf{c}}\left(\overline{i, h(\overline{\boldsymbol{x}})}\right)$$

Este índice s, que depende de i e de $\boldsymbol{x}$, verifica a relação

$$\phi_s(\overline{\boldsymbol{y}}) = \phi_i(\overline{\boldsymbol{x}, \boldsymbol{y}})$$

do enunciado do teorema. Portanto, definindo a função

$$s:\ (a,b)\mapsto \phi_{\mathsf{c}}\left(\overline{a,h\,(b)}\right)$$

resulta que s é uma função computável total que verifica a igualdade (4.15). □

Vejamos algumas consequências do teorema s-m-n. Primeiro, a igualdade da equação (4.15) permite «reduzir» a função (de aridade $m+n$) $f\,(\boldsymbol{x},\boldsymbol{y})=\phi_i\,(\overline{\boldsymbol{x},\boldsymbol{y}})$ à função (de aridade n) $g_{\boldsymbol{x}}\,(\boldsymbol{y})=\phi_{s(i,\overline{\boldsymbol{x}})}\,(\overline{\boldsymbol{y}})$, fixando o primeiros m argumentos. Por exemplo, sabendo que a função binária $\mathsf{soma}:\ x,y\mapsto x+y$ é computável, pelo teorema s-m-n, também é computável a função unária $\mathsf{soma}_{735}:y\mapsto 735+y$. Além disso, a função que resulta desta redução é computável (pois a função s é computável total) e, note-se bem, *não depende da função f em causa*, pois o índice i de f é um dos argumentos de s.

A segunda consequência, mais subtil e de maior fôlego, parte da observação anterior para justificar que se passe das operações computáveis sobre números naturais (que os programas URM e as funções computáveis fazem) para *operações computáveis sobre programas e funções computáveis*! E podemos dar um sentido preciso a este prolongamento da interpretação de «*computável*» pois a cada programa corresponde, via codificação, um número natural e *vice-versa*. Assim, «confundindo» os programas com os seus códigos[1] — ou, dito de outra forma, as funções computáveis com os seus índices — um programa/função que opere sobre números naturais/índices também opera sobre programas/funções.

Vejamos um exemplo de uma operação sobre programas/funções que é «computável»:

Exemplo 4.2.4 A soma de funções computáveis é computável.

[1]Esta ideia tem lugar pela primeira vez com Kurt Gödel, ao demonstrar, em 1936, o seu famoso, profundo, teorema (muitas vezes mal compreendido e pior citado) sobre a impossibilidade de demonstrar todas as verdades em qualquer teoria formal que contenha a aritmética.

Primeiro, note-se que este enunciado não diz respeito *à soma de números naturais. Não falamos da função*

$$\begin{array}{rccc} \mathsf{soma}: & \mathbb{N}\times\mathbb{N} & \to & \mathbb{N} \\ & (x,y) & \mapsto & x+y \end{array}$$

mas sim da soma de funções computáveis*, a operação*

$$\begin{array}{rccc} \mathsf{SOMA}: & \mathcal{C}\times\mathcal{C} & \to & \mathcal{C} \\ & (f,g) & \mapsto & f+g \end{array}$$

onde, por sua vez, a função $f+g\in\mathcal{C}$ *está definida por*

$$\begin{array}{rccc} f+g: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & f(x)+g(x)=\mathsf{soma}\left(f(x),g(x)\right) \end{array}.$$

Resumindo, $\mathsf{SOMA}(f,g)$ *é uma função e será computável se ambas as funções* f *e* g *forem computáveis, enquanto que* $\mathsf{soma}(x,y)$ *é «apenas» um número natural.*

Nos capítulos anteriores justificou-se que a função soma *é computável construindo uma prova axiomática. Para justificar agora que* SOMA *é computável, consideremos o esquema seguinte:*

1. *Supondo que o índice de* f *é* i *e o índice de* g *é* j, $f=\phi_i$, $g=\phi_j$;
2. *quer-se mostrar que existe um índice* k *tal que*
$$\phi_k\left(\overline{i,j,x}\right)=\phi_i(x)+\phi_j(x)=\mathsf{SOMA}\left(\phi_i,\phi_j\right)(x)$$
3. *pois, nesse caso, pelo teorema s-m-n,*
$$\phi_k\left(\overline{i,j,x}\right)=\phi_{s\left(k,\overline{i,j}\right)}(x)$$
4. *e portanto, para cada* i,j, $s\left(k,\overline{i,j}\right)$ *é um índice da função*
$$\mathsf{SOMA}\left(\phi_i,\phi_j\right)=\phi_i+\phi_j:\ x\mapsto\phi_i(x)+\phi_j(x),$$
isto é, a função $(i,j)\mapsto s\left(k,\overline{i,j}\right)$ *define o índice da «soma das funções de índices* i *e* j*»,* i.e., $\mathsf{SOMA}\left(\phi_i,\phi_j\right)$.

A dificuldade (e especificidade deste exemplo) está no ponto 2, em determinar um índice k naquelas condições particulares. Porém, para tal, é suficiente verificar que a função $(i,j,x) \mapsto \phi_i(x) + \phi_j(x)$ é computável pois cada função computável (na versão unária) tem, pelo menos, um índice. Usando a função universal ϕ_{u},

$$\phi_i(x) + \phi_j(x) \quad = \quad \phi_{\mathsf{u}}\left(\overline{i,x}\right) + \phi_{\mathsf{u}}\left(\overline{j,x}\right)$$

e, como as funções ϕ_{u}, Π e soma *são computáveis, pela igualdade anterior também a função $(i,j,x) \mapsto \phi_i(x) + \phi_j(x)$ é computável.*

O teorema s-m-n tem ainda um papel importante nas demonstrações de (semi-) indecidibilidade. Tornaremos a encontrá-lo, desempenhando um papel discreto mas determinante, no exemplo 4.3.10.

4.3 Linguagens Decidíveis e Semi-decidíveis

Lembremos, resumidamente, da subsecção 2.2, algumas definições e propriedades sobre linguagens.

Uma linguagem $A \subseteq \mathbb{N}$ é **decidível** (ou **recursiva**) se a sua função característica

$$\mathsf{c}_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{caso contrário} \end{cases}$$

for computável[2]. Uma linguagem é **indecidível** se não for decidível.

A intersecção, união e diferença de linguagens decidíveis são, ainda, linguagens decidíveis.

Uma linguagem A é **semi-decidível** (ou **recursivamente enumerável**, abreviadamente **r.e.**) se a função

$$\mathsf{c}'_A(x) = \begin{cases} 1 & \text{se } x \in A \\ \perp & \text{caso contrário} \end{cases}$$

for computável. Uma linguagem é **semi-indecidível** se não for semi-decidível.

[2]A função característica de uma linguagem é, por definição, total.

Se a linguagem A e a sua complementar $\overline{A} = \mathbb{N} - A$ forem semi-decidíveis então A (e $\overline{A}$) é decidível.

Para uma linguagem *decidível* A, sabemos que existe um certo programa que, para qualquer argumento $x \in \mathbb{N}$, termina em qualquer uma das duas possibilidades $x \in A$ ou $x \notin A$.

Para uma linguagem *semi-decidível* Z, «apenas» sabemos que existe um programa (digamos $\texttt{Teste}_Z$) que *só* termina se $x \in Z$: para descobrir se um $x \in \mathbb{N}$ qualquer está em Z, teremos de usar o programa $\texttt{Teste}_Z$. Mas, no caso de $x \notin Z$, corremos o risco da computação $\texttt{Teste}_Z\,(x)$ não terminar e portanto de esperar indefinidamente pela resposta.

O dilema que se põe é o seguinte: supondo que queremos saber se $294 \in Z$. Dispondo do programa $\texttt{Teste}_Z$, iniciamos a computação $\texttt{Teste}_Z\,(294)$, aguardamos 10, 100, um milhão de passos mas a computação ainda não terminou... O que podemos concluir? Não podemos afirmar que $294 \in Z$, pois tal só acontece se a computação terminar e isso ainda não aconteceu. Mas também não podemos afirmar que $294 \notin Z$ pois embora tenhamos efectuado 1000000 de passos sem que a computação termine, pode acontecer que termine no passo 1000001 ou no passo $2007^{10^{15}}$, levando à conclusão que, afinal, $294 \in Z$.

Para as linguagens decidíveis temos a garantia que, esperando, encontraremos uma resposta, mas as linguagens semi-decidíveis podem prender-nos em computações sem fim.

Esta pequena revisão mostrou algumas operações interessantes (união, intersecção, diferença) para construir linguagens decidíveis a partir de outras, também decidíveis. Trataremos, de seguida, de um problema relacionado: *como descobrir linguagens indecidíveis.*

4.3.1 Uma Linguagem Indecidível

Começaremos o estudo das linguagens indecidíveis por definir uma linguagem, construída por diagonalização, que tem um papel especial no estudo da computação (ver mais sobre a diagonalização em A.3, p.261). Para as linguagens *semi*-indecidíveis é necessária um pouco mais de técnica, pelo que estas serão tratadas mais tarde, na secção 4.3.4.

Teorema 4.3.1 (Uma linguagem indecidível) *A linguagem*

$$\mathsf{H} \;=\; \{x \mid \phi_x(x)\downarrow\} \tag{4.16}$$

é indecidível[3]*.*

Demonstração. Por absurdo, supondo que H seria decidível, então a função

$$\mathsf{c}_{\mathsf{H}}(x) = \begin{cases} 1 & \text{se } \phi_x(x)\downarrow \\ 0 & \text{caso contrário} \end{cases} \tag{4.17}$$

seria computável (e total). Mas então, também a função

$$h(x) = \begin{cases} 0 & \text{se } \mathsf{c}_{\mathsf{H}}(x) = 0 \\ \perp & \text{caso contrário} \end{cases}$$

teria de ser computável. Portanto, h teria um certo índice i (*i.e.*, $h = \phi_i$) e resultaria então que

$$\begin{aligned} \phi_i(i) = 0 \;&\Leftrightarrow\; h(i) = 0 \\ &\Leftrightarrow\; \mathsf{c}_{\mathsf{H}}(i) = 0 \\ &\Leftrightarrow\; i \notin \mathsf{H} \\ &\Leftrightarrow\; \phi_i(i)\uparrow \end{aligned}$$

o que é uma contradição.

Assim h, e consequentemente c_{H}, não podem ser funções computáveis. Logo, H não é uma linguagem decidível. □

4.3.2 A Intercalação

Apesar da linguagem H não ser decidível, é semi-decidível. Para verificar que pode ser **enumerada** por uma função computável — isto é, que existe uma função computável $f : \mathbb{N} \to \mathsf{H}$ tal que $\mathsf{H} = \mathsf{im}f =$

[3]Também é costume definir $\mathsf{H} = \{x \mid x \in \mathcal{W}_x\}$, equivalente à forma enunciada.

$\{f(0), f(1), \ldots\}$ — empregaremos uma técnica conhecida, na literatura anglo-saxónica, por *dovetail*, que traduzida directamente para o português significa «a cauda da rola». Neste livro será designada por «**intercalação**». A intercalação emprega-se quando é necessário pesquisar os resultados de várias computações, sendo algumas destas potencialmente infinitas.

Por exemplo, consideremos a função (computável)

$$f(x) = \begin{cases} \perp & \text{se } x = 1 \\ 2x & \text{caso contrário} \end{cases} \tag{4.18}$$

e suponhamos que queremos resolver, computacionalmente, a seguinte questão: «Existe um x tal que $f(x) = 4$?». Um tentativa ingénua para resolver este problema seria, resumidamente, assim:

```
1.  def DecisorIngénuo :
2.      #  Como f é computável, existe F que a computa
3.      x ← 0
4.      enquanto x ∈ ℕ :    #  ciclo infinito...
5.          se F(x) ↓ 4 :
6.              retorna SIM
7.          x ← x + 1
8.      retorna NÃO
```

O problema com este algoritmo é que, na linha 5, onde se calcula $F(x)$, no valor $x = 1$ a computação não termina e, portanto, não se chega a verificar o caso $x = 2$. No entanto, nesse valor ter-se-ia uma resposta positiva: $F(2) \downarrow 4$. *Então, como resolver este problema?* A solução consiste em observar sucessivamente cada passo elementar nas computações envolvidas.

Lembremos, dos capítulos anteriores, que cada computação é uma sucessão de passos em que cada passo depende do anterior e que, eventualmente, a sucessão termina. Ainda para o exemplo anterior, podemos determinar, *por esta ordem*, o primeiro passo de $F(0)$, o primeiro passo de $F(1)$ e o segundo de $F(0)$, o primeiro de $F(2)$, o segundo de $F(1)$ e o terceiro de $F(0)$, *etc.* Desta forma, ao fim de algum tempo (finito), percorreram-se todos os passos necessários para terminar a computação

de $F(2)$ e responder afirmativamente ao problema posto. Mais importante, intercalando as computações, o facto da computação de $F(1)$ não terminar não compromete a pesquisa. A figura 4.1 dá-nos uma representação visual da intercalação.

$\exists x:\ F(x)\downarrow 4?$	1.º	2.º	3.º	4.º	$\cdots$
$x=0$	1	3	6	10	$\cdots$
$x=1$	2	5	9	$\cdots$	*computação infinita*
$x=2$	4	8	$\cdots$		$x=2$ é a solução.
$x=3$	7	$\cdots$			
$\vdots$	$\cdots$				

Figura 4.1: A pesquisa por *intercalação*: cada linha tem todos os passos da computação de um certo $F(x)$ e cada coluna um passo dessas computações. Assim, na célula $(0,1)$ está o primeiro passo da computação de $F(0)$ e em $(2,3)$ o terceiro passo da computação de $F(2)$. Os números nas células indicam a ordem pela qual estas são visitadas: $(0,1) \to (1,1) \to (0,2) \to (2,1) \to (1,2) \to (0,3) \to \cdots$. O terceiro passo da computação de $F(1)$ será o nono a ser visitado. A região numerada assemelha-se (vagamente) à cauda de uma rola, e daí o nome (em inglês) desta técnica.

Note-se que esta técnica de pesquisa não tem a garantia de terminar sempre, independentemente do problema em questão. Por exemplo, as pesquisas «Existe x tal que $f(x)=3?$» ou «Existe x tal que $f(x)\uparrow?$» não terminam. O que fica garantido é que *se o problema tiver solução então esta será encontrada*, mesmo que esteja «acompanhada» por vizinhos que levam a computações infinitas!

A questão seguinte é saber *como usar algoritmicamente a intercalação*. Embora a tabela anterior sugira, intuitivamente, que tal é possível, falta definir uma função computável apta à aplicação da intercalação. Para isso, usaremos a função (binária, primitiva recursiva) $\Pi : \mathbb{N}^2 \to \mathbb{N}$, as respectivas projecções $\Pi_1, \Pi_2 : \mathbb{N} \to \mathbb{N}$ (em A.3, p.259) e a função p.r. resultado, definida no teorema seguinte.

Teorema 4.3.2 *Existe uma função primitiva recursiva,* **resultado**, *tal que, para quaisquer* $x, i \in \mathbb{N}$*:*

1. *se* $\mathsf{resultado}\,(i,x,t) \neq 0$ *então*

$$\mathsf{resultado}\,(i,x,t) = \phi_i\,(x) + 1$$

2. $\phi_i\,(x)\downarrow$ *se, e só se, existe* t_x *tal que para cada* $t \geq t_x$*,*

$$\mathsf{resultado}\,(i,x,t) \neq 0.$$

Demonstração. Definimos

$$\mathsf{resultado}\,(i,x,t) \;=\; \begin{cases} \mathsf{S}\left(\phi_{\mathsf{u}}\left(\overline{i,x}\right)\right) & \text{se } \mathsf{final}\,(i,x,t) = 1 \\ 0 & \text{caso contrário} \end{cases} \tag{4.19}$$

onde

$$\mathsf{final}\,(i,x,t) \;=\; \overline{\mathsf{sg}}\,(\mathsf{instr}\,(\mathsf{comput}\,(\mathsf{config}\,(i,1,x)\,,t))) \tag{4.20}$$

é uma função primitiva recursiva e indica se a computação da função ϕ_i, com argumento x, é computada em t passos elementares.

Com efeito, $\mathsf{instr}\,(s)$ indica a instrução seguinte na computação s, valendo 0 se essa computação terminou (*i.e.* não há próxima instrução). Portanto,

$$\overline{\mathsf{sg}}\,(\mathsf{instr}\,(s)) = 1$$

se, e só se, a computação s terminou.

Por outro lado, $c = \mathsf{comput}\,(s,t)$ é (o código de) a computação que resulta ao fim de t passos a partir da configuração (de código) $s = \mathsf{config}\,(i,1,x)$ do programa (de código) i com configuração inicial dos registos (de código) x.

Supondo agora que a computação de $\phi_i\,(x)$ é efectuada em n passos elementares, para qualquer $t < n$, $\mathsf{final}\,(i,x,t) = 0$ e, para $t \geq n$, $\mathsf{final}\,(i,x,t) = 1$. Isto é, $\mathsf{final}\,(i,x,t)$ indica se a computação do programa

(de código) i, a partir da configuração inicial (de código) x termina em t passos elementares.

Vejamos agora que a função $\mathsf{resultado}$ verifica as condições (1) e (2) do teorema. Sejam $i, x \in \mathbb{N}$,

1. Se $\mathsf{resultado}\,(i,x,t) = z \neq 0$, então, por definição, $\mathsf{final}\,(i,x,t) = 1$: bastam t passos para a computação do programa (de código) i, com configuração inicial (de código) x, terminar. Portanto, existe um certo y tal que $\phi_i\,(x) \downarrow y$ ou seja, $\phi_{\mathsf{u}}\left(\overline{i,x}\right) \downarrow y$. Resta confirmar que $z = y{+}1$. Ora, pela definição de $\mathsf{resultado}$, $z = \mathsf{S}\left(\phi_{\mathsf{u}}\left(\overline{i,x}\right)\right) = y{+}1$.

2. Se $\phi_i\,(x) \downarrow$ então a computação do programa i com argumento x termina. Seja então t_x o número de passos elementares necessários para essa computação terminar. Assim, para qualquer $t \geq t_x$, $\mathsf{final}\,(i,x,t) = 1$. Como o menor valor que $\phi_{\mathsf{u}}\left(\overline{i,x}\right)$ pode tomar é 0, conclui-se que $\mathsf{resultado}\,(i,x,t) \geq \mathsf{S}\,(0) = 1$. Portanto, $\mathsf{resultado}\,(i,x,t) \neq 0$.

   Reciprocamente, se $\mathsf{resultado}\,(i,x,t_x) \neq 0$ então $\mathsf{final}\,(i,x,t_x) = 1$. Portanto, com t_x passos a computação do programa (de código) i com configuração inicial x termina. Além disso, essa computação também termina com $t > t_x$ passos elementares. Mas então $\phi_{\mathsf{u}}\left(\overline{i,x}\right) \downarrow$ e $\phi_i\,(x) \downarrow$.

□

A função $\mathsf{resultado}$ é mais do que um teste ao número de passos elementares necessários para terminar um programa: se $\mathsf{resultado}\,(i,x,t) = 0$ então são necessários mais do que t passos elementares para terminar a computação de $\phi_i\,(x)$. Pelo contrário, se $\mathsf{resultado}\,(i,x,t) > 0$ então bastam t passos para terminar a computação de $\phi_i\,(x)$ e ainda podemos obter o valor de $\phi_i\,(x)$:

$$\phi_i\,(x) \quad = \quad \mathsf{pred}\,(\mathsf{resultado}\,(i,x,t))\,. \tag{4.21}$$

Exemplo 4.3.3 *Vejamos como usar a função* $\mathsf{resultado}$ *para fazer a intercalação. A função* f*, na equação (4.18), é computável (por* F*),*

embora não seja total (pois $f(1) \uparrow$*), tendo um certo índice, digamos* $i = \mathsf{cod}(F)$*. A função*

$$g(x,t) = \begin{cases} 1 & \text{se } \mathsf{resultado}(i,x,t) = 5 \\ 0 & \text{caso contrário.} \end{cases}$$

é primitiva recursiva. Note-se que $g(x,t) = 1$ *significa que* $F(x) \downarrow 4$ *em* t *passos e* $g(x,t) = 0$ *o contrário: ou* $F(x) \downarrow y$ *com* $y \neq 4$ *ou então* $F(x)$ *precisa de mais do que* t *passos para terminar.*

Agora, a função (nulária[4]*)*

$$h() = \mu_z \left[g\left(\Pi_1(z), \Pi_2(z)\right) \right]$$

é computável. Se $h() \downarrow$*, a resposta ao problema «Existe um* x *tal que* $f(x) = 4$*?» é sim. Pelo contrário, se* $h() \uparrow$*, a resposta é não. Assim, temos uma prova de que este problema é semi-decidível.*

Importa registar, deste exemplo, que é possível tratar computacionalmente, com as ferramentas já definidas, a técnica da pesquisa por intercalação. Uma aplicação importante da intercalação é o seguinte:

Teorema 4.3.4 *A linguagem* $\mathsf{H} = \{x \mid \phi_x(x) \downarrow\}$ *é semi-decidível.*

Demonstração. A linguagem H é semi-decidível pois

1. a função
$$1 + \mathsf{Z}\left(\mu_t \left[\mathsf{resultado}(x,x,t) > 0\right]\right)$$
é computável e

2.
$$\mathsf{c'}_{\mathsf{H}}(x) = 1 + \mathsf{Z}\left(\mu_t \left[\mathsf{resultado}(x,x,t) > 0\right]\right)$$

[4] As funções nulárias, isto é, com 0 argumentos, correspondem a programas URM que ignoram os argumentos dados.

Primeiro, a função definida no ponto 1 é computável pois resulta da aplicação da composição e minimização das funções $1, \mathsf{Z}$, soma e $\mathsf{resultado}$, que sabemos serem computáveis.

Segundo, seja $x \in \mathsf{H}$. Então $\phi_x(x) \downarrow$, portanto, para algum t,

$$\mathsf{resultado}(x, x, t) \neq 0.$$

Podemos então garantir que

$$\mu_t [\mathsf{resultado}(x, x, t) > 0] \downarrow$$

e que

$$1 + \mathsf{Z}(\mu_t [\mathsf{resultado}(x, x, t) > 0]) = 1.$$

Por outro lado, se $x \notin \mathsf{H}$, $\phi_x(x) \uparrow$ logo

$$\mu_t [\mathsf{resultado}(x, x, t) > 0] \uparrow$$

e também

$$1 + \mathsf{Z}(\mu_t [\mathsf{resultado}(x, x, t) > 0]) \uparrow .$$

Conclui-se que

$$\mathsf{c'}_{\mathsf{H}}(x) = 1 + \mathsf{Z}(\mu_t [\mathsf{resultado}(x, x, t) > 0]) .$$

é computável e, logo, H é semi-decidível. □

A demonstração anterior assenta numa aplicação da intercalação. Para cada $x \in \mathbb{N}$ são testados $t = 1, 2, \ldots$ passos na computação de $\phi_x(x)$. Se esta computação terminar, então $\mathsf{resultado}(x, x, t) > 0$ para algum t suficientemente grande e $\mathsf{c'}_{\mathsf{H}}(x) \downarrow$. Por outro lado, se $\phi_x(x) \uparrow$ então, para qualquer valor de t, $\mathsf{resultado}(x, x, t) = 0$ e, também, $\mathsf{c'}_{\mathsf{H}}(x) \uparrow$.

Temos ainda a seguinte consequência imediata e importante do teorema 4.3.4, sobre o complementar de H:

Corolário 4.3.5 *A linguagem* $\overline{\mathsf{H}} = \{x \mid \phi_x(x) \uparrow\}$ *é semi-indecidível.*

Demonstração. Por absurdo, se, além de H, também $\overline{\mathsf{H}}$ fosse semi-decidível, pelas propriedades das linguagens decidíveis e semi-decidíveis, concluir-se-ia que H teria de ser decidível, contradizendo o teorema 4.3.1.

Portanto, $\overline{\mathsf{H}}$ tem de ser semi-indecidível. □

Nesta secção introduzimos a técnica de pesquisa por intercalação para a análise de linguagens semi-decidíveis. Numa primeira aplicação (não muito sofisticada) encontrámos uma linguagem semi-decidível (H) e uma linguagem semi-indecidível ($\overline{\mathsf{H}}$). Equipados com a técnica da intercalação e com a função $\mathsf{resultado}$, podemos agora estudar as propriedades das linguagens semi-indecidíveis e obter, daí, as ferramentas necessárias para o estudo da semi-(in)decidibilidade de outras linguagens.

4.3.3 Linguagens Semi-decidíveis

O próximo teorema estabelece duas importantes caracterizações de qualquer linguagem semi-decidível, pois determina uma relação crucial entre linguagens semi-decidíveis e funções computáveis:

> *Qualquer linguagem semi-decidível é a imagem e o domínio de funções computáveis, e vice-versa, as imagem e os domínios das funções computáveis são linguagens semi-decidíveis.*

Teorema 4.3.6 *Seja $A \subseteq \mathbb{N}$ uma linguagem. As seguintes afirmações são equivalentes:*

1. *A é semi-decidível;*
2. *Existe uma função computável f tal que $A = \mathsf{dom}\,(f)$;*
3. *Se $A \neq \emptyset$, existe uma função computável total g tal que $A = \mathsf{im}\,(g)$;*

É devido à propriedade da alínea 3 do teorema 4.3.6 que as linguagens semi-decidíveis também se designam **recursivamente enumeráveis** (**r.e.**). Com efeito, graças a esta propriedade, se A for semi-decidível

então pode ser *enumerada* por uma função computável, *i.e.*, para alguma função computável g,

$$A = \{g(0), g(1), \ldots\}.$$

Note-se que descrevendo a linguagem A desta forma, esta *não tem de estar ordenada*, *i.e.* a função g *não é, necessariamente, crescente.*

Demonstração. Para provar as equivalências basta mostrar as seguintes três implicações: (1) $\Rightarrow$ (2) $\Rightarrow$ (3) $\Rightarrow$ (1).

(1) $\Rightarrow$ (2): supondo que A é r.e., queremos encontrar uma função computável f tal que $A = \{x \in \mathbb{N} \mid f(x) \downarrow\}$. Pela definição de linguagem r.e., basta tomar $f = c'_A$.

(2) $\Rightarrow$ (3): Seja então $A = \mathsf{dom}(f)$, com f computável e i um índice de f. Então $A = \mathsf{dom}(\phi_i) = \mathcal{W}_i$ e tem de se mostrar que alguma função computável total, g, enumera $\mathcal{W}_i$. Tal função será definida por recursão: seja g definida por

$$\begin{aligned} g(0) &= \Pi_1(\mu_z[\mathsf{resultado}(i, \Pi_1(z), \Pi_2(z)) > 0]) \\ g(n+1) &= \begin{cases} g(n) & \text{se } \mathsf{resultado}(i, \Pi_1(n+1), \Pi_2(n+1)) = 0 \\ \Pi_1(n+1) & \text{se } \mathsf{resultado}(i, \Pi_1(n+1), \Pi_2(n+1)) > 0 \end{cases} \end{aligned}$$

A base $g(0)$ da recursão determina o «primeiro» $x \in \mathcal{W}_i$ («primeiro», seguindo a numeração de $\mathbb{N}^2$ induzida por Π). O passo $g(n+1)$ alarga, a valores sucessivamente maiores, a pesquisa — por intercalação — de elementos em $\mathcal{W}_i$. Sempre que alguma computação $\phi_i(\Pi_1(n+1))$ termina (no caso $\mathsf{resultado}(i, \Pi_1(n+1), \Pi_2(n+1)) > 0$), o valor de g muda para $\Pi_1(n+1)$ e permanece constante até ser encontrado outro valor em $\mathcal{W}_i$.

Agora, se $A \neq \emptyset$, então $g(0) \downarrow$ e, portanto, g é total. Além disso, se $x \in \mathcal{W}_i$, então, para um n suficientemente grande, $g(n) = x$ e, por outro lado, se $x \notin \mathcal{W}_i$, nenhum n faz $g(n) = x$. Portanto $A = \mathcal{W}_i = \mathsf{im}(g)$.

(3) $\Rightarrow$ (1): seja g uma função computável total e $A = \{g(x) \mid x \in \mathbb{N}\}$. Queremos mostrar que existe uma função computável c tal que $x \in A$ se, e só se, $c(x) \downarrow 1$. Ora, aqui basta fazer

$$c(x) = 1 + \mathsf{Z}(\mu_y[g(y) = x])$$

pois, se existir um y tal que $g(y) = x$, então $c(x) \downarrow 1$ e caso contrário, se para qualquer y, $g(y) \neq x$, então $\mu_y [g(y) = x] \uparrow$ e também $c(x) \uparrow$. □

Veremos de seguida algumas propriedades sobre linguagens definidas pelo quantificador existencial, $\exists$, que podem ser demonstradas, como exercícios, aplicando a intercalação e a função resultado.

Teorema 4.3.7 *Uma linguagem A é semi-decidível se, e só se, existir uma linguagem decidível B tal que*

$$x \in A \leftrightarrow \exists y : \overline{x, y} \in B$$

Demonstração. Exercício. □

Teorema 4.3.8 *Se uma linguagem A for semi-decidível, também é a linguagem*

$$\{x \mid \exists y : \overline{x, y} \in A\}$$

Demonstração. Exercício. □

4.3.4 Reduções

Para provar que uma linguagem dada, A, é decidível, basta mostrar que a sua função caraterística, c_A, é computável — por exemplo, construindo um programa URM que compute c_A. Analogamente, para mostrar que a linguagem A é r.e. basta encontrar um programa URM que compute $\mathsf{c'}_A$. Mas, como se fará se a linguagem dada não for (semi-) decidível? Nesse caso, nenhum programa computa c_A (ou $\mathsf{c'}_A$).

Conhecemos já uma linguagem indecidível e semi-decidível, H, e também sabemos que $\overline{\mathsf{H}}$ é semi-indecidível. Veremos, de seguida, uma técnica, a *redução*, que permite abordar problemas deste tipo, construindo relações adequadas entre a linguagem em questão (digamos A) e H ou $\overline{\mathsf{H}}$.

Definição 4.3.9 (redução)
Uma ***redução*** *da linguagem A para a linguagem B é uma função computável total $f : \mathbb{N} \to \mathbb{N}$ tal que*

$$x \in A \text{ se, e só se, } f(x) \in B. \tag{4.22}$$

Nesse caso, escreve-se $A \propto_f B$.

Com reduções é possível estender a informação sobre a (semi-) decidibilidade de uma linguagem para outra: se B for (semi-) decidível então, pela redução f, também A o será, pois $\mathsf{c}'_A = \mathsf{c}'_B \circ f$ e, também, $\mathsf{c}_A = \mathsf{c}_B \circ f$. Reciprocamente, se A for (semi-) indecidível, então B é (semi-) indecidível.

A tabela da figura 4.2 resume os sentidos em que se pode concluir sobre a (semi-) decidibilidade.

A	$\propto_f$	B
não decidível	$\Rightarrow$	não decidível
não r.e.	$\Rightarrow$	não r.e.
decidível	$\Leftarrow$	decidível
r.e.	$\Leftarrow$	r.e.

Figura 4.2: Reduzir a (semi-) decidibilidade de uma linguagem à (semi-) decidibilidade de outra.

Exemplo 4.3.10 (Ver, também, o exercício 4.5.24, p.169) A linguagem

$$\mathsf{I} = \{x \mid \phi_x \equiv \mathsf{id}\}$$

é decidível? *A função $\mathsf{Z} : \mathbb{N} \to \mathbb{N}, x \mapsto 0$ é computável. Seja agora $\psi(\overline{x,y}) = y + \mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x}))$. Como ψ é computável, existe um certo índice i tal que $\psi = \phi_i$. Agora, pelo teorema s-m-n, seja $j = s(i,x)$, tal que $\phi_j(y) = \psi(\overline{x,y})$. Repare-se que se $x \in \mathsf{H}$ então $\phi_j \equiv \mathsf{id}$ e que, se $x \notin \mathsf{H}$ então $\phi_j \equiv \perp$. Portanto, $x \mapsto j = s(i,x)$ é uma redução de H para I: $x \in \mathsf{H} \Leftrightarrow j \in \mathsf{I}$. Como H é indecidível, também I o é.*

Este exemplo permite-nos delinear um método geral para a resolução de exercícios sobre a decidibilidade. Neste método usaremos a **função totalmente indefinida**

$$\begin{array}{rccc} \omega: & \emptyset & \to & \mathbb{N} \\ & x & \mapsto & \bot \end{array} \tag{4.23}$$

que diverge em qualquer argumento mas é computável, por exemplo, pelo programa `Omega` (ver a definição na equação 2.19, p.28).

Supondo que pretendemos mostrar que uma certa linguagem X é indecidível:

1. Se ω não está em X (mais precisamente, se nenhum dos índices de ω está em X), reduziremos $\mathsf{H} \propto_h X$ seguindo estes passos:
   (a) Seja f uma função (com algum índice) em X. Isto é, $f = \phi_i$ e $i \in X$. Esta função, f, é computável;
   (b) Seja $\psi(x,y) = f(y) + \mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x}))$. Também ψ é computável. Além disso, se, para um certo x, $\phi_{\mathsf{u}}(\overline{x,x}) \uparrow$ então, para esse x, a função $y \mapsto f(y) + \mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x}))$ coincide com ω;
   (c) Portanto, existe um certo índice i tal que $\psi(x,y) = \phi_i(\overline{x,y})$;
   (d) Pelo teorema s-m-n,
   $$\psi(x,y) = \phi_i(\overline{x,y}) = \phi_{s(i,x)}(y)$$
   (e) Seja $h : x \mapsto s(i,x)$;
   (f) Resulta que $\mathsf{H} \propto_h X$ é uma redução de H para X. Confirmemos:

   Se $x \in \mathsf{H}$ então $h(x) \in X$. Se $x \in \mathsf{H}$ então
   $$\mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x})) \downarrow 0$$
   porque $\phi_{\mathsf{u}}(\overline{x,x}) \downarrow$, logo
   $$\psi(x,y) = f(y)$$
   e portanto $h(x) \in X$;

Se $h(x) \in X$ **então** $x \in \mathsf{H}$**.** Verifiquemos a implicação equivalente, *se* $x \notin \mathsf{H}$ *então* $h(x) \notin X$: Se $x \notin \mathsf{H}$ então

$$\mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x})) \uparrow$$

porque $\phi_{\mathsf{u}}(\overline{x,x}) \uparrow$, logo

$$\psi(x,y) \uparrow$$

e portanto $h(x) \notin X$.

(g) Como H é indecidível, conclui-se que X também é indecidível.

2. Se (algum índice de) $\omega \in X$, reduziremos $\mathsf{H} \propto_h \overline{X}$:

   (a) Basta seguir para $\overline{X}$ o procedimento descrito para X.

   (b) Sabendo que $\overline{X}$ é indecidível, pelas propriedades das linguagens decidíveis (*cf.* secção 2.5), também X é indecidível.

À semelhança do procedimento anterior, é possível esquematizar o estudo de linguagens semi-decidíveis: sabendo que a linguagem $\overline{\mathsf{H}}$ é semi-indecidível, suponha-se agora que $\overline{\mathsf{H}} \propto_h X$ é uma redução. Se X fosse semi-decidível, então $\mathsf{c'}_{\mathsf{H}} = \mathsf{c'}_X \circ h$ verificaria as condições na definição de linguagem semi-decidível, o que não pode acontecer (porque $\overline{\mathsf{H}}$ não é semi-decidível). Portanto, X não pode ser semi-decidível. No entanto, será necessária alguma cautela com a escolha do análogo da função auxiliar ψ do exemplo anterior. Vejamos, com um exemplo, como pode ser definida esta função.

Exemplo 4.3.11 $\mathsf{T} = \{x \mid \phi_x \text{ é total}\}$ é recursivamente enumerável? *Vamos definir uma redução* $\overline{\mathsf{H}} \propto_h \mathsf{T}$*, mas não podemos fazer simplesmente*

$$\psi(x,y) = \begin{cases} 1 & x \in \overline{\mathsf{H}} \\ \bot & x \in \mathsf{H} \end{cases}$$

porque o caso $x \in \overline{\mathsf{H}}$ *(equivalente a* $x \notin \mathsf{H}$*) não pode ser decidido por uma computação (finita). Porém, usando a função* resultado *do teorema*

4.3.2, esta inconveniência pode ser ultrapassada. Definimos

$$\psi(x,y) = \begin{cases} 1 & \mathsf{resultado}(x,x,y) = 0 \\ \perp & \mathsf{resultado}(x,x,y) > 0 \end{cases}.$$

Portanto, se $x \in \overline{\mathsf{H}}$*, para qualquer* y*, tem-se* $\mathsf{resultado}(x,x,y) = 0$ *e* $\psi(x,y) = 1$*. Mais exactamente, para cada* $x \in \overline{\mathsf{H}}$ *a função* $\psi(x,y) : y \mapsto 1$ *coincide com a função* $\mathsf{um} : z \mapsto 1$*. Por outro lado, se* $x \notin \overline{\mathsf{H}}$ *então* $x \in \mathsf{H}$ *portanto existe um* y_0 *tal que* $\mathsf{resultado}(x,x,y_0) > 0$ *e, para* $y \geq y_0$*,* $\psi(x,y) \uparrow$*. Por outro lado, como a função* ψ *é computável, existe um certo índice* i *tal que*

$$\psi(x,y) = \phi_i(\overline{x,y})$$

e, pelo teorema s-m-n,

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

Assim, $h : x \mapsto s(i,x)$ *é uma redução de* $\overline{\mathsf{H}}$ *para* T *pois*

- *se* $x \in \overline{\mathsf{H}}$ *então* $\phi_{h(x)} = \mathsf{um}$ *é uma função total e* $h(x) \in \mathsf{T}$ *e,*
- *se pelo contrário,* $x \notin \overline{\mathsf{H}}$ *então* $x \in \mathsf{H}$ *e para algum* y *tem-se* $\phi_{h(x)}(y) \uparrow$ *o que indica que esta função não é total e que* $h(x) \notin \mathsf{T}$*.*

Finalmente, como $\overline{\mathsf{H}}$ *não é r.e., também* T *não o é.*

De novo, por analogia com a resolução da decidibilidade de linguagens, podemos sistematizar o estudo das linguagens semi-decidíveis, com uma ressalva importante:

1. Se ω não está em X, reduziremos $\overline{\mathsf{H}} \propto_h X$:
   (a) seja f uma função (com índice) em X (portanto f é computável);
   (b) e g uma função computável (com índice) em $\overline{X}$, *cuidadosamente escolhida*;

(c) definindo

$$\psi(x,y) = \begin{cases} f(y) & \mathsf{resultado}(x,x,y) = 0 \\ g(y) & \mathsf{resultado}(x,x,y) \neq 0 \end{cases}$$

resulta que ψ também é computável;

(d) portanto, existe um certo índice i tal que $\psi(x,y) = \phi_i(\overline{x,y})$;

(e) pelo teorema s-m-n, tem-se

$$\psi(x,y) = \phi_i(\overline{x,y}) = \phi_{s(i,x)}(y)$$

(f) Seja $h : x \mapsto s(i,x)$;

(g) $\overline{\mathsf{H}} \propto_h X$ é uma redução de $\overline{\mathsf{H}}$ para X. É preciso provar que:

Se $x \in \overline{\mathsf{H}}$ então $h(x) \in X$: Se $x \in \overline{\mathsf{H}}$, $\mathsf{resultado}(x,x,y) = 0$ logo $\psi(x,y) = f(y)$ e $h(x) \in X$;

Se $h(x) \in X$ então $x \in \overline{\mathsf{H}}$: Para este sub-caso é preciso usar-se g. Por esta razão é que a função g deve ser cuidadosamente escolhida. A prova desta implicação depende das propriedades de g.

(h) Como $\overline{\mathsf{H}}$ não é r.e., conclui-se que X também não o é.

2. Se $\omega \in X$, reduziremos $\overline{\mathsf{H}} \propto_h X$ usando

$$\psi(x,y) = f(x) + \mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x}))$$

em que a função computável f é escolhida de forma tal que os seus índices *não* pertençam a X.

Neste esquema é necessário ter atenção à escolha da função g. No esquema para a decidibilidade basta ter em atenção se $\omega \in X$. Porém, para a semi-decidibilidade, o esquema indicado refere uma certa função g, *cuidadosamente escolhida*. O exercício resolvido 4.5.1 ilustra como pode ser feita essa escolha.

Temos então esboçado um método geral, baseado em reduções e na linguagem H, para determinar se uma dada linguagem é (semi-) decidível.

Porém, para o caso da semi-decidibilidade, este método requer engenho para a escolha da função g e, falhando esta escolha, não se pode garantir a redução.

Por outro lado, certas linguagens podem ser estudadas com abordagens mais simples. Uma possibilidade é quando temos uma generalização de uma linguagem já conhecida. Como as propriedades verificadas para o caso geral implicam as mesmas propriedades para o caso particular, a negação de uma propriedade no caso particular implica a negação dessa propriedade no geral.

Exemplo 4.3.12 *Sabemos que a linguagem* $\overline{\mathsf{H}} = \{x \mid \phi_x(x) \uparrow\}$ *não é decidível. Mas esta linguagem é um caso particular de* $K = \{\overline{x,y} \mid \phi_x(y) \uparrow\}$. *Portanto* K *também não é decidível. Neste exemplo, a propriedade é a* decidibilidade *de linguagens, o caso particular é* H *e o geral* K. *Como o caso particular não verifica a propriedade indicada, o caso geral também não a verifica. (**exercício**: explique este argumento em termos de adaptar um eventual decisor de* K *a um decisor de* $\overline{\mathsf{H}}$.*)*

4.4 O Teorema de Rice

O facto de termos esboçado um método geral (algorítmico) para a resolução de exercícios sobre a (semi-) decidibilidade de linguagens sugere que deve existir algum resultado teórico mais profundo sobre este assunto. O teorema que segue faz precisamente isso, mas apenas no caso particular da decidibilidade.

Teorema 4.4.1 (O teorema de Rice) *Seja* X *um conjunto de funções parciais recursivas. Então a linguagem* $\mathcal{I}_X = \{x \mid \phi_x \in X\}$ *é decidível se, e só se,* $\mathcal{I}_X = \emptyset$ *ou* $\mathcal{I}_X = \mathbb{N}$.

Demonstração. Tanto $\emptyset$ como $\mathbb{N}$ são linguagens recursivas. Supondo que $\mathcal{I}_X$ não é nem $\emptyset$ nem $\mathbb{N}$, o conjunto X terá de conter pelo menos uma função computável, digamos f, mas não todas as funções parciais

recursivas. Sem perda de generalidade (**exercício**: porquê?) supomos que $\omega \notin X$ e definimos

$$\psi(x,y) = f(y) + \mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x}))\,.$$

Como ψ é primitiva recursiva existe um certo índice, digamos i, tal que $\phi_i(\overline{x,y}) = \psi(x,y)$ e, pelo teorema s-m-n, sendo $j = s(i,x)$, obtemos $\phi_j(y) = f(y) + \mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x}))$. Agora, se $x \in H$ então $\phi_j = f \in X$ e se $x \notin H$ então $\phi_j = \omega \notin X$. Temos então uma redução de H para $\mathcal{I}_X$ pelo que esta última linguagem não é recursiva. □

É necessária alguma atenção nas aplicações do teorema de Rice. Notemos que o conjunto X do enunciado é um conjunto de *funções* e não de *programas*. Isto é, se $f \in X$ então $\mathcal{I}_X$ contém *todos* os índices da função f — e sabemos que a cada função computável correspondem infinitos índices.

Exemplo 4.4.2 *O conjunto*

$$\mathsf{Menores} = \{x \mid x \text{ é o menor índice da função } \phi_x\}$$

separa os índices de cada função computável em dois conjuntos disjuntos. Seja f uma função computável. Sabemos que existe uma infinidade de índices de f. Seja $\mathcal{I}_f$ o conjunto (infinito) de todos os índices de f. Como $\mathcal{I}_f \cap \mathsf{Menores}$ tem exactamente um elemento (o menor elemento de $\mathcal{I}_f$), resulta que Menores *separa $\mathcal{I}_f$ em dois conjuntos* não vazios*:*

$$\begin{aligned} \mathcal{I}_f &= (\mathcal{I}_f \cap \mathsf{Menores}) \cup (\mathcal{I}_f \setminus \mathsf{Menores}) \\ \mathcal{I}_f \cap \mathsf{Menores} &\neq \emptyset \\ \mathcal{I}_f \setminus \mathsf{Menores} &\neq \emptyset \end{aligned}$$

Portanto o conjunto Menores *está fora do âmbito do teorema de Rice.*

Já os conjuntos (dos índices) de todas as funções definidas numa infinidade de argumentos, ou dos pares de funções (f,g) tais que $g(x) = 2f(x)$, são contemplados pelo teorema de Rice (e, portanto, são conjuntos indecidíveis).

Para investigar a decidibilidade das linguagens ilustradas pelo exemplo do conjunto Menores é necessário recorrer a técnicas diferentes, algumas acessíveis, como ilustrado no exemplo 4.3.12, outras *ad hoc* ou tecnicamente demasiado complexas para serem referidas aqui.

Apesar de não contemplar todas as linguagens, o teorema de Rice é pertinente: para um conjunto significativo de linguagens, a classificação como decidível/indecidível é trivial. Existe também uma versão deste teorema para linguagens semi-decidíveis (o teorema de Rice-Shapiro) que, devido à sua complexidade, está fora do âmbito deste livro.

4.5 Exercícios

Exercícios Resolvidos

Exercício 4.5.1 *Mostre que a linguagem*

$$L = \{x \mid \exists y\, \phi_x(y) \downarrow \wedge \forall z\, \phi_x(z) \neq 2\phi_x(y)\}$$

não é r.e.

Resolução. Para resolvermos este exercício é boa ideia explorar alguns exemplos de funções (com índices) em L e outras em $\overline{L}$. Por exemplo, a função ω não está em L pois não existe y tal que $\omega(y) \downarrow$.

Por outro lado, a função

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ \perp & \text{se } n > 0 \end{cases}$$

já está em L pois $f(0) \downarrow 1$, $f(n) \uparrow$ em qualquer $n \neq 0$ e $2 \notin \mathsf{im}(f)$. *Para garantir que (o índice de) uma função f esteja em L é necessário assegurar que, se $f(y) \downarrow b$, então $2b \notin \mathsf{im}(f)$.*

Uma função (total computável) cujo índice não está em L é $\mathsf{exp2} : n \mapsto 2^n$. Esta função em particular, além de ser total, até tem outra propriedade crucial para este exercício: *para qualquer n, existe m tal que* $\mathsf{exp2}(m) = 2 \times \mathsf{exp2}(n)$.

Tentemos agora resolver este exercício... A linguagem L não é decidível (*exercício*). Vamos reduzir $\overline{\mathsf{H}} \propto_h L$. Como $\omega \notin L$ definimos

$$\psi(x,y) = \begin{cases} 1 & \text{se } \mathsf{resultado}(x,x,y) = 0 \\ \mathsf{exp2}(y) & \text{se } \mathsf{resultado}(x,x,y) > 0 \end{cases}$$

Mas ψ é computável logo existe um índice i tal que

$$\psi(x,y) = \phi_i(\overline{x,y}) \overset{(por\ s\text{-}m\text{-}n)}{=} \phi_{s(i,x)}(y).$$

Seja, então, $h : x \mapsto s(i,x)$ e vejamos que $\overline{\mathsf{H}} \propto_h L$:

- Se $x \in \overline{\mathsf{H}}$, então, para cada y, $\mathsf{resultado}(x,x,y) = 0$ logo $\psi(x,y) = 1$ donde $\phi_{h(x)} \equiv \mathsf{um}$. Portanto, $h(x) \in L$.

- Se $x \notin \overline{\mathsf{H}}$, então $x \in \mathsf{H}$ logo existe um y_0 tal que, se $y \geq y_0$,

  $$\mathsf{resultado}(x,x,y) > 0.$$

  Mas então, para $y \geq y_0, \psi(x,y) = \mathsf{exp2}(y)$ isto é, para $y \geq y_0, \phi_{h(x)}(y) = \mathsf{exp2}(y)$. Mas, então, $\phi_{h(x)}(y_0) \downarrow$ e $\phi_{h(x)}(y_0+1) = 2\phi_{h(x)}(y_0)$. Portanto, $h(x) \notin L$.

Concluímos que h é, de facto, uma redução de $\overline{\mathsf{H}}$ para L, portanto L não é recursivamente enumerável.

Exercício 4.5.2 *Sejam $A \subseteq \mathbb{N}$ uma linguagem e f uma função computável total. Mostre que*

$$x \in \mathcal{W}_x \text{ se, e só se } f(x) \notin A$$

implica que A não é semi-decidível.

Resolução. Com vista a um absurdo, supondo que A é semi-decidível, então a função

$$\mathsf{c'}_A : n \mapsto \begin{cases} 1 & \text{se } n \in A \\ \perp & \text{caso contrário} \end{cases}$$

é computável. Logo a função

$$\mathsf{c}'_A \circ f : x \mapsto \begin{cases} 1 & \text{se } f(x) \in A \\ \perp & \text{caso contrário} \end{cases}$$

é computável e coincide com

$$\mathsf{c}'_B : x \mapsto \begin{cases} 1 & \text{se } x \notin \mathcal{W}_x \\ \perp & \text{caso contrário} \end{cases}$$

em que $B = \{x \mid x \notin \mathcal{W}_x\}$ e conclui-se que B é semi-decídivel.

Vejamos que B não pode ser semi-decidível. Definindo $C = \overline{B} = \{x \mid x \in \mathcal{W}_x\}$:

1. C é semi-decidível pois $\mathsf{c}_C : x \mapsto \mu_{\overline{a,c}}[\mathsf{resultado}(x,a,t) = \mathsf{S}(x)]$;
2. C não é decidível. Uma redução $\mathsf{H} \propto_h C$ é definida por:
   (a) $\mathsf{cod}(\mathsf{id}) \in C$: $\mathsf{im}(\mathsf{id}) = \mathbb{N}$ logo $\mathsf{cod}(\mathsf{id}) \in \mathsf{im}(\mathsf{id})$ portanto $\mathsf{cod}(\mathsf{id}) \in \mathcal{W}_{\mathsf{cod}(\mathsf{id})}$;
   (b) $\mathsf{cod}(\omega) \notin C$: $\mathsf{im}(\omega) = \emptyset$ logo $\mathsf{cod}(\omega) \notin \mathsf{im}(\omega) = \mathcal{W}_{\mathsf{cod}(\omega)}$;
   (c) seja $\psi(x,y) = \mathsf{id}(y) + \mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x}))$;
   (d) como ψ é computável, pelo teorema s-m-n, tem-se, para um certo índice i,
   $$\psi(x,y) = \phi_i(\overline{x,y}) = \phi_{s(i,x)}(y)$$
   (e) Seja então $h : x \mapsto s(i,x)$;
   (f) se $x \in \mathsf{H}$ então $\phi_x(x) \downarrow$ logo $\mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x})) = 0$ portanto
   $$\forall y\ \phi_{h(x)}(y) = \mathsf{id}(y)$$
   donde $\phi_{h(x)} \equiv \mathsf{id}$ e, assim, $h(x) \in C$;
   (g) se, pelo contrário, $x \notin \mathsf{H}$ resulta que tem de ser $\phi_x(x) \uparrow$ logo $\mathsf{Z}(\phi_{\mathsf{u}}(\overline{x,x})) \uparrow$ portanto
   $$\forall y\ \phi_{h(x)}(y) = \omega(y)$$
   donde $\phi_{h(x)} \equiv \omega$ e, assim, $h(x) \notin C$;

(h) Portanto, $h : \mathsf{H} \to C$ é uma redução.

3. Como $h : \mathsf{H} \to C$ é uma redução, C é semi-decidível e indecidível, conclui-se que $\overline{C} = B$ não é semi-decidível;

Desta contradição resulta que A não pode ser semi-decidível.

Exercícios por Resolver

Exercício 4.5.3 *Prove que a diferença, o produto, o cociente, o resto e a potência de funções computáveis é computável.*

Exercício 4.5.4 *Explique rigorosamente os significados das expressões*

1. $\mathsf{soma}\,(3,2) = 5$;
2. $\overline{\mathsf{soma}}\,(39) = 5$;
3. $\overline{\mathsf{soma}}\,\left(\overline{3,2}\right) = 5$;
4. $\overline{\mathsf{soma}}\,(3,2) = 5$;
5. $\mathsf{soma}\,(\Pi_1\,(39)\,, \Pi_2\,(39)) = 5$;

Exercício 4.5.5 *Sejam A e B linguagens semi-decidíveis. Mostre que as linguagens $A \cap B$ e $A \cup B$ são semi-decidíveis mas que $\mathbb{N} \setminus A$ não tem de ser semi-decidível.*

Exercício 4.5.6 *Sejam f uma função total computável, A uma linguagem decidível e B uma linguagem semi-decidível. Mostre que a linguagens $f^{-1}\,(A)$ é decidível e que as linguagens $f\,(A)$, $f\,(B)$ e $f^{-1}\,(B)$ são semi-decidíveis. O que mais é possível dizer sobre estas três últimas linguagens, no caso de f ser bijectiva?*

Exercício 4.5.7 *Tente resolver o exercício resolvido 4.5.1 usando outras funções auxiliares em vez daquelas apresentadas (1 e $\mathsf{exp2}$). Tenha em especial atenção a prova da implicação $x \notin \overline{\mathsf{H}} \Rightarrow h\,(x) \notin L$. Por exemplo, substitua $\mathsf{exp2}$ por ω e certifique-se que não consegue fazer a prova.*

Exercício 4.5.8 *Mostre que* não *existe uma função computável total* $f(x,y)$ *que verifique a seguinte propriedade:* Se $\mathsf{dec}(x)(y)\downarrow$, essa computação tem, no máximo, $f(x,y)$ passos.

Exercício 4.5.9 *Sejam* $K_0 = \{x \mid \phi_x(x) = 0\}$ *e* $K_1 = \{x \mid \phi_x(x) = 1\}$. *Mostre que* K_0 *e* K_1 *são* recursivamente inseparáveis, *isto é, que:*

1. $K_0 \cap K_1 = \emptyset$;
2. Não *existe uma linguagem decidível* C *tal que* $K_0 \subset C$ *e* $K_1 \subset \overline{C}$.

Para isso

1. *mostre que existe uma linguagem decidível* Q *tal que*
   (a) $y \in \mathcal{E}_x$ *se, e só se,* $\exists z : \overline{x,y,z} \in Q$;
   (b) *Se* $y \in \mathcal{E}_x$ *e* $\overline{x,y,z} \in Q$ *então* $\phi_x(\Pi_1(z)) = y$
2. *deduza que existe uma função computável* $g(x,y)$ *tal que*
   (a) $g(x,y)\downarrow$ *se, e só se,* $y \in \mathcal{E}_x$;
   (b) *Se* $y \in \mathcal{E}_x$ *então* $g(x,y) \in \mathcal{W}_x$ *e* $\phi_x(g(x,y)) = y$, i.e. $g(x,y) \in \phi_x^{-1}(\{y\})$
3. *conclua que se* f *for computável injectiva (não necessariamente total ou sobrejectiva) então* f^{-1} *é computável.*

Exercício 4.5.10 *Seja* $g(x,y)$ *a função definida no exercício 4.5.9. Mostre que existe uma função computável total* h *tal que* $g(x,y) = \phi_{h(x)}(y)$ *e deduza que*

1. $\mathcal{W}_{h(x)} = \mathcal{E}_x$;
2. (a) $\mathcal{E}_{h(x)} \subseteq \mathcal{W}_x$;
   (b) *Se* $y \in \mathcal{E}_x$ *então* $\phi_x \circ \phi_{h(x)} : n \mapsto n$.
3. *Se* ϕ_x *for injectiva,* $\phi_{h(x)} \equiv \phi_x^{-1}$ *e* $\mathcal{E}_{h(x)} = \mathcal{W}_x$;

Exercício 4.5.11 *Mostre que, para cada $m \in \mathbb{N}$, existe uma função $(m+1)$-ária computável total $f^{(m)}$ tal que, para cada $n \in \mathbb{N}$,*

$$\phi_e^{(m+n)}(x, y) \equiv \phi^{(n)}_{f^{(m)}(e,x)}(y);$$

Exercício 4.5.12 *Mostre que existem funções computáveis totais h tais que*

1. *para cada $i \in \mathbb{N}$, se ϕ_i for a função característica da linguagem A, então $\phi_{h(i)}$ é a característica de*
   (a) *$\overline{A}$;*
   (b) *$\{x \mid \exists y \leq m : \overline{x, y} \in A\}$, para cada $m \in \mathbb{N}$;*
   (c) *$\{x \mid \forall y \leq m : \overline{x, y} \in A\}$, para cada $m \in \mathbb{N}$;*
2. *para cada $i \in \mathbb{N}$, $\mathcal{E}_{h(i)} = \mathcal{W}_i$;*
3. *para cada $i, j \in \mathbb{N}$, $\mathcal{E}_{h\left(\overline{i,j}\right)} = \mathcal{E}_i \cup \mathcal{E}_j$;*
4. *as linguagens $\mathsf{dom}(h)$ e $\mathsf{im}(h)$ são ambas indecidíveis.*

Exercício 4.5.13 *Mostre que existem funções computáveis totais s tais que*

1. *$\phi_{s(x,y)} \equiv \phi_x \phi_y$;*
2. *$\mathcal{W}_{s(x,y)} = \mathcal{W}_x \cup \mathcal{W}_y$*

Exercício 4.5.14 *Mostre que, para cada função computável h, a linguagem $\mathsf{dom}(h)$ é r.e.;*

Exercício 4.5.15 *Seja f uma função computável total binária. Para cada m, seja $g_m : n \mapsto f(m, n)$. Defina uma função computável total h tal que, para cada m, $h \not\equiv g_m$.*

Exercício 4.5.16 *Mostre que existem funções computáveis totais f tais que, para cada $n \in \mathbb{N}$,*

1. $f(n)$ *é um índice da função* $x \mapsto [\sqrt[n]{x}]$;
2. $\mathcal{W}_{f(n)}$ *é o conjunto das potências de* n;
3. *se* $n \geq 1$, $\mathcal{W}^{(n)}_{f(x)} = \{(y_1, \ldots, y_n) \mid y_1 + \cdots + y_n = x\}$;

Exercício 4.5.17 *Seja* $f_0, f_1, \ldots$ *uma enumeração de funções parciais* $\mathbb{N} \to \mathbb{N}$. *Defina (a partir desta enumeração) uma função* $g : \mathbb{N} \to \mathbb{N}$ *tal que* $\mathsf{dom}(g) \neq \mathsf{dom}(f_i)$ *para cada* i.

Exercício 4.5.18 *Seja* $f : \mathbb{N} \to \mathbb{N}$ *uma função parcial e* $m \in \mathbb{N}$. *Defina uma função* não *computável* g *tal que* $g(x) = f(x)$ *para* $x \leq m$.

Exercício 4.5.19 *Para cada* $a \in \mathbb{N}$, *seja* $\mathcal{W}^a_y = \{x \mid \phi_y(x) = a\}$. *Mostre que, para cada* a, *a linguagem* $\mathcal{W}^a_y$ *é semi-decidível. Será que a enumeração* $\mathcal{W}^a_0, \mathcal{W}^a_1, \mathcal{W}^a_2, \ldots$ *inclui todas as linguagens semi-decidíveis?*

Exercício 4.5.20 *Mostre que a linguagem* $\{x \mid \phi_x$ não é injectiva$\}$ *é semi-decidível.*

Exercício 4.5.21 *Mostre que existem funções computáveis totais* e *e* d *tais que* $\mathcal{W}_x = \mathcal{E}_{e(x)}$ *e* $\mathcal{E}_x = \mathcal{W}_{d(x)}$.

Exercício 4.5.22 *Seja* A *uma linguagem semi-decidível. Mostre que as linguagens* $\bigcup_{x \in A} \mathcal{W}_x$ *e* $\bigcap_{x \in A} \mathcal{E}_x$ *são ambas semi-decidíveis mas que* $\bigcap_{x \in A} \mathcal{W}_x$ *pode não o ser.*

Exercício 4.5.23 *Sejam* f *uma função unária computável,* $A \subseteq \mathsf{dom}(f)$ *e* $g = f|_A$ *(*g *é a* restrição *de* f *a* A*:* $g(x) = f(x)$ *se* $x \in A$ *e* $g(x)$ *está indefinido se* $x \notin A$*). Mostre que* g *é computável se, e só se* A *é semi-decidível.*

Exercício 4.5.24 *Classifique — sem usar o teorema de Rice — cada uma das seguintes linguagens como (**D**) decidível; (**RE**) indecidível, mas r.e.; (**N**) não r.e.*

1. $L1 = \{x \mid \phi_x$ é total$\}$;

2. $L2_{a,b} = \{x \mid \phi_x(a) = b\}$;
3. $L3_a = \{x \mid \phi_x \equiv \phi_a\}$;
4. $L4_a = \{x \mid \phi_a(x)\downarrow\}$ *(o problema da* aceitação*);*
5. $L5_a = \{x \mid \exists y : \phi_a(y) \downarrow x\}$ *(o problema da* escrita*);*
6. $L6 = \{x \mid \mathcal{E}_x = \emptyset\}$;
7. $L7 = \{x \mid \mathcal{W}_x$ é infinito$\}$;
8. $L8 = \{x \mid x \in \mathcal{E}_x\}$;
9. $L9 = \{\overline{x,y} \mid \mathcal{W}_x = \mathcal{W}_y\}$;
10. $L10 = \{x \mid \phi_x(x) = 0\}$;
11. $L11 = \{\overline{x,y} \mid \phi_x(y) = 0\}$;
12. $L12 = \{\overline{x,y} \mid x \in \mathcal{E}_y\}$;
13. $L13 = \{x \mid \phi_x$ é total e constante$\}$
14. $L14 = \left\{x \mid \mathcal{E}_x^{(n)} \neq \emptyset\right\}$;
15. $L15 = \{\overline{x,y} \mid \phi_x(y)$ é quadrado perfeito$\}$;
16. $L16 = \{x \mid \pi = 3,14159\ldots$ tem exactamente x setes consecutivos$\}$;
17. $L17 = \{x \mid x \notin \mathcal{W}_x\}$;
18. $L18 = \{\overline{x,y} \mid y \notin \mathcal{W}_x\}$;
19. $L19 = \{x \mid \phi_x$ não é total$\}$;

Exercício 4.5.25 *Suponha que $A \subseteq \mathbb{N}$ e seja f uma função computável total tal que $x \in \mathcal{W}_x$ se, e só se, $f(x) \notin A$. Mostre que A não é r.e.*

Use o teorema da recursão para determinar se existem índices i com as seguintes propriedades:

1. *O domínio de ϕ_i é o conjunto singular $\{i^2\}$;*
2. *O domínio de ϕ_i é $\mathbb{N} - \{i\}$;*
3. *O domínio de ϕ_i é H e também contém i;*

Entre capítulos

Termina aqui o estudo mais teórico dos limites da computação: aquilo que *em princípio* pode ser resolvido por um artefacto. De seguida consideraremos com mais atenção — impondo limites — as consequências daquele «*em princípio*», tendo em conta quer o tempo, quer a memória, que uma computação precisa para terminar.

Capítulo 5

Complexidade

A teoria da complexidade computacional procura compreender os limites *práticos* da computação. O conceito base na complexidade é o *número de passos elementares de uma computação*, considerando o tamanho dos argumentos. Estamos interessados em caracterizar os problemas cujo estudo computacional envolve um número razoavelmente pequeno de passos. E, no estudo computacional dos problemas, encontramos uma das mais importantes questões que a ciência do século XXI herdou do século XX, derivada da distinção entre *encontrar* uma solução de um problema (*e.g* qual é o quociente de 12 por 3?) e *verificar* um candidato a solução (*e.g.* se multiplicar 3 por 4, será que obtenho 12?).

Este capítulo está organizado da seguinte forma: depois de uma introdução informal iremos adaptar o modelo URM de computação para tratar as questões da complexidade computacional. Este passo é mais complicado do que poderia parecer à primeira vista porque as instruções atómicas dos programas URM manipulam um número arbitrário de bits — por exemplo, se no registo R_1 estiver o número 1023, a instrução $\mathtt{Z}(1)$ muda, num só passo, 10 bits. Como a complexidade é medida, em geral, em termos de operações que manipulam apenas um bit, torna-se necessário optar entre introduzir um outro modelo de computação, desperdiçando todo o conhecimento desenvolvido sobre o modelo URM ou então, como vamos fazer, estender o modelo URM de forma a manipular

bits individuais. Depois de resolver esta questão técnica, com o modelo «binário» de computação URM, definimos as classes de complexidade **P** e **NP**. Na segunda secção iremos estudar algumas propriedades elementares da complexidade. As reduções entre problemas serão abordadas nas duas secções finais: na terceira secção serão definidas as reduções polinomiais, a noção de problema «completo» e enunciado o **Teorema de Cook**, sobre o (importante) problema $\mathcal{SAT}$, de satisfação de fórmulas proposicionais. Na última secção exploramos reduções polinomiais dentro da Teoria dos grafos.

Introdução

A complexidade computacional assenta na contabilidade dos recursos necessários para uma computação. Essa contabilidade mede dois tipos principais de recursos: o *tempo* e o *espaço*. Mais precisamente,

- *o tempo de uma computação* é o número de estados intermédios entre o estado inicial e o estado terminal e
- *o espaço de uma computação* é o número máximo de bits usados pela memória do programa, considerado entre todos os estados intermédios.

Há uma observação elementar, mas importante, sobre o espaço e o tempo envolvidos numa computação: se limitarmos o tempo, também o espaço fica limitado — porque não é possível, em n estados intermédios, mudar mais do que n registos. Mas a recíproca não é verdadeira; um programa pode ter uma computação de tempo infinito sem alterar qualquer bit: $[\mathtt{J}\,(1,1,1)]$.

Assim, o tempo de uma computação, por também abranger o espaço, torna-se mais pertinente e interessante de analisar[1]. É o que faremos neste capítulo, concentrando-nos na medida do tempo das computações.

[1]Tecnologicamente, o espaço mede-se pela capacidade de armazenamento (seja na *RAM* ou em discos rígidos, *pen drives*, *etc.*) enquanto o (inverso do) tempo é medido pela velocidade do *CPU*.

O tempo, como medida de consumo de recursos computacionais, pode ser classificado em duas grandes categorias, que correspondem, grosso modo, às noções intuitivas de «acessível» e «excessivo». Dado um programa URM, F, queremos estimar o tempo necessário para terminar a computação $F(x)$, sabendo que o argumento x tem n bits. Isto é, queremos encontrar uma função $t : \mathbb{N} \to \mathbb{N}$ de modo que sejam suficientes $t(n)$ estados intermédios para terminar a computação $F(x)$, sabendo que x tem, no máximo, n bits. Esta função, t, determina a classificação do programa F: se t for um polinómio, diremos que F é um programa **polinomial**, *i.e.* «acessível». Caso contrário, se a nossa melhor estimativa t «crescer mais depressa que qualquer polinómio», diremos que F é **não-polinomial**, *i.e.* «excessivo». Esta associação de «polinomial» a «acessível» não é pacífica mas não dispomos, de momento, de instrumentos que nos permitam fazer uma crítica fundamentada. Podemos porém adiantar que, tecnicamente, um programa que efectue $1010^{1101^{1110}}$ passos elementares para terminar qualquer computação é polinomial. Porém, em termos práticos, seriam necessárias várias vezes a idade do universo para terminar uma computação deste programa, em qualquer processador que possa ser fisicamente construído.

O estudo da complexidade não se limita a classificar programas, ou os problemas que estes resolvem, em termos de «acessível» ou «excessivo». Um outro aspecto central na complexidade tem a ver com um conceito já estudado, na secção 4.3.4: a redução de um problema a outro. Aqui, preocupados com o custo das computações, consideraremos **reduções polinomiais**. No contexto da complexidade, uma aplicação das reduções fica ilustrada pelo seguinte cenário:

Dada a linguagem $\mathsf{M}_4 = \{x \mid x \text{ é múltiplo de } 4\}$, queremos definir um decisor (*i.e.* um programa que compute a função característica) de M_4. Neste exemplo, estamos interessados em explorar o caso particular em que já temos um decisor da linguagem $\mathsf{Pares} = \{x \mid x \text{ é par}\}$. Antes de definir, de raiz, um decisor para M_4, pergunta-mo-nos se *será possível «adaptar» o decisor de* Pares *para decidir* M_4*?* e, sendo possível, *que desvantagens, se algumas, podem resultar dessa adaptação?*

As adaptações de decisores de uma linguagem, B, a outra, A, e também os processos computacionais implicados nessas adaptações, são exemplos de *reduções*, computações especialmente interessantes para a teoria da complexidade. Sabendo que (1) a linguagem A tem um decisor «acessível» e que (2) adaptar um decisor de B para um decisor de A também é «acessível», podemos suspeitar que (3) a linguagem B tem um decisor «acessível». Esta transitividade, por reduções «acessíveis», fornece uma ferramenta importante para o estudo da complexidade de problemas ainda por analizar.

Há um outro aspecto central na complexidade computacional. Até aqui temo-nos centrado na questão de «resolver» um problema, isto é, de encontrar uma solução. Mas há uma outra abordagem importante sobre as soluções de problemas: *Verificar se um determinado candidato é, ou não, solução.* Por exemplo, para o «problema da divisão», «resolver» significa definir um algoritmo que, dados x e y, compute $z = x/y$. Já «verificar» significa definir um algoritmo que dados x, y e z responda «sim» ou «não» à questão «$z = x/y$?», sendo z o candidato a solução. Neste problema, em particular, para «verificar» basta saber multiplicar: $z = x/y \Leftrightarrow x = z \times y$. Mas para «resolver» é necessário percorrer atentamente os passos de um algoritmo não trivial até se chegar ao valor correcto de z.

A questão de relacionar «resolver» e «verificar», à primeira vista, parece simples:

- «verificar» nunca é muito mais («excessivamente») difícil do que «resolver» e até poderá ser muito mais simples;
- por outro lado (inspirados pelo exemplo da divisão, suspeitamos também que) devem existir problemas muito mais difíceis de «resolver» do que de «verificar».

Primeiro, observamos que podemos «transformar acessivelmente» uma tarefa de «verificar» numa outra, de «resolver»: querendo «verificar» se z é solução e sabendo dividir x por y, basta observar se o resultado dessa divisão é igual a z. Assunto arrumado. Mas o segundo

ponto levanta uma das mais importantes questões científicas em aberto. A dificuldade está em encontrar um problema (1) *com* um algoritmo «acessível» para «verificar» mas (2) *sem* um algoritmo «acessível» para «resolver».

Este problema é famoso pela igualdade

$$\mathbf{P} = \mathbf{NP}$$

que iremos explicar nas secções seguintes.

O que esta igualdade, sendo verdadeira, acrescenta ao que já sabemos é que os problemas com «verificação acessível» também têm «resolução acessível» (a afirmação recíproca, como vimos no parágrafo anterior, é simples). A nossa intuição, sustentada pela experiência da divisão, insiste que «resolver» deveria ser, em geral, muito mais difícil do que «verificar»; deveriam abundar exemplos que mostrem definitivamente essa diferença. Certo? Errado.

Demonstrar esta aparente diferença, intuitivamente evidente, tem-se mostrado um desafio intransponível, desde 1971, quando **Stephen Cook** (1938–presente) publicou o seu artigo «*The complexity of theorem proving procedures*».[2]

5.1 Programas URM Binários

O estudo da complexidade assenta na contagem do número mínimo de passos elementares numa computação. Para esse efeito o modelo URM, tal como o temos usado, é particularmente mal adaptado. Ilustremos com um exemplo, a soma:

$$\mathtt{Soma} = [\mathtt{J}\,(2,3,5)\,,\mathtt{S}\,(1)\,,\mathtt{S}\,(3)\,,\mathtt{J}\,(1,1,1)]\,.$$

O número de passos elementares para computar $n+m$ é $1+4m$. Mas existe uma forma muito mais eficaz de computar $n+m$: o algoritmo

[2] Actualmente, em 2009, o problema $\mathbf{P} = \mathbf{NP}$ é um dos seis problemas em aberto no *Millennium Prize Problems*. A resolução de cada um destes problemas tem um prémio de um milhão de dólares.

aprendido na escola primária. Para somar, por exemplo, $8375 + 1784$ não são necessárias as $1 + 4 \times 1784 = 7137$ operações elementares do programa `Soma`, mas apenas cinco somas de dois ou três dígitos. É uma diferença muito significativa, especialmente quando o nosso interesse é contabilizar as operações efectuadas.

Qual a origem desta enorme diferença no desempenho dos dois algoritmos para a soma? No modelo URM, os números naturais são representados numa **notação unária**. Para escrevermos a constante n no primeiro registo são necessários $n + 1$ passos elementares:

$$\mathtt{Z}\,(1)\,,\overbrace{\mathtt{S}\,(1)\,,\ldots,\mathtt{S}\,(1)}^{n \text{ vezes}}$$

Nesta forma o número inteiro n é representado por $n+1$ contas[3] «•»

$$n = \bullet\overbrace{\bullet\cdots\bullet}^{n \text{ contas}},$$

que é, essencialmente, a **notação romana** (ver mais a este respeito em A.3, p.254).

Porém, com a **notação posicional**, a escrita dos números fica muito compactada. De facto, fica *exponencialmente* compactada. Em vez de n contas para representar n (ou $n + 1$, se precisarmos do número «zero») bastam, em base 10, $\lceil \log_{10}(n) \rceil$ símbolos. Dito de outra forma, em notação unária, usando n contas podemos escrever, no máximo, n números mas na notação posicional, com n símbolos, passamos a poder escrever (em base dez), 10^n números! Por exemplo, para escrever o número «um milhão» bastam sete símbolos, «1000000», enquanto que com a notação unária seria necessário um milhão de «•» (ou um milhão de «I», ou mil «M», em notação romana). Como as instruções URM usam os números naturais como se estivessem escritos numa notação unária, perdemos a compactação exponencial proporcionada pela escrita posicional.

[3] Usamos $n+1$ contas para representar n porque o número 0 deve ser explicitamente representado. A convenção mais simples é precisamente a que adoptamos: $0 \mapsto \bullet$, $1 \mapsto \bullet\bullet$, *etc.*

Vamos estender o modelo URM de forma a incorporar a escrita posicional, em particular a **base binária**[4]. Continuaremos a supor que nos registos estão números naturais e acrescentamos novas instruções que, intuitivamente, correspondem a ler e escrever bits nos registos. Além da leitura e da escrita, também é necessária uma instrução que indique quantos bits descrevem o número num dado registo:

leitura de um bit escrever o r_j-ésimo bit de r_i no registo R_p ;

escrita de um bit substituir o r_q-ésimo bit de r_j pelo r_p- ésimo bit de r_i;

número de bits de um número escrever o número de bits de r_i no registo R_j;

Vamos precisar, ainda, de definir algumas funções auxiliares.

Definição 5.1.1 (Funções da representação binária) *Seja $\alpha \in \mathbb{N}$. O **tamanho** (binário) de $\alpha \in \mathbb{N}$ é o número mínimo de bits na representação binária de α, e representa-se por $|\alpha|$. Sendo*

$$\alpha = \sum_i \alpha_i 2^i$$

com $\alpha_j \in \{0,1\}$, então o tamanho binário de α representa-se por δ_L (ou $|\cdot|$) e é definido por

$$\delta_L(\alpha) = \begin{cases} 0 & \alpha = 0 \\ \max\{k \mid \alpha_k \neq 0\} & \alpha \neq 0 \end{cases} \tag{5.1}$$

*Também definimos as funções j**-ésimo bit** (δ_R) e **substituição** (δ_W) por*

$$\delta_R(\alpha, j) = \alpha_j; \tag{5.2}$$

$$\delta_W(\alpha, i, \beta, j) = \alpha_i 2^j + \sum_{k \neq j} \beta_k 2^k; \tag{5.3}$$

[4]Não há nenhuma vantagem exponencial em escolher, por exemplo, a base 10 em vez da base 2 para a escrita posicional dos números. A prática corrente na computação é escolher a base *mais simples*, isto é, 2.

com $\alpha = \sum_k \alpha_k 2^k$ *e* $\beta = \sum_k \beta_k 2^k$.

Formalmente, redefinimos **instrução**, **programa** e **computação** (da secção 2.2) de forma a incluir instruções sobre os bits dos registos:

Definição 5.1.2 (Programas e Computações BURM)
As ***instruções*** *são as instruções da definição 2.2.1 e, adicionalmente, para* $a, b, c, d \in \mathbb{N} \setminus \{0\}$*:*

$$\mathtt{R}\,(a,b,c)\,;\quad \mathtt{W}\,(a,b,c,d)\,;\quad \mathtt{L}\,(a,b)\,; \tag{5.4}$$

Um ***programa*** *é como na definição 2.2.1.*

O ***espaço de trabalho*** *do programa* P *é o maior índice de registo nas instruções de* P*, tal como na definição 2.2.1, mas agora considerando que as instruções também podem ser dos tipos* $\mathtt{R}, \mathtt{W}$ *e* $\mathtt{L}$*.*

Um ***estado instantâneo*** *para um programa* P *é como na definição 2.2.1.*

Uma ***computação*** *do programa* $P = [I_1, \ldots, I_m]$ *com argumentos* $x_1, \ldots, x_n$ *é como na definição 2.2.1 e, adicionalmente, para definir o* ***próximo estado****, quando a instrução* I_q *é*

$$\mathtt{R}\,(a,b,c) \;:\; \left\{ \begin{array}{ll} r'_c & = \delta_R\,(r_a, r_b) \\ q' & = q+1 \end{array} \right. ; \tag{5.5}$$

$$\mathtt{W}\,(a,b,c,d) \;:\; \left\{ \begin{array}{ll} r'_c & = \delta_W\,(r_a, r_b, r_c, r_d) \\ q' & = q+1 \end{array} \right. ; \tag{5.6}$$

$$\mathtt{L}\,(a,b) \;:\; \left\{ \begin{array}{ll} r'_b & = \delta_L\,(r_a) \\ q' & = q+1 \end{array} \right. ; \tag{5.7}$$

As definições de «computação que ***termina****» e «****resultado*** *de uma computação» podem ser usadas como enunciadas na definição 2.2.1. Também as notações* $P\,(x_1, \ldots, x_n) \downarrow y$ *e* $P\,(x_1, \ldots, x_n) \uparrow$ *se mantêm.*

Com as novas instruções de acesso aos bits dos registos podemos refazer alguns programas URM, tirando partido da notação posicional binária. Por exemplo, a soma pode ser computada pelo programa no exemplo

5.1.3. Embora bastante mais comprido que o programa `Soma`, é muito mais eficaz: para somar $n+m$ bastam apenas $3+13\lceil \mathsf{log}_2(\mathsf{max}(n,m))\rceil$ passos elementares. O gráfico da figura 5.1 ilustra a diferença dramática entre os crescimentos linear, do primeiro algoritmo e logarítmico, deste último.

Exemplo 5.1.3 (Soma binária)

Começamos este exemplo por explorar um caso particular da soma em binário: $(11010)_2 + (1100)_2 = (100110)_2$

$k.^{\text{o}}$ bit	7	6	5	4	3	2	1
transporte	0	1	1	0	0	0	0
x_k	0	0	1	1	0	1	0
y_k	0	0	0	1	1	0	0
$(x+y)_k$	0	1	0	0	1	1	0

Com este exemplo podemos concluir que, para o algoritmo da soma posicional usamos três parcelas para cada dígito (bit): x_k, y_k *e um* transporte.

Para definir um programa para computar a soma de bits numa BURM, consideremos o seguinte estado típico dos registos:

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	$\cdots$
x	y	$k.^{\text{o}}$ bit	transporte	x_k	y_k	$x+y$	$\lvert x\rvert$	$\lvert y\rvert$	0	

Podemos agora definir algoritmo posicional para a soma em binário, no modelo computacional BURM. As parcelas estão em R_1 *e* R_2*. O resultado é calculado somando os bits das parcelas. O registo* R_3 *é o índice dos bits a serem somados. Em certos casos é necessário fazer um «transporte» (na linguagem do ensino básico, «e vão ...»). O número de «ciclos» neste algoritmo depende de* $|x|$ *e de* $|y|$*, calculados nas duas*

primeiras instruções.

$$
\texttt{Somabin} = \left[\begin{array}{rl||rl}
\textit{iniciar}: 1 & \texttt{L}\,(1,8) & 18 & \texttt{J}\,(3,9,34) \\
2 & \texttt{L}\,(2,9) & 19 & \texttt{J}\,(1,1,3) \\
\textit{principal}: 3 & \texttt{S}\,(3) & \textit{sub}0: 20 & \texttt{W}\,(10,3,7,3) \\
4 & \texttt{R}\,(1,3,5) & 21 & \texttt{Z}\,(4) \\
5 & \texttt{R}\,(2,3,6) & 22 & \texttt{J}\,(1,1,17) \\
6 & \texttt{J}\,(4,10,12) & \textit{sub}1: 23 & \texttt{W}\,(1,8,7,3) \\
7 & \texttt{J}\,(5,10,10) & 24 & \texttt{Z}\,(4) \\
8 & \texttt{J}\,(6,10,26) & 25 & \texttt{J}\,(1,1,17) \\
9 & \texttt{J}\,(1,1,30) & \textit{sub}2: 26 & \texttt{W}\,(10,3,7,3) \\
10 & \texttt{J}\,(6,10,23) & 27 & \texttt{Z}\,(4) \\
11 & \texttt{J}\,(1,1,26) & 28 & \texttt{S}\,(4) \\
12 & \texttt{J}\,(5,10,15) & 29 & \texttt{J}\,(1,1,17) \\
13 & \texttt{J}\,(6,10,23) & \textit{sub}3: 30 & \texttt{W}\,(1,8,7,3) \\
14 & \texttt{J}\,(1,1,26) & 31 & \texttt{Z}\,(4) \\
15 & \texttt{J}\,(6,10,20) & 32 & \texttt{S}\,(4) \\
16 & \texttt{J}\,(1,1,23) & 33 & \texttt{J}\,(1,1,17) \\
17 & \texttt{J}\,(3,8,34) & \textit{terminar}: 34 & \texttt{T}\,(7,1)
\end{array}\right] \tag{5.8}
$$

Neste momento, equipados com instruções de acesso a bits, estamos quase em condições de definir as classes de complexidade.

5.1.1 As Classes P e NP

Um aspecto importante no estudo da complexidade é a relação entre o número de bits do argumento e o número de passos elementares da computação. A fim de fundamentar bem esta distinção, usaremos o **tamanho** (binário) de um número natural e definiremos a **ordem de crescimento de uma função**. De seguida, poderemos definir as principais classes de complexidade computacional, **P** e **NP**. A designação **P** abrevia *polynomial (time decisor)*. Já **NP** abrevia *non deterministic polynomial (time decisor)* — e não *non polynomial*. O *non deterministic*

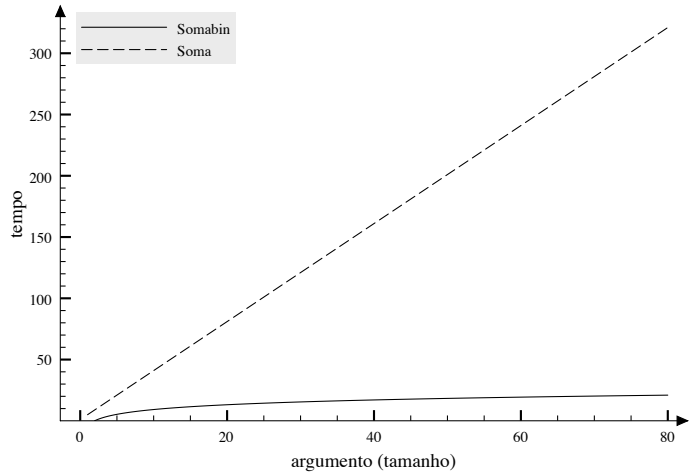


Figura 5.1: Gráfico que mostra os tempos de computação (eixo vertical) de `Soma` (linha tracejada) e `Somabin` (linha sólida) em função do tamanho (número de bits) dos argumentos (eixo horizontal). Note-se que a escala vertical está comprimida em relação à horizontal. Se expandirmos a escala vertical de forma que a linha de `Somabin` fique recta, a diferença para a `Soma` fica exponencialmente crescente.

refere-se ao facto de as computações de «verificação» de um «candidato» poderem recorrer a um «testemunho», dado por algum processo não necessariamente computacional (ou seja, não determinista).

Devemos notar que para a escrita binária de números, seguimos a mesma convenção da escrita decimal: *Os dígitos são escritos da direita para a esquerda.* Isto significa que $2 = (10)_2$ e não $2 = (01)_2$.

Exemplo 5.1.4 *Sobre o tamanho dos números naturais, observemos que, em base* 10*, diríamos que* $1, \ldots, 9$ *têm tamanho* 1*;* $10, 11, \ldots, 99$ *têm tamanho* 2*;* 100, 101, ..., 999 *têm tamanho* 3*;* etc. *Porém, em base* 2*,* 1 *tem tamanho* 1*, enquanto que* $2 = (10)_2, 3 = (11)_2$ *têm tamanho* 2

e $4 = (100)_2, 5 = (101)_2, 6 = (110)_2, 7 = (111)_2$ *têm tamanho* 3, etc.

$x = 0 = (0)_2$	$\lvert x\rvert = 0$
$x = 2 = (10)_2$	$\lvert x\rvert = 2$
$x = 7 = (111)_2$	$\lvert x\rvert = 3$
$x = 674 = (11100101)_2$	$\lvert x\rvert = 8$
$x = 83765 = (10101100111000101000)_2$	$\lvert x\rvert = 20$

Equipados com o tamanho de um número (*i.e.* de um argumento de um programa BURM), queremos relacionar esse tamanho com o *tempo (ou comprimento) da computação*. Com esta relação pretendemos dar um sentido formal à afirmação seguinte:

> *Se o argumento x tem n bits então, ao fim de t passos elementares, a computação $P(x)$ termina.*

Mas esta proposição é, ainda, uma aproximação do que se estuda na complexidade computacional. De facto, não é muito importante conhecer exactamente o limite t. Em vez disso, interessa-nos mais saber como é que t varia em função de n. Em particular, queremos saber se t «cresce demasiado depressa» quando aumentamos n.

Este interesse tem uma razão muito prática de ser. Supondo que dispomos de um certo programa, `Encriptar`, a que pretendemos dar muito uso, com argumentos «grandes» (por exemplo, para trocar mensagens com o nosso banco). Não basta garantir que esse programa implementa correctamente um dado algoritmo. Além de «correcto», o programa tem de ser «prático», *i.e.* não pode ser demasiado exigente em termos dos principais recursos, **tempo** e **memória** (espaço). Um programa que contrarie esta condição não passa de uma curiosidade teórica. Além disso, a questão de saber se são necessários exactamente x ou (digamos) $x + K$ passos (com K constante) para o programa terminar é pouco importante, e por duas razões: (1) quando x é um número «grande», a diferença entre x e $x + K$ torna-se pouco significativa (porque $\lim_{x\to+\infty} \frac{x}{x+K} = 1$) e (2) com um processador suficientemente rápido, os K passos extra requerem relativamente pouco esforço.

Imaginemos então que o programa `Encriptar` precisa aproximadamente de n^n passos elementares para processar mensagens com n bits e que vamos usar um computador que processa 10^9 operação elementares por segundo. Para mensagens de 9 bits basta um segundo; porém para processar uma mensagem de 128 bits (e é uma mensagem muito curta) seriam necessários

$$\frac{128^{128}}{10^9}\text{segundos} \approx 1.67 \times 10^{252}\text{anos}.$$

Compare-se este tempo com a idade estimada do universo, 1.37×10^{10} anos. Como é evidente, está fora de questão usar o programa `Encriptar` para negociar um empréstimo.

Por outro lado, se um outro programa, `EncriptarV2`, efectuar 10^{10} passos elementares, independentemente do número de bits do argumento, então para mandar uma mensagem de 128 bits, ou de 9 bits, ou de outro tamanho qualquer, seriam necessários 10 segundos.

Os tempos de computação destes dois exemplos são significativamente diferentes. Não apenas nos valores concretos que apresentam, mas, principalmente, na «forma como crescem» com o tamanho do argumento.

É a «forma de crescimento» que pretendemos definir formalmente. Para tal usamos a **ordem** de uma função, que agrupa funções com crescimentos semelhantes. Os conceitos relacionados com a ordem de uma função (principalmente as equações A.8 e A.9) estão definidos em A.3, p.260.

Para relacionar o tempo de computação com o tamanho dos argumentos, considere-se, por exemplo, o programa da soma unária

$$\texttt{Soma} = [\texttt{J}\,(3,2,5)\,,\texttt{S}\,(1)\,,\texttt{S}\,(3)\,,\texttt{J}\,(1,1,1)]$$

em que, com dois argumentos x e y é necessário efectuar $1+4\times y$ passos elementares. Agora, se $x, y \leq N$ então são efectuados, no máximo, $1+4\times N \in \mathcal{O}\,(N)$ passos elementares. Se N tiver n bits é, no máximo,

$N = 2^n$. Podemos então garantir que *se* $\max\{|x|, |y|\} = n$ *então a computação* $\texttt{Soma}(x, y)$ *termina em* $\mathcal{O}(2^n)$ *passos elementares*, ou seja, que

$$\texttt{Soma} \in \mathcal{O}(2^n).$$

Este é o tipo de relação que pretendemos estabelecer entre o tamanho (em bits) dos argumentos e o comprimento (em passos elementares) da computação. Deste exemplo em particular, apenas podemos concluir que conhecemos um algoritmo exponencial para efectuar a soma de dois números naturais, o que não é especialmente significativo.

Além da contabilidade do número de bits dos argumentos e do número de passos da computação, ainda há uma distinção que nos interessa. Recordemos o problema «Divisível por quatro?» e consideremos as duas situações seguintes:

1. Queremos saber se 84 é divisível por quatro: *dividimos* 84 *por quatro, verificamos que deixa resto* 0 *e respondemos «SIM»*;

2. Queremos saber se 88 é divisível por quatro e temos uma ajuda: «considere o número 22». *Neste caso calculamos* 4×22*, observamos que é* 88 *e respondemos «SIM»;*

Estes dois cenários ilustram a diferença entre «decidir» se 84 é divisível por quatro e «verificar» se 88 é divisível por quatro. Com estes casos em mente, definimos **decisor** e **verificador**.

Definição 5.1.5 (Decisor e verificador) *Seja* $A \subseteq \mathbb{N}$ *uma linguagem.*

1. *O* ***decisor*** *de* A *é a função característica de* A, c_A*:*

$$x \in A \;\Leftrightarrow\; \mathsf{c}_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases} \tag{5.9}$$

2. *Um* ***verificador*** *de* A *é uma função* $v_A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *tal que*

$$x \in A \;\Leftrightarrow\; \exists y \in \mathbb{N} : v_A(x; y) = 1 \tag{5.10}$$

ou, o que é equivalente,

$$x \notin A \;\Leftrightarrow\; \forall y \in \mathbb{N}, v_A(x;y) \neq 1 \tag{5.11}$$

*Se $x \in A$, e $v_A(x;y) = 1$ dizemos que y é um **testemunho** ou **certificado** de x.*

É importante observarmos que esta definição refere «o» decisor e «um» verificador, porque para cada linguagem existem inúmeros verificadores mas apenas um decisor. Podemos (facilmente) definir vários verificadores a partir desse decisor:

$$\begin{aligned} v_A^{(1)}(x;y) &= \mathsf{c}_A(x) \text{ (qualquer } y \text{ é testemunho de } x) \\ v_A^{(2)}(x;y) &= \mathsf{c}_A(x) \times y \text{ (1 é testemunho de } x) \end{aligned}$$

No primeiro caso, se $x \in A$, qualquer $y \in \mathbb{N}$ é testemunho enquanto que usando $v_A^{(2)}$, cada $x \in A$ admite um único testemunho, $y = 1$. Por outro lado, por exemplo, se definirmos

$$u_A(x;y) = \mathsf{c}_A(x) + y$$

não obtemos um verificador (**exercício:** porquê?).

Se a característica c_A, ou algum verificador v_A, for computável, podemos garantir que existe um programa BURM D que computa essa função e considerar «a forma de crescimento» da função $\mathsf{tempo}(\mathsf{cod}(D), \boldsymbol{x})$, que indica o número de passos elementares na computação $D(\boldsymbol{x})$ (comparar com a definição da função $\mathsf{tempo}(c, m)$, na secção 3.45, p.119)

Definição 5.1.6 (Tempo de computação) *Sejam D um programa BURM e $\boldsymbol{x} = x_1, \ldots, x_k$ um vector de naturais. Então o **tempo da computação** $D(\boldsymbol{x})$,*

$$\mathsf{t}_D(\boldsymbol{x}) \;=\; \mathsf{tempo}(\mathsf{cod}(D), 2^{x_1}3^{x_2}\cdots p_k^{x_k}) \tag{5.12}$$

é o número de passos elementares necessários para a computação $D(\boldsymbol{x})$ terminar.

Estamos agora em condições de definir as classes **P** e **NP**.

Definição 5.1.7 (A classe P) *Para as computações deterministas (de decisão),*

1. *Seja D um programa BURM, $\boldsymbol{x} = x_1, \ldots, x_k$ e definindo $|\boldsymbol{x}| = \mathsf{max}\{|x_1|, \ldots, |x_k|\}$. Então*

$$\mathsf{dt}_D(n) \quad = \quad \mathsf{max}\{\mathsf{t}_D(\boldsymbol{x}) \mid |\boldsymbol{x}| \leq n\} \tag{5.13}$$

   é o menor número de passos em que qualquer *computação de D com argumentos de tamanho máximo n termina*[5]*;*

2. *Seja A uma linguagem, D um programa BURM que compute o decisor c_A e $f : \mathbb{N} \to \mathbb{R}$. Se*

$$\mathsf{dt}_D \in \mathcal{O}(f)$$

   *diz-se que A é **decidível** em tempo f;*

3. *O conjunto das linguagens decidíveis em tempo polinomial (quando f é um polinómio) é **P**. Isto significa que*

$$\begin{array}{c} A \in \mathbf{P} \\ \Leftrightarrow \\ \exists D, k : (x \in A \Leftrightarrow D(x) \downarrow 1) \wedge \left(\mathsf{dt}_D \in \mathcal{O}\left(n^k\right)\right) \end{array} \tag{5.14}$$

Da afirmação «$A \in \mathbf{P}$» podemos deduzir que

1. a linguagem A é decidível: existem programas BURM que computam a função característica de A;

[5]Por exemplo, se $\mathsf{dt}_D(n) = 10$ todas as computações $D(\boldsymbol{x})$, com $|\boldsymbol{x}| \leq n$, terminam em 10 passos. Também terminam em $11, 12, \ldots$ passos, mas 10 é o *menor* número suficientemente grande para terminar todas essas computações. Pode acontecer que, para um certo $\boldsymbol{x}_0$ em particular cheguem 3 passos, para outro 8 serão suficientes, mas também sabemos que, entre todos os $\boldsymbol{x}$, com $|\boldsymbol{x}| \leq n$, existe um que precisa exactamente dos 10 passos para terminar.

2. além disso, entre todos esses programas, pelo menos um deles, digamos D, é «polinomial» *i.e.* existe um número natural $k \in \mathbb{N}$ tal que
$$\mathtt{t}_D(\boldsymbol{x}) < |\boldsymbol{x}|^k;$$
3. o limite anterior, no tempo de computação, é válido para qualquer argumento $\boldsymbol{x}$, independentemente deste ser, ou não, elemento da linguagem A.

É importante sublinhar que o expoente $k \in \mathbb{N}$ depende *apenas* da linguagem. Isto é, quando dizemos «*existe um* $k \in \mathbb{N}$ *tal que* $\mathtt{t}_D(\boldsymbol{x}) < |\boldsymbol{x}|^k$», este k é válido para qualquer $\boldsymbol{x}$. Portanto k *caracteriza a linguagem* e não os seus elementos individuais. Podemos assim dizer, por exemplo, que uma certa linguagem é «linear» ($k = 1$), «quadrática» ($k = 2$), *etc.*, independentemente do tamanho dos seus elementos. A classe **P** é a união das linguagens caracterizadas por $k = 1$, $k = 2$, *etc.*

Definição 5.1.8 (A classe NP) *Para as computações não deterministas (de verificação; com recurso a um testemunho),*

1. *Seja D um programa BURM e $\boldsymbol{x} = x_1, \ldots, x_k$. Então*
$$\mathsf{vt}_D(n) = \mathsf{max}\{\mathtt{t}_D(\boldsymbol{x}, y) \mid |\boldsymbol{x}| \leq n \wedge \exists y : D(\boldsymbol{x}, y) \downarrow 1\} \quad (5.15)$$
é o menor número de passos em que qualquer *verificação computada por D com argumentos de tamanho máximo n termina;*

2. *Seja A uma linguagem, v_A um verificador de A, V um programa BURM que compute v_A e $f : \mathbb{N} \to \mathbb{R}$. Se*
$$\mathsf{vt}_V \in \mathcal{O}(f)$$
diz-se que A é ***verificável*** *em tempo f.*

3. *O conjunto das linguagens verificáveis em tempo polinomial é* **NP**. *Isto significa que*
$$\begin{gathered} A \in \mathbf{NP} \\ \Leftrightarrow \\ \exists V, k : (\boldsymbol{x} \in A \Leftrightarrow \exists y : V(\boldsymbol{x}, y) \downarrow 1) \wedge \left(\mathsf{vt}_V \in \mathcal{O}\left(n^k\right)\right) \end{gathered} \quad (5.16)$$

Sabendo que «$A \in \mathbf{NP}$» podemos dizer que

1. a linguagem A é verificável: alguns verificadores de A são computáveis;

2. pelo menos um desses verificadores, digamos v_A, é «acessível»: existe um programa BURM V que computa v_A e, além disso, também existe um número natural $k \in \mathbb{N}$ tal que:

   (a) se $\boldsymbol{x} \in A$ então existe um testemunho $y \in \mathbb{N}$ tal que

   $$\mathsf{t}_V(\boldsymbol{x}; y) < |\boldsymbol{x}|^k;$$

   (b) neste caso deve-se notar que o tempo da computação *não* depende do tamanho do testemunho y, mas *apenas* do tamanho do argumento $\boldsymbol{x}$;

   (c) se $\boldsymbol{x} \notin A$ pode-se garantir que *não existe* qualquer testemunho y de modo que $V(\boldsymbol{x}; y) \downarrow 1$.

5.2 Propriedades Elementares

Dadas as definições de **P** e **NP**, veremos agora algumas propriedades destas classes.

Teorema 5.2.1

$$\mathbf{P} \subseteq \mathbf{NP} \tag{5.17}$$

Demonstração. Uma forma de demonstrar este teorema é mostrar que o decisor de cada linguagem é, também, um verificador dessa linguagem, se ignorar o testemunho. Vejamos como isso se faz.

Seja A uma linguagem em **P**. Então existe um programa BURM D e um inteiro k tais que

$$x \in A \;\Leftrightarrow\; D(x) \downarrow 1 \wedge \mathsf{dt}_D \in \mathcal{O}\left(n^k\right).$$

Para mostrar que $A \in \mathbf{NP}$, precisamos de encontrar um programa BURM V e um inteiro r que verifique a condição

$$x \in A \;\Leftrightarrow\; \exists y : V(x;y) \downarrow 1 \wedge \mathsf{vt}_V \in \mathcal{O}(n^r).$$

Podemos confirmar a condição $\exists y : V(x;y) \downarrow 1$ construindo o programa verificador V a partir do decisor D; basta acrescentar a instrução $\mathtt{Z}(2)$ ao início de D:

$$V = [\mathtt{Z}(2), D[1 \to 1]]$$

A computação $V(x;y)$ é, essencialmente, a mesma que $D(x,0)$ e que $D(x)$ (confimar, na definição 5.1.2, dos programas BURM, como estão definidas as computações com argumentos). Da forma como V e D estão definidos,

$$\begin{array}{rcl} x \in A & \Leftrightarrow & D(x) \downarrow 1 \\ & \Leftrightarrow & V(x,0) \downarrow 1 \\ & \Leftrightarrow & \exists y : V(x;y) \downarrow 1 \\ & \therefore & \\ x \in A & \Leftrightarrow & \exists y : V(x;y) \downarrow 1 \end{array}$$

Resta-nos atender à condição

$$\mathsf{vt}_V \in \mathcal{O}(n^r).$$

Desta vez fazemos $r = k$ e, de novo, usamos as propriedades de D:

$$\begin{array}{rcl} \mathsf{dt}_D & \in & \mathcal{O}\left(n^k\right) \\ & \Leftrightarrow & \\ \forall n, \mathsf{max}\{\mathsf{t}_D(x) \mid |x| \leq n\} & \leq & \alpha n^k \\ & \Leftrightarrow & \\ \forall n, \mathsf{max}\{\mathsf{t}_V(x,0) \mid |x| \leq n\} & \leq & \alpha n^k \\ & \Leftrightarrow & \\ \forall n, \mathsf{max}\{\mathsf{t}_V(x,y) \mid |x| \leq n \wedge \exists y : V(x,y) \downarrow 1\} & \leq & \alpha n^k \\ & \Leftrightarrow & \\ \mathsf{vt}_V & \in & \mathcal{O}\left(n^k\right) \end{array}$$

Portanto acabámos de mostrar como podemos transformar um decisor da linguagem A num verificador da mesma linguagem, sem aumentar significativamente o comprimento da computação. □

Se uma linguagem A está em **NP** então existe um certo verificador v_A e, para cada $a \in A$, podemos escolher um testemunho τ_a: $v_A(a;\tau_a) = 1$. Também existe um programa BURM V que computa v_A, e um inteiro k tal que $\mathsf{vt}_V \in \mathcal{O}\left(n^k\right)$. Ficou por saber *como pode ser encontrado o testemunho* τ_a... O próximo teorema diz-nos que podemos usar uma pesquisa de força bruta para o encontrar. A desvantagem deste método é que podemos despender demasiado tempo nessa pesquisa.

A ideia para demonstrar o teorema é simples: se o argumento x tem n bits e a computação $V(x,\tau_x)$ tem, no máximo, n^k passos elementares, então, do testemunho τ_x terão sido «usados», no máximo, $N = n^k$ bits. Quantos números é que existem com $N = n^k$ bits? Exactamente 2^N. Portanto o testemunho é um desses 2^N candidatos e «basta» testar cada um para ver qual funciona bem como testemunho.

Teorema 5.2.2 *Se $A \in$ **NP** então, para algum inteiro k, A é decidível em tempo $\mathcal{O}\left(2^{n^k}\right)$.*

Demonstração. Como A está em **NP** então

- existe um verificador v_A de A:

$$x \in A \Leftrightarrow \exists \tau_x : v_A(x,\tau_x) = 1;$$

- existe um programa BURM V, que computa v_A;
- existe um inteiro k tal que, para cada n,

$$\mathsf{vt}_V(n) \in \mathcal{O}\left(n^k\right).$$

Antes de usarmos o esquema de demonstração esboçado no parágrafo anterior precisamos considerar que, se $x \notin A$, as computações $V(x;i)$

podem não terminar... Portanto não podemos usar o programa do verificador como módulo do algoritmo de pesquisa. Em vez disso, simulamos a computação de V até um certo número de passos, no seguinte pseudo-código:

```
def Decidir_A(x) :
    n ← |x|
    m ← n^k
    y ← 2^m
    para cada i ∈ {0, ..., y} :
        se V(x; i) ↓ 1 em m passos :
            retorna 1
    retorna 0
```

É crucial notar que a computação $V(x;i)$ não é feita pelo programa V mas simulada passo a passo, usando, por exemplo, as funções computáveis da definição 3.3.2 (comput, config, *etc.*). Mais precisamente, o teste «$V(x;i) \downarrow 1$ em m passos» é computado por

$$\mathsf{comput}\left(\mathsf{config}\left(\mathsf{cod}\left(V\right), 1, \overline{x,i}\right), m\right) = 2$$

porque $\mathsf{comput}\left(s,m\right) = 0$ significa que a computação iniciada no estado s não terminou em m passos enquanto que se $\mathsf{comput}\left(s,m\right) = y+1 > 0$ então a computação iniciada em s termina em m passos e no primeiro registo está y.

Este pseudo-código ilustra um algoritmo que, com argumento x, devolve 1 se $x \in A$ e devolve 0 se $x \notin A$. Por outras palavras, este algoritmo computa a função característica de A. No entanto, para decidir se $x \in A$ (em particular quando $x \notin A$), podem ser necessários $2^{|x|^k}$ passos elementares. □

5.3 Transformações Polinomiais

O estudo da complexidade computacional assenta na classificação dos recursos suficientes para resolver certos problemas. Os domínios desses

problemas não têm de ser apenas sub-conjuntos dos números naturais. Podemos trabalhar em domínios mais gerais, desde que consigamos, de alguma forma, estabelecer uma correspondência entre os elementos desses domínios e os números naturais. Uma vez feita essa correspondência podemos usar os programas BURM para resolver problemas sobre grafos, fórmulas, álgebra, *etc.*

Porém, não temos ainda uma definição precisa do que é um «problema». Intuitivamente, vamos definir «problema» como um conjunto de «instâncias», separadas em dois subconjuntos disjuntos: as «instâncias positivas» e as «instâncias negativas»:

Definição 5.3.1 (Problema) *Um* ***problema*** X *é um conjunto*

$$\mathcal{I}_X = \mathcal{P}_X \cup \mathcal{N}_X$$

em que $\mathcal{P}_X \cap \mathcal{N}_X = \emptyset$*. Aos elementos de* $\mathcal{I}_X$ *chamamos* ***instâncias*** *de* X*; Aos elementos de* $\mathcal{P}_X$ ***instâncias positivas*** *e aos de* $\mathcal{N}_X$ ***instâncias****.*

Não impomos qualquer restrição sobre o conjunto das instâncias. Desta forma a definição de «problema» permite um leque muito abragente de objectos matemáticos, por exemplo, números naturais, espaços vectoriais, grafos e redes, *etc.*

Exemplo 5.3.2 *O problema «*Divisível por quatro?*» define-se por:*

Instâncias *Qualquer número natural* $n \in \mathbb{N}$*;*

Instâncias positivas *As instâncias* n *tais que o resto da divisão de* n *por* 4 *é* 0*;*

Instâncias negativas *As instâncias* n *tais que o resto de divisão de* n *por* 4 *é diferente de* 0*;*

Podemos pensar num programa BURM que, dado um natural qualquer x, indique se x é uma instância positiva ou se é negativa. Por exemplo, a função unária computada pelo programa BURM

$$\texttt{CDiv4} = \left[\texttt{Quatro}\left[\rightarrow 2\right], \texttt{Mod}\left[1, 2 \rightarrow 1\right], \overline{\texttt{SG}}\left[1 \rightarrow 1\right]\right]$$

é um decisor da linguagem formada pelas instâncias positivas deste problema. Neste exemplo pudemos definir directamente um programa para decidir se uma instância é positiva ou negativa porque estas são números naturais. Mas se quisermos resolver problemas sobre grafos, fórmulas, *etc.* não conseguimos representar «directamente» tais instâncias nos registos das máquinas BURM.

Assim, para tratar computacionalmente os problemas com a generalidade necessária, precisamos de «tradutores» que convertam instâncias em números naturais, para serem processadas por programas BURM.

Definição 5.3.3 (Esquema de código) *Um* ***esquema de código*** *de um problema X é uma função*

$$f : \mathcal{I}_X \to \mathbb{N}.$$

Dado um problema X e um esquema de código f, os números naturais ficam separados em três sub-conjuntos:

1. $\mathbb{N} \setminus f(\mathcal{I}_X)$*, números que não codificam instâncias;*
2. $f(\mathcal{N}_X)$*, números que codificam instâncias negativas;*
3. $f(\mathcal{P}_X)$*, números que codificam instâncias positivas;*

Em geral, um dado problema X pode admitir diferentes esquemas de código. Portanto, uma instância α pode ser representada por um número natural a, de acordo com um certo esquema f enquanto que, usando um esquema diferente, f', ser representado por $a' \neq a$.

Por enquanto basta-nos considerar que um esquema de código transforma as instâncias de um problema em números naturais, aptos a serem processados por programas BURM. Para o estudo da complexidade interessa-nos identificar as instâncias positivas, ou melhor, uma vez fixado um dado esquema de código f, identificar os números que correspondem às instâncias positivas. Para sublinhar não só o interesse no conjunto das instâncias positivas, mas também para explicitar o esquema de código usado temos a definição seguinte:

Definição 5.3.4 (Linguagem de um problema) *Seja X um problema e f um esquema de código. Então a imagem, por f, das instâncias positivas de X,*

$$\mathsf{L}\left[X; f\right] = f\left(\mathcal{P}_X\right) \tag{5.18}$$

*é a **linguagem de** X **(pelo esquema** f**).***

A principal característica do esquemas de código é que, ao transformarem instâncias de um problema num domínio arbitrário em números naturais, passamos a puder usar programas BURM para responder computacionalmente a questões sobre domínios diferentes do conjunto $\mathbb{N}$. Dito de outra forma, através dos esquemas de código, basta-nos *concentrar em linguagens — e programas — sobre os números naturais* e ignorar os domínios restantes. Fizemos algo semelhante quando codificámos os números inteiros nos naturais ($\alpha_{\mathbb{Z}}$, na secção 2.2, p.34)

5.3.1 Linguagens NP-completas

Um aspecto importante da complexidade computacional é o estudo das relações que diferentes problemas têm entre si. Mais precisamente, estamos interessados em saber se podemos «resolver» um dado problema indirectamente, através da resolução de outro problema.

Esta abordagem não é propriamente novidade: foi definida na secção 4.3.4 para o estudo da decidibilidade. Aí definimos «**redução** da linguagem A para B» como uma função computável total $f : \mathbb{N} \to \mathbb{N}$ tal que

$$x \in A \Leftrightarrow f\left(x\right) \in B$$

e escrevemos $A \propto_f B$. Agora estamos especialmente interessados em **reduções polinomiais**: a função f, além de computável total também tem de ser polinomial, *i.e.* tem de existir um programa F que compute f e $\mathsf{dt}_F \in \mathcal{O}\left(n^k\right)$.

Como veremos, esta resolução indirecta não só é possível como, além disso, podemos identificar um problema em particular que (num certo sentido técnico), resolve *todos* os problemas da classe **NP**.

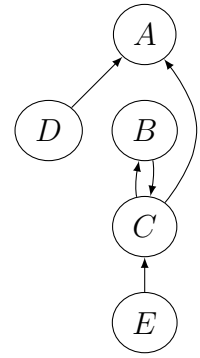


Figura 5.2: Um exemplo de hierarquia de linguagens. As setas indicam a existência de uma redução polinomial. Nesta figura a linguagem A é completa: um algoritmo que decida (resp. verifique) A pode ser adaptado para decidir (resp. verificar) as restantes linguagens.

Definição 5.3.5 (Linguagem NP completa) *Uma linguagem* $C \subseteq \mathbb{N}$ *é* **NP-*completa*** *se*

1. $C \in \mathbf{NP}$;
2. *para qualquer* $L \in \mathbf{NP}$ *existe uma redução polinomial* $L \propto_f C$.

Uma linguagem **NP**-completa proporciona um grande contributo para o estudo da complexidade pois permite que o estudo dos algoritmos para decidir/verificar as linguagens de **NP** fique bem organizado: dado que uma certa linguagem C é **NP**-completa, e dada uma qualquer linguagem $L \in \mathbf{NP}$, sabemos que existe uma redução polinomial $L \propto_f C$; Para verificar se $x \in L$ basta verificar se $f(x) \in C$. Como a redução $x \mapsto f(x)$ é polinomial e $C \in \mathbf{NP}$, também a verificação $f(x) \in C$ é computada em tempo polinomial. Portanto temos um processo indirecto (e polinomial) de verificar se $x \in L$.

A noção de linguagem **NP**-completa permite uma generalização directa, substituindo a classe **NP** por uma outra qualquer classe de linguagens. Por ser interessante, apresentamos aqui a definição:

Definição 5.3.6 *Seja $\mathcal{A}$ uma classe de linguagens e $C \subseteq \mathbb{N}$ uma linguagem. Dizemos que C é $\mathcal{A}$-completa se*

1. *$C \in \mathcal{A}$;*
2. *para qualquer linguagem $L \in \mathcal{A}$ existe uma redução polinomial $L \propto_f C$.*

Não basta enunciar a definição de linguagem **NP**-completa para garantir que existe, de facto, uma tal linguagem. Este é o problema que devemos abordar de seguida: *mostrar que existe (pelo menos) uma linguagem* **NP***-completa.* É uma grande tarefa, que não fica completa em algumas linhas, e não tentaremos explorar os detalhes técnicos[6]. Ficaremos pelo enunciado e por um esboço da demonstração.

Definição 5.3.7 (A linguagem $\mathcal{SAT}$) *A linguagem $\mathcal{SAT}$ é o conjunto das fórmulas conjuntivas satisfazíveis (rever A.2.3, p.242). Uma fórmula boolena, a, é* ***conjuntiva*** *se estiver na* forma normal conjuntiva*:*

$$a = (\lambda_{11} \vee \cdots \vee \lambda_{1k_1}) \wedge \cdots \wedge (\lambda_{n1} \vee \cdots \vee \lambda_{nk_n})$$

onde os λ_{ij} são ***literais*** *(positivas ou negativas) e é* ***satisfazível*** *se existir uma valoração v tal que $v \models a$, isto é que satisfaça a.*

Por exemplo, a fórmula $\varphi = a \vee \neg b$ é conjuntiva e é satisfazível porque a valoração $v = \begin{pmatrix} a & b \\ 1 & 0 \end{pmatrix}$ satisfaz φ:

$$\begin{aligned} \hat{v}(\varphi) = \hat{v}(a \vee \neg b) &= \mathsf{max}\{v(a), \hat{v}(\neg b)\} \\ &= \mathsf{max}\{1, 1 - v(b)\} \\ &= 1 \end{aligned}$$

Portanto $a \vee \neg b \in \mathcal{SAT}$. Já $a \wedge \neg a$ embora seja conjuntiva, não é satisfazível: seja u uma valoração e consideremos as duas possibilidades: $u_1(a) =$

[6] completamente cobertos, por exemplo, no livro clássico *Computational Complexity* de Christos H. Papadimitriou.

1 ou $u_2(a) = 0$; no primeiro caso fica $\hat{u}_1(a \wedge \neg a) = \mathsf{min}\{1, 1 - u_1(a)\} = \mathsf{min}\{1, 0\} = 0$; no segundo caso $\hat{u}_2(a \wedge \neg a) = \mathsf{min}\{0, 1 - u_2(a)\} = 0$. Concluímos que nenhuma valoração satisfaz $a \wedge \neg a$ logo esta fórmula não é satisfazível: $a \wedge \neg a \notin \mathcal{SAT}$.

Teorema 5.3.8 (Teorema de Cook) *$\mathcal{SAT}$ é* **NP***-completa.*

Para provarmos que uma linguagem $L \subseteq \mathbb{N}$ é **NP**-completa precisamos de mostrar que $L \in \mathbf{NP}$ e que $L \propto A$ para qualquer linguagem $A \in \mathbf{NP}$. No caso da linguagem $\mathcal{SAT}$ há uma complicação inicial: as fórmulas de $\mathcal{SAT}$ não são números naturais. Para ultrapassar esta dificuldade temos de definir uma codificação das fórmulas em números naturais. Além disso, em vez de trabalharmos com fórmulas conjuntivas arbitrárias, facilita-nos a demonstração escolher uma forma equivalente, mas com mais estrutura, os **conjuntos finitos de cláusulas** com três literais.

Portanto o esboço da demonstração deste teorema fica dividido em três fases:

1. Existe um esquema de código polinomial $e : \mathcal{SAT} \to \mathbb{N}$; Este esquema é a composição de três reduções polinomiais:
   (a) $e_1 : \mathcal{SAT} \to \mathcal{CFC}$ reduz fórmulas conjuntivas para conjuntos finitos de cláusulas (os elementos de $\mathcal{CFC}$ são os conjuntos finitos de cláusulas);
   (b) $e_2 : \mathcal{CFC} \to 3\mathcal{SAT}$ reduz conjuntos finitos de cláusulas para conjuntos finitos de cláusulas com três literais;
   (c) $e_3 : 3\mathcal{SAT} \to \mathsf{3SAT}$ ($\mathsf{3SAT} \subset \mathbb{N}$) reduz instâncias de $3\mathcal{SAT}$ para números naturais, por meio de um esquema de código semelhante ao que já foi usado para calcular os números de Gödel dos programas URM .
2. $\mathcal{SAT} \in \mathbf{NP}$; Exibir um algoritmo que verifique, em tempo polinomial, se uma fórmula (candidata) é satisfeita por uma dada valoração (que é o testemunho);

3. $\mathcal{SAT}$ é completa; Mostrar que, para qualquer linguagem $A \in \mathbf{NP}$, existe um algoritmo polinomial que converte instâncias de A em instâncias de $\mathcal{SAT}$;

A principal consequência deste teorema é que passamos a dispor de um método para estudar as relações entre linguagens de **NP** isto é, através de reduções polinomiais em instâncias de $\mathcal{SAT}$. Esta linguagem, porque é completa, tem a capacidade de representar *todas as restantes linguagens de* **NP**. Podemos concentrar os esforços para estudar a verificação das linguagens **NP** apenas numa só linguagem, $\mathcal{SAT}$, ou, se preferirmos, $3\mathcal{SAT}$, $\mathcal{CFC}$, 3SAT ou outra qualquer linguagem equivalente. Esta conclusão tem grande alcance: um algoritmo que *decida* $\mathcal{SAT}$ em tempo polinomial pode ser (polinomialmente) adaptado para decidir qualquer linguagem de **NP**. Se tal algoritmo existir, então

$$\mathbf{P} = \mathbf{NP}.$$

Por outro lado, se for provado que uma certa linguagem em **NP** não tem nenhum decisor polinomial fica assente que tem de ser

$$\mathbf{P} \neq \mathbf{NP}.$$

Vejamos agora um esboço (ainda que alargado) dos diferentes passos para demonstrar que a linguagem $\mathcal{SAT}$ é **NP**-completa. O primeiro passo é descrever algoritmicamente a redução polinomial e_1, que re-escreve uma fórmula conjuntiva α num conjunto finito de cláusulas equivalente. Ora para esta redução basta substituir (sintacticamente) os conectivos por vírgulas e os parênteses por chavetas. Por exemplo,

$$((a \vee b) \wedge (c \vee d)) \mapsto \{\{a, b\}, \{c, d\}\}.$$

Este algoritmo é polinomial no número de símbolos da fórmula:

A redução $e_1 : \mathcal{SAT} \to \mathcal{CFC}$ é polinomial.

O passo seguinte consiste em mostrar um algoritmo polinomial que transforme um conjunto finito de cláusulas numa instância de $3\mathcal{SAT}$, isto é, um conjunto finito de cláusulas em que cada cláusula tem, exactamente, três literais. Se necessário, pode rever estes conceitos na definição A.2.5, p.244 dos anexos. Em vez de apresentar formalmente esse algoritmo, vamos ilustrar como funciona com alguns exemplos. Um conjunto finito de cláusulas $\{C_1, \ldots, C_n\}$ é transformado processando uma cláusula de cada vez; Por sua vez, cada cláusula C_i é substituída pelas cláusulas $D_{i_1}, \ldots, D_{i_k}$, considerado o número de literais em C_i:

1. *se a cláusula tem exactamente uma literal*, $C = \{\lambda\}$: consideramos uma literal nova, ω, e substituímos C por $D = \{\lambda, \omega, \neg\omega\}$. Se $v \models C$ então $v(\lambda) = 1$ e $v(\lambda \vee \omega \vee \neg\omega) = 1$ também. Reciprocamente, se $v \models D$ então (ver caso a caso) $v(\lambda) = 1$. Conclusão: C e D são satisfazíveis pelas mesmas valorações: $\models C$ se, e só se $\models D$;

2. *se a cláusula tem exactamente duas literais*, $C = \{\lambda_1, \lambda_2\}$: consideramos uma literal nova, ω, e substituímos C por $D_1 = \{\lambda_1, \lambda_2, \omega\}$ e $D_2 = \{\lambda_1, \lambda_2, \neg\omega\}$. Se $v \models C$ então $v(\lambda_1) = 1$ ou $v(\lambda_2) = 1$. Então $v \models D_1$ e, também, $v \models D_2$. Reciprocamente, se $v \models D_1$ e $v \models D_2$ então tem de ser $v(\lambda_1) = 1$ ou $v(\lambda_2) = 1$. Portanto $\models C$ se, e só se $\models \{D_1, D_2\}$;

3. *se a cláusula tem exactamente três literais*, $C = \{\lambda_1, \lambda_2, \lambda_3\}$, fica $D = C$;

4. *se a cláusula tem quatro, ou mais literais*, $C = \{\lambda_1, \ldots, \lambda_n\}$. Consideremos, e.g., que tem cinco literais: $C = \{\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5\}$. Substituímos C por $D_1 = \{\lambda_1, \lambda_2, \omega_1\}$, $D_2 = \{\neg\omega_1, \lambda_3, \omega_2\}$, $D_3 = \{\neg\omega_2, \lambda_4, \lambda_5\}$. Analisando caso a caso (*i.e.* explorando as possibilidades de $v(\omega_i)$), concluímos que, também neste caso, $v \models C$ se, e só se $v \models \{D_1, D_2, D_3\}$; Nos casos em que C tem mais literais, são formados mais conjuntos «encadeados» como D_2.

Em qualquer dos casos ilustrados partimos de um conjunto de literais, C, e obtivemos novas cláusulas $\{D_1, \ldots, D_k\}$, equivalente a C, e em que

cada cláusula D_i tem, exactamente, três literais. Isto é, definimos e_2, que transforma instâncias de $\mathcal{CFC}$ em instâncias de $3\mathcal{SAT}$. O número de «passos elementares» nesta redução é polinomial no número de literais de C:

A redução $e_2 : \mathcal{CFC} \to 3\mathcal{SAT}$ é polinomial.

Para transformar instâncias de $3\mathcal{SAT}$, que são fórmulas, em instâncias de 3SAT, que são números naturais, é preciso recorrer a um esquema tipo código de Gödel. Uma forma é transformar sintacticamente as instâncias de $3\mathcal{SAT}$ em números naturais. Para isso definimos

$$\begin{array}{lcl} v_i & \mapsto & e_3(v_i) = 2i \\ \neg v_i & \mapsto & e_3(\neg v_i) = 2i+1 \\ \{\lambda_1, \lambda_2, \lambda_3\} & \mapsto & e_3(\{\lambda_1, \lambda_2, \lambda_3\}) = 2^{e_3(\lambda_1)} 3^{e_3(\lambda_2)} 5^{e_3(\lambda_3)} \\ \{C_1, \ldots, C_n\} & \mapsto & e_3(\{C_1, \ldots, C_n\}) = 2^{e_3(C_1)} \cdots p_n^{e_3(C_n)} \end{array} \tag{5.19}$$

Como funciona esta redução? Supondo que é dado um conjunto finito de cláusulas (tem de ser uma instância de $3\mathcal{SAT}$) com variáveis $v_1, \ldots$, por exemplo, $C = \{\{v_2, \neg v_3, v_1\}, \{\neg v_1, v_2, v_4\}\}$, resulta

$$e_3(C) = 2^{2^4 3^7 5^2} 3^{2^3 3^4 5^{16}}.$$

A redução e_3 depende (essencialmente) das funções soma, mult e exp, que são todas polinomiais[7]:

A redução $e_3 : 3\mathcal{SAT} \to 3\mathsf{SAT}$ é polinomial.

Compondo os vários algoritmos transformamos qualquer fórmula booleana, α, num número inteiro n que é o código numérico de um conjunto finito de cláusulas (com três literais cada cláusula) equivalente a α. Desta

[7] A função **exp** é polinomial. Com efeito, $a^b = 2^{b \mathsf{log}_2(a)}$. Para calcular 2^x (em notação binária) basta escrever $x-1$ dígitos «0» e um dígito 1 ($2^x = (10\cdots0)_2$ com $x-1$ dígitos 0) e $\mathsf{log}_2(a)$ é o número de dígitos binários em a.

forma concluímos o primeiro passo esboçado para demonstrar o teorema 5.3.8.

O passo seguinte na demonstração da completude **NP** de $\mathcal{SAT}$ é mostrar que este problema é verificável em tempo polinomial, isto é, que existe um algoritmo `Verificar` que, quando os argumentos são uma fórmula conjuntiva satisfazível α, e um testemunho y, `Verificar` termina em tempo polinomial (no tamanho da fórmula[8]) com resposta «sim». Este passo é bastante simples se considerarmos o caso em que o testemunho é um modelo v de α. Calculamos $v(\alpha)$ e testamos se é 1. Como o cálculo de $v(\alpha)$ é polinomial no tamanho de α, concluímos que $\mathcal{SAT} \in \mathbf{NP}$.

Finalmente, resta-nos mostrar que $\mathcal{SAT}$ é completo. Para resolver este passo é preciso um pouco de fôlego. Consideremos uma qualquer linguagem **NP**, digamos A. Podemos então dizer que existe um verificador de A, um certo algoritmo `Ver`. Além disso, se $x \in A$ então existe um testemunho $y \in \mathbb{N}$ tal que `Ver` (x, y) termina com resposta «sim», em tempo polinomial no tamanho de x. Consideremos a computação `Ver` (x, y), uma sequência de estados $S_0, S_1, \ldots$, governada pelo próprio programa `Ver` e pelos argumentos x, y. Com algum esforço podemos usar fórmulas booleanas para descrever

- o programa `Ver`;
- o estado inicial S_0;
- o efeito de cada instrução elementar na transição de um estado para o seguinte;
- cada estado intermédio S_i;
- se o resultado é «sim»;

Mais, podemos definir uma fórmula conjuntiva, Θ, que seja satisfazível se, e só se, $x \in A$, relacionando uma valoração que satisfaz Θ com um

[8]Não é necessário sermos excessivamente formais na definição de «tamanho de uma fórmula». Há várias possibilidades para esta função: a mais simples é a contagem de símbolos atómicos (variáveis, conectivos e parênteses).

testemunho de x:

$$\exists v : v(\Theta) = 1 \Leftrightarrow \exists y : \texttt{Ver}(x, y) = \text{sim}$$

A questão seguinte é *como definir tal fórmula*. Vejamos os passos iniciais. Seja $P = [I_1, \ldots, I_n]$ um programa BURM e r o maior registo referido nas instruções de P. Definimos um conjunto de *variáveis booleanas* para descrever as instruções que constam no programa. Para $j, k \in \{1, \ldots, n\}, p, q, u, v \in \{1 \ldots, r\}$, temos as variáveis

$$\begin{array}{ccc} Z_{j,p} & S_{j,p} & J_{j,p,q,k} \\ R_{j,p,q,u} & W_{j,p,q,u,v} & L_{j,p,q} \end{array} \tag{5.20}$$

Este conjunto de variáveis permite-nos construir uma fórmula conjuntiva que descreve o programa P. Por exemplo, $P = [\texttt{Z}(1), \texttt{S}(1)]$ é descrito pela fórmula

$$Z_{1,1} \wedge S_{2,1}$$

em que, na escrita

$$S_{2,1}$$

o «S» identifica o tipo de instrução, o primeiro índice (2) que é a segunda instrução do programa e o segundo e restantes índices (1) especificam os argumentos da instrução. Portanto a variável $S_{2,1}$ está a indicar que a segunda instrução do programa em causa é $\texttt{S}(1)$.

Uma valoração v que satisfaça esta fórmula tem $v(Z_{1,1}) = 1$. Pretendemos que a valoração informe qual é cada uma das instruções. Mas, como esta fórmula está definida, pode acontecer que também $v(S_{1,2}) = 1$ e ficamos sem saber, pela valoração v, que instrução é I_1. Por isso é preciso completar aquela fórmula com informação que exclua outras instruções de serem I_1:

$$\lambda^Z_{1,1} = \neg Z_{1,2} \wedge \cdots \wedge \neg Z_{1,r} \wedge \neg S_{1,1} \wedge \cdots \wedge \neg S_{1,r} \wedge \cdots \wedge \neg L_{1,1,1} \wedge \cdots \wedge \neg L_{1,r,r}$$

Agora a fórmula conjuntiva

$$\Lambda^Z_{1,1} \quad = \quad Z_{1,1} \wedge \lambda^Z_{1,1} \tag{5.21}$$

«garante» que a primeira instrução é apenas $\mathtt{Z}(1)$. À semelhança do que vimos para o caso da primeira instrução ser $\mathtt{Z}(1)$, podemos fazer fórmulas análogas para as restantes instruções, considerando as posição que ocupam e referem no programa e também os registos que manipulam. Obtemos então um conjunto de fórmulas conjuntivas

$$\begin{array}{ccc} \Lambda^Z_{j,p} & \Lambda^S_{j,p} & \Lambda^J_{j,p,q,k} \\ \Lambda^R_{j,p,q,u} & \Lambda^W_{j,p,q,u,v} & \Lambda^L_{j,p,q} \end{array} \tag{5.22}$$

que nos permite descrever exactamente cada instrução de cada programa BURM. Convencionemos então que cada programa BURM P é descrito por uma certa fórmula conjuntiva Λ_P, usando as fórmulas $\Lambda^I_{j,p,\ldots}$. Para o exemplo anterior, $P = [\mathtt{Z}(1), \mathtt{S}(1)]$, fica

$$\Lambda_P = \Lambda^Z_{1,1} \wedge \Lambda^S_{2,1}.$$

Agora precisamos de juntar fórmulas que descrevam a computação efectuada por P: qual é a instrução «activa», o conteúdo de cada registo e o efeito de cada instrução. Fixamos que $j, k \in \{1, \ldots, n\}, p, q, u, v \in \{1, \ldots, r\}, t, a, b, c, d \in \mathbb{N}$

- $Q_{j,t}$ se e só se no passo t a instrução «activa» é I_j;
- $R_{p,a,t}$ se e só se no passo t no registo R_p está o inteiro a;

Com estas novas variáveis já podemos definir um estado inicial com dois argumentos, x e y:

$$\Sigma_{x;y} = Q_{1,1} \wedge R_{1,x,1} \wedge R_{2,y,1} \wedge R_{3,0,1} \wedge \cdots \wedge R_{r,0,1} \tag{5.23}$$

Vejamos como descrever o efeito de cada instrução[9]. A título de exemplo,

$$\Delta^Z_{j,t,p} = (Q_{j,t} \wedge Z_{j,p}) \Rightarrow (Q_{j+1,t+1} \wedge R_{p,0,t+1}).$$

[9]Usamos $a \Rightarrow b$ como abreviatura da fórmula conjuntiva $\neg b \vee a$, o que facilita a escrita e a leitura. Mas é importante não esquecer que $a \Rightarrow b$ não é uma fórmula, mas apenas uma convenção de escrita.

Esta fórmula especifica que, se $\mathtt{Z}(p)$ for a instrução «activa», o valor do registo R_p passa a zero nesse passo e a próxima instrução activa é a que lhe segue em P. Fica como **exercício** definir fórmulas que descrevam as transições das restantes instruções. Temos então, um novo conjunto de fórmulas

$$\begin{array}{ccc} \Delta^{Z}_{j,t,p} & \Delta^{S}_{j,t,p,a} & \Delta^{J}_{j,t,p,q,k,a} \\ \Delta^{R}_{j,t,p,q,u,a,b} & \Delta^{W}_{j,t,p,q,u,v,a,b,c,d} & \Delta^{L}_{j,t,p,q,a} \end{array} \tag{5.24}$$

que garantem a correcta descrição da evolução da computação.

É ainda preciso descrever o estado final de uma computação, isto é, no instante t a instrução é maior que n e, no registo 1 está o valor y:

$$\Omega_{t;y} = R_{1,y,t} \wedge Q_{n+1,t} \tag{5.25}$$

É importante observar que a variável $Q_{n+1,t}$ «afirma» que o estado é final, de acordo com a convenção que nas instruções $\mathtt{J}(i,j,p)$, se for $p > n$ então tem de ser $p = n+1$.

Quantas variáveis são necessárias para descrever a computação que P faz, em T passos elementares? Supondo que o estado inicial é descrito por $\Sigma_{x,y}$, o maior valor possível para os índices t, a, b, c e d é (seguramente majorado por) $M = T + \mathsf{max}(x,y)$; portanto bastam

1. n variáveis para descrever as instruções;
2. $r+1$ variáveis para descrever o estado inicial;
3. $r \times M$ variáveis $Q_{j,t}$;
4. $n \times M^2$ variáveis $R_{p,a,t}$;
5. $n \times M \times r$ «variáveis» $\Delta^{Z}_{j,t,p}$;
6. $n \times M^2 \times r$ «variáveis» $\Delta^{S}_{j,t,p,a}$;
7. $n^2 \times M^2 \times r^2$ «variáveis» $\Delta^{J}_{j,t,p,q,k,a}$;

8. $n \times M^3 \times r^3$ «variáveis» $\Delta^R_{j,t,p,q,u,a,b}$;
9. $n \times M^5 \times r^4$ «variáveis» $\Delta^W_{j,t,p,q,u,v,a,b,c,d}$;
10. $n \times M^2 \times r^2$ «variáveis» $\Delta^L_{j,t,p,q,a}$;

Ora, somando estes valores todos, obtemos um polinómio em n, M e r. Se, por sua vez, T for majorado por um polinómio em $|x|$, podemos concluir que existe uma fórmula conjuntiva $\Theta_{P;x}$ que descreve a computação de $P(x)$ com um número polinomial (em $|x|$) de variáveis.

Voltemos à linguagem A e suponhamos que $x \in A$. Então existe um testemunho y de x, de forma que $\texttt{Ver}(x; y) \downarrow 1$. A fórmula conjuntiva

$$\Theta_y = \Lambda_{\texttt{Ver}} \wedge \Sigma_{x;y} \wedge \Omega_{|x|^k;1} \tag{5.26}$$

tem um número polinomial de variáveis e sabemos que é válida: um modelo desta formula é uma valoração que «acompanha» a computação $\texttt{Ver}(x; y)$: temos a equivalência

$$\models \Theta_y \Leftrightarrow \texttt{Ver}(x; y) \downarrow 1$$

porque a fórmula Θ_y está construída de forma a reflectir as regras e também cada passo da computação $\texttt{Ver}(x; y)$; se Θ_y tiver um modelo, as variáveis «verdadeiras» nessa valoração descrevem uma computação de $\texttt{Ver}(x; y)$ que termina num número polinomial de passos elementares, com resultado 1.

A demonstração da completude de $\mathcal{SAT}$ está concluída: cada linguagem $A \in \mathbf{NP}$ tem um verificador $\texttt{Ver}$ e as computações de «aceitação» feitas por $\texttt{Ver}$ são «equivalentes» à validade de uma certa fórmula Θ. Isto é, escrevemos sob a forma de uma instância de $\Theta \in \mathcal{SAT}$ a questão de saber se $x \in A$.

Com a demonstração de que $\mathcal{SAT}$ é **NP**-completo concluímos a apresentação dos principais resultados teóricos da teoria da complexidade. De seguida vamos seguir um rumo mais «aplicado», estudando a relação com a Teoria dos grafos.

5.4 Exemplos de Teoria dos Grafos

Aqui vamos ilustrar, na teoria dos grafos, os conceitos da complexidade computacional que acabámos de ver. Ajuda, neste momento, rever o resumo dedicado aos grafos, em A.1, p.233. A atenção dada à teoria dos grafos tem boas justificações: não só as principais noções e resultados são intuitivamente acessíveis como também é um corpo da matemática bem estudado, com muitos resultados sólidos e aplicações em diversas áreas.

O principal elo entre a complexidade e os grafos estabelece que um certo problema sobre grafos é **NP**-completo. Começamos por definir esse problema particular e logo de seguida demonstramos que é **NP**-completo.

Definição 5.4.1 (O problema da cobertura de vértices ($\mathcal{COBV}$)) *As* instâncias *do* ***problema da cobertura de vértices****, $\mathcal{COBV}$, são pares (G, k) em que G é um grafo e $k \in \mathbb{N}$; Uma instância (G, k) é* positiva *se tem uma cobertura de vértices de tamanho máximo k.*

O próximo teorema é interessante, não só pelo seu enunciado mas, também, pela técnica empregue na demonstração. Nesta, depois de mostrarmos que $\mathcal{COBV}$ é um problema **NP**, precisamos verificar que é completo. Para provar a completude definimos uma redução polinomial, f, que associa a cada fórmula $\alpha \in 3\mathcal{SAT}$ um grafo $f(\alpha) = G \in \mathcal{COBV}$ tal que α é válida se e só se G tem uma cobertura de vértices.

Porque é que estamos a reduzir $3\mathcal{SAT} \propto_f \mathcal{COBV}$ e não ao contrário? Seja X um problema **NP** qualquer. Como já sabemos que $3\mathcal{SAT}$ é **NP**-completo, existe uma redução, g, tal que $X \propto_g 3\mathcal{SAT}$. Portanto, a composição $h = f \circ g$ reduz X a $\mathcal{COBV}$: $X \propto_h \mathcal{COBV}$.

Para provar que um dado problema X é **NP**-completo, podemos seguir o seguinte processo:

1. provar que $X \in \mathbf{NP}$, exibindo um algoritmo polinomial para computar um verificador de X;

2. escolher um problema Y, **NP**-completo;

3. definir um algoritmo polinomial que transforme instâncias de Y em instâncias de X;

Teorema 5.4.2 *$\mathcal{COBV}$ é* **NP***-completo.*

Demonstração. Primeiro demonstramos que $\mathcal{COBV} \in \mathbf{NP}$.

Obtemos um algoritmo polinomial[10] para verificar se uma instância $((V, A), k) \in \mathcal{COBV}$ (o candidato), com testemunho $U \subseteq V$, é positiva considerando que os testes necessários são:

1. $|U| \leq k$: basta «ir contando» elementos de U; se estes se «esgotarem» antes de chegarmos a k, U é aceitável; se, pelo contrário, contarmos $k+1$ elementos distintos em U, este testemunho não serve; Para este passo são suficientes $k+1$ operações elementares do tipo «obter o próximo elemento da lista X»;

2. Cada aresta $\{a, b\} \in A$ tem pelo menos um vértice em U: É necessário confirmar se $a \in U \vee b \in U$ para cada aresta $(a, b) \in A$; Para este passo são suficientes $2 \times |A| \times k$ operações elementares do tipo «o elemento x ocorre na lista X?»;

De seguida provamos a parte «completo» de «$\mathcal{COBV}$ é **NP**-completo». Para isso definimos uma redução polinomial de $3\mathcal{SAT}$ em $\mathcal{COBV}$.

Sejam $C = \{C_1, \ldots, C_m\}$ uma instância de $3\mathcal{SAT}$ (portanto cada $C_j \in C$ tem exactamente três literais) e $\mathrm{Var}(C) = \{\alpha_1, \ldots, \alpha_n\}$ o conjunto das variáveis que ocorrem em C. Definimos três grupos de grafos, para representar (1) as variáveis $\alpha_1, \ldots, \alpha_n$; (2) as cláusulas $C_1, \ldots, C_m$ e (3) as relações entre as variáveis e as cláusulas.

Para cada variável $\alpha \in \mathrm{Var}(C)$ definimos um grafo

$$G_\alpha = (V_\alpha = \{\alpha, \neg\alpha\}, A_\alpha = \{(\alpha, \neg\alpha)\})$$

[10] Há várias formas de «medir» o tamanho de um grafo. Podemos, por exemplo, somar o número de vértices e de arestas. Para as instâncias de $\mathcal{COBV}$, uma medida do tamanho das instâncias pode ser dado, por exemplo, por $|V| + |A| + \log_2(k)$.

Para cada cláusula $C_j = \{a, b, c\} \in C$ definimos um grafo

$$G'_j = \left(V'_j = \{a, b, c\}, A'_j = \{(a, b), (b, c), (c, a)\}\right)$$

Para relacionar as n variáveis com as cláusulas, começamos por enumerar as $2n$ literais:

positivas $\lambda_i = \alpha_i$, para $i = 1, 2, \ldots, n$;

negativas $\lambda_{n+i} = \neg\alpha_i$, para $i = 1, 2, \ldots, n$.

Por exemplo, se as variáveis de C forem α_1, α_2, ficamos com

$$\lambda_1 = \alpha_1, \lambda_2 = \alpha_2, \lambda_3 = \neg\alpha_1, \lambda_4 = \neg\alpha_2$$

Agora, usando esta enumeração das literais, podemos re-escrever cada cláusula

$$C_j = \{a, b, c\} \quad = \quad \{\lambda_{j_1}, \lambda_{j_2}, \lambda_{j_3}\}$$

e ligar as literais às cláusulas

$$A''_j \quad = \quad \{(a, \lambda_{j_1}), (b, \lambda_{j_2}), (c, \lambda_{j_3})\}.$$

Finalmente, podemos definir o grafo G, que representa C:

$$\begin{aligned}
V &= \left(\bigcup_{\alpha \in \mathrm{Var}(C)} V_\alpha\right) \cup \left(\bigcup_{j=1}^{m} V'_j\right); \\
A &= \left(\bigcup_{\alpha \in \mathrm{Var}(C)} A_\alpha\right) \cup \left(\bigcup_{j=1}^{m} A'_j\right) \cup \left(\bigcup_{j=1}^{m} A''_j\right); \\
G &= (V, A); \\
k &= n + 2m.
\end{aligned}$$

Este grafo pode ser construído, seguindo estes passos, a partir de C em tempo polinomial. Resta mostrarmos que

C é satisfazível se, e só se, existe uma cobertura de vértices de G de tamanho máximo k.

Observando a construção de G concluímos que, existindo, uma cobertura de vértices de G terá de conter, necessariamente, pelo menos

1. um elemento de cada V_α, para cada variável α; porque se nenhum vértice de V_α estiver na cobertura, a aresta $(\alpha, \neg\alpha)$ não fica coberta;

2. dois elementos de cada V'_j, para cada cláusula C_j; porque se apenas um, ou nenhum, vértice de V'_j estiver na cobertura, pelo menos uma aresta de V'_j não fica coberta;

Portanto,

uma eventual cobertura de vértices de G com $k = n + 2m$ elementos tem, exactamente, um vértice em cada V_i e dois vértices em cada V'_j.

Para uma tal cobertura, U, definimos a valoração

$$v(\alpha) = \begin{cases} 1 & \text{se } \alpha \in U \\ 0 & \text{se } \neg\alpha \in U \end{cases} .$$

Exercício: esta igualdade define uma valoração porque exactamente um de $\alpha, \neg\alpha$ está em U; *porquê*?

Agora mostramos que $v \models C$. Para isso, consideremos uma cláusula $C_j \in C$ e as arestas A''_j. Apenas duas destas arestas têm um vértice em $V'_j \cap U$. Portanto, existe uma terceira aresta em A''_j com um vértice em U e o outro num certo A'_i. Para a literal λ correspondente à cláusula C_j temos então $v(\lambda) = 1$. Portanto, para cada cláusula C_j, $\hat{v}(C_j) = 1$ donde $\hat{v}(C) = 1$, isto é, $v \models C$ e concluímos que C é satisfazível (uma instância positiva de $3\mathcal{SAT}$).

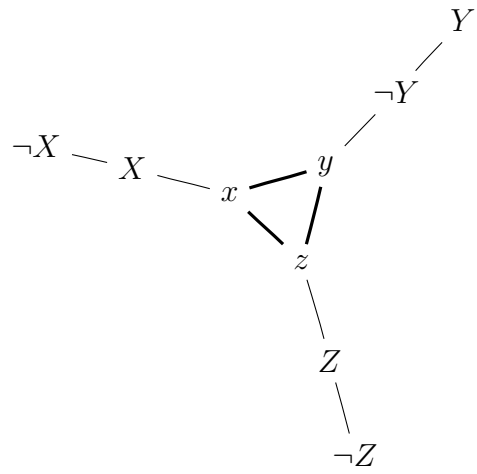
Reciprocamente, se v for um modelo de C, seja

$$U_0 = \{\alpha \mid v(\alpha) = 1\} \cup \{\neg\alpha \mid v(\alpha) = 0\}$$

o conjunto das *literais* verdadeiras.

Então U_0 contém um vértice de, pelo menos, uma aresta de A''_j (porque tem de ser $\hat{v}(C_j) = 1$). Acrescentamos então a U_0 os vértices de V'_j das outras arestas de A''_j. O conjunto U que resulta de aplicar este procedimento a todas as cláusulas é uma cobertura de vértices de G, e tem, no máximo, $k \leq n + 2m$ elementos.

A título de ilustração, consideremos a figura seguinte:



A cláusula C_j está representada pelos vértices $\{x, y, z\}$ e as arestas A''_j são (x, X), $(y, \neg Y)$ e (z, Z). Apenas dois dos vértices x, y, z podem estar em U. Supondo que $z \notin U$, para que a aresta (z, Z) fique coberta por U, tem de ser $Z \in U$. Portanto, $v(Z) = v(z) = 1$ e $\hat{v}(x \vee y \vee z) = 1$, isto é, $\hat{v}(C_j) = 1$. Como este argumento pode ser aplicado a todas as cláusulas, concluímos que $\hat{v}(C) = 1$.

Reciprocamente, se, por exemplo, $Z \in U_0$ então acrescentamos x e y a U_0. Fazendo isto a todas as cláusulas, o conjunto U que resulta cobre todos os vértices de G e com o número de elementos indicado.

□

A «equivalência» entre *validade* de fórmulas e *cobertura* de vértices de grafos, que acabámos de provar, ilustra a diversidade dos domínios onde podemos encontrar as noções da complexidade (e, claro, da computação).

Antes de explorarmos, dentro da Teoria dos grafos, a completude **NP** de $\mathcal{COBV}$ vamos desenvolver um exemplo. Consideremos a fórmula booleana

$$\phi = a \Rightarrow (b \wedge c)\,.$$

Esta fórmula é satisfazível (**exercício:** porquê?). Portanto, deve haver alguma instância positiva $G \in \mathcal{COBV}$ que lhe corresponda. Vamos construir essa instância, seguindo os passos da demonstração do teorema 5.4.2. Porém, antes de podermos «entrar» na demonstração, temos de converter ϕ numa instância de $3\mathcal{SAT}$. Uma forma normal conjuntiva de ϕ é

$$(\neg a \vee b) \wedge (\neg a \vee c)$$

mas ainda não é uma instância de $3\mathcal{SAT}$. Para chegarmos a essa fase precisamos de converter esta fórmula no cfc

$$\{\{\neg a, b\}, \{\neg a, c\}\}$$

e acrescentar variáveis auxiliares x, y para que cada cláusula tenha exactamente três literais

$$C = \{\{\neg a, b, x\}, \{\neg a, b, \neg x\}, \{\neg a, c, y\}, \{\neg a, c, \neg y\}\}\,.$$

O cfc C é uma instância de $3\mathcal{SAT}$ que representa a fórmula booleana ϕ. Podemos agora seguir os passos da demonstração do teorema para construir um certo grafo G:

1. C tem $m = 4$ cláusulas e as $n = 5$ variáveis $\mathrm{Var}\,(C) = \{a, b, c, x, y\}$; portanto procuramos uma cobertura de $k = 2 \times 4 + 5 = 13$ vértices;

2. para cada variável de $\mathrm{Var}\,(C)$ definimos um grafo

   - $G_a = (\{a, \neg a\}, \{(a, \neg a)\})$
   - $G_b = (\{b, \neg b\}, \{(b, \neg b)\})$
   - $G_c = (\{c, \neg c\}, \{(c, \neg c)\})$
   - $G_x = (\{x, \neg x\}, \{(x, \neg x)\})$

- $G_y = (\{y, \neg y\}, \{(y, \neg y)\})$

3. para cada cláusula de C também definimos um grafo

   - $G'_1 = (\{\neg a_1, b_1, x_1\}, \{(\neg a_1, b_1), (b_1, x_1), (x_1, \neg a_1)\})$
   - $G'_2 = (\{\neg a_2, b_2, \neg x_2\}, \{(\neg a_2, b_2), (b_2, \neg x_2), (\neg x_2, \neg a_2)\})$
   - $G'_3 = (\{\neg a_3, c_3, y_3\}, \{(\neg a_3, c_3), (c_3, y_3), (y_3, \neg a_3)\})$
   - $G'_4 = (\{\neg a_4, c_4, \neg y_4\}, \{(\neg a_4, c_4), (c_4, \neg y_4), (\neg y_4, \neg a_4)\})$

4. ligamos as «variáveis» do primeiro conjunto de grafos aos vértices que lhes correspondem nos grafos do segundo conjunto. Isto é, juntamos as arestas

$$\begin{gathered}(\neg a, \neg a_1), (\neg a, \neg a_2), (\neg a, \neg a_3), (\neg a, \neg a_4), \\ (b, b_1), (b, b_2), \\ (c, c_3), (c, c_4), \\ (x, x_1), (\neg x, \neg x_2), \\ (y, y_3), (\neg y, \neg y_4)\end{gathered}$$

5. no grafo resultante, ilustrado na figura 5.3, pretendemos relacionar a validade da fórmula ϕ com a existência uma cobertura de $k = 13$ vértices;

6. consideremos, por exemplo, U uma cobertura de 13 vértices de G (**exercício:** verifique que U é mesmo uma cobertura de vértices de G):

$$U = \{\neg a, b, c, \neg x, \neg y, b_1, x_1, b_2, x_2, c_3, y_3, c_4, \neg y_4\}$$

   (a) a cobertura U define uma valoração u de a, b, c, x, y:

$$u = \begin{pmatrix} a & b & c & x & y \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

(b) a extensão desta valoração é um modelo de ϕ:

$$\begin{aligned}\hat{u}(\phi) &= \hat{u}(a \Rightarrow (b \wedge c)) \\ &= \hat{u}(\neg a \vee (b \wedge c)) \\ &= \mathsf{max}\{\hat{u}(a), \hat{u}(b \wedge c)\} \\ &= \mathsf{max}\{1 - u(a), u(b) \times u(c)\} \\ &= \mathsf{max}\{1, 1\} = 1\end{aligned}$$

7. por outro lado, um modelo de ϕ define uma cobertura de 13 vértices de G:

   (a) consideremos um modelo de ϕ

   $$v = \begin{pmatrix} a & b & c & x & y \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

   (**exercício:** verifique que v é mesmo modelo de ϕ)

   (b) começamos por definir V_0 a partir de v, juntando as literais verdadeiras (isto é, as literais λ com $v(\lambda) = 1$):

   $$V_0 = \{\neg a, b, c, x, y\}$$

   (c) os vértices de V_0 cobrem todas as arestas de todos nos subgrafos G_α (com $\alpha \in \mathrm{Var}(C)$);

   (d) resta cobrir as arestas dos grafos G'_i e as arestas que ligam estes dois conjuntos de grafos; para isso basta «seguir» as arestas que «saem» de $\neg a$ para os G'_i (indo-se parar aos vértices $\neg a_i$) e juntar os outros vértices dos G'_i :

   $$V_1 = \{b_1, x_1, b_2, \neg x_2, c_3, y_3, c_4, \neg y_4\}$$

   (e) fazendo $V = V_0 \cup V_1$ têm-se 13 vértices que cobrem todas as arestas de G (**exercício:** verifique):

   $$V = \{\neg a, b, c, x, y, b_1, x_1, b_2, \neg x_2, c_3, y_3, c_4, \neg y_4\}$$

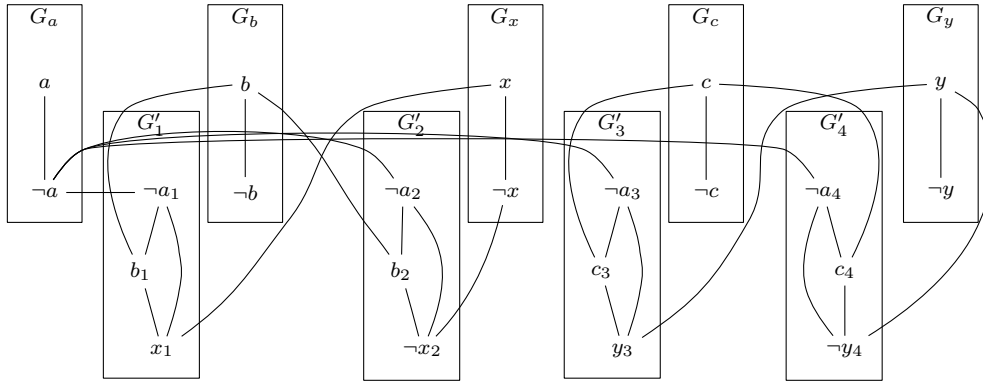


Figura 5.3: O grafo G que resulta da redução da fórmula $\phi = a \Rightarrow (b \wedge c)$ a uma instância de $\mathcal{COBV}$.

Agora que conhecemos de um problema **NP**-completo na teoria dos grafos, podemos explorar as equivalências entre problemas de grafos para descobrir mais problemas que também são **NP**-completos. É essa exploração que começamos no próximo teorema.

Teorema 5.4.3 *Os três problemas seguintes são* **NP***-completos:*

$\mathcal{COBV}$ *descrito acima (definição 5.4.1);*

$\mathcal{CONI}$ *(conjunto independente) As instâncias são pares* (G, k) *em que* G *é um grafo e* k *um número natural; Uma instância* (G, k) *é* positiva *se* G *tem um conjunto independente de tamanho mínimo* k*;*

$\mathcal{CLIQ}$ *(clique) As instâncias são pares* (G, k) *em que* G *é um grafo e* k *um número natural; Uma instância* (G, k) *é* positiva *se* G *tem um clique de tamanho mínimo* k*;*

Demonstração. Segue-se das equivalências do teorema A.1.2 nos anexos, do facto de cada uma das equivalências na demonstração desse teorema definir uma redução polinomial entre instâncias e, por fim, do teorema 5.4.2. □

Terminamos a exploração entre a complexidade computacional e os grafos com um exercício resolvido (ligado ao conhecido **problema do caixeiro viajante**).

5.5 Exercícios

Exercícios Resolvidos

Exercício 5.5.1 *Mostre que o problema $\mathcal{CIRH}$ (circuito hamiltoniano) — cujas instâncias são grafos e uma instância é* positiva *se tiver um circuito hamiltoniano — é* **NP***-completo.*

Este exercício vai ser resolvido seguindo o processo de três fases antes usado para provar que $\mathcal{COBV}$ é **NP**-completo:

1. provar que $\mathcal{CIRH} \in \mathbf{NP}$, exibindo um algoritmo polinomial para computar um verificador de $\mathcal{CIRH}$;
2. escolher $\mathcal{COBV}$ como problema **NP**-completo já conhecido;
3. definir uma transformação polinomial das instâncias de $\mathcal{COBV}$ em instâncias de $\mathcal{CIRH}$;

É na terceira fase que vamos investir a maior parte do esforço. Entretanto consideremos $G' = (V', A')$ uma instância positiva de $\mathcal{CIRH}$ e $v = (v_1, \ldots, v_n)$ um caminho hamiltoniano em G'. Para verificar que v é, de facto, hamiltoniano basta confirmar que os vértices de G são mesmo $v_1, \ldots, v_n$ (isto é, que $v = V'$) e que cada segmento (v_i, v_{i+1}), e também (v_n, v_1), são arestas de G'. Estas verificações podem ser feitas em tempo polinomial no número de arestas de G'.

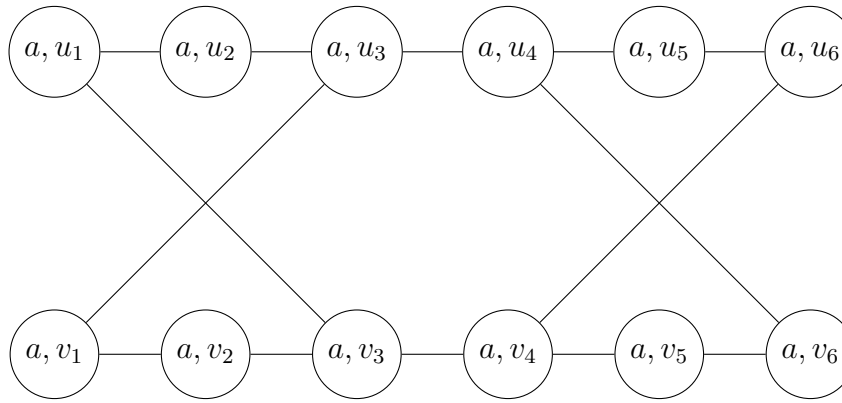
O problema **NP**-completo que vamos reduzir a $\mathcal{CIRH}$ é $\mathcal{COBV}$. Assim temos de definir um algoritmo polinomial que transforme uma instância $(G = (V, A), k)$ de $\mathcal{COBV}$ numa instância $G' = (V', A')$ de $\mathcal{CIRH}$. Além disso, essa transformação tem de ser de forma que (G, k) tenha uma cobertura de k vértices se, e só se, G' tem um caminho hamiltoniano. O esquema para esta redução é o seguinte:

1. a cada aresta (u, v) de G associamos um certo sub-grafo $T_{u,v}$, a que chamamos «componente de teste»;
2. cada vértice em G é representado por uma certa «cadeia» em G', que liga as «componentes de teste» das arestas que incidem nesse vértice;
3. a cada vértice v da cobertura de G associamos um «vértice de selecção» α_v de G';
4. ligamos cada α_v ao princípio e fim de cada «cadeia».

Portanto, *primeiro*, a cada aresta $a = (u, v)$ de G associamos a «componente de teste» T_a definido

- por doze vértices $(a, u_1), \ldots, (a, u_6), (a, v_1), \ldots, (a, v_6)$;
- por dez arestas $((a, u_i), (a, u_{i+1}))$ e $((a, v_i), (a, v_{i+1}))$, para $i = 1, \ldots, 5$;
- e ainda mais quatro arestas $((a, u_1), (a, v_3))$, $((a, v_1), (a, u_3))$, $((a, u_4), (a, v_6))$, $((a, v_4), (a, u_6))$.

Cada grafo T_a fica com o seguinte aspecto:

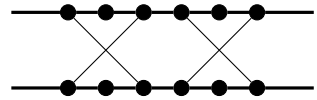


Nas «componentes de teste» estamos interessados nos «cantos»: (a, u_1), (a, u_6), (a, v_1) e (a, v_6). A cada vértice x da aresta corresponde, na componente, uma «linha» $(a, x_1) - (a, x_2) - \cdots - (a, x_6)$ conexa.

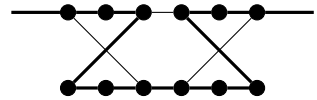
Importa agora observar que

cada «componente de teste» só *pode ser percorrida, num eventual circuito hamiltoniano de G', por uma das três formas seguintes:*

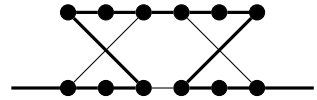
- ambos os vértices estão na cobertura:



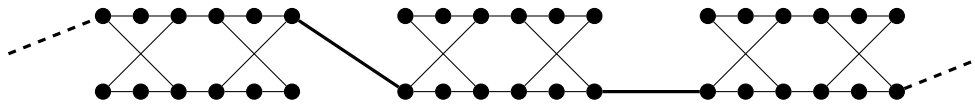
- só o vértice «de cima» está na cobertura:



- só o vértice «de baixo» está na cobertura:



Segundo, para cada vértice v de G, consideremos as arestas que passam em v, com uma ordem arbitrária $a_1, \ldots, a_m$. Cada uma destas arestas está associada a uma certa «componente de teste» T_{a_i}. Vamos ligar essas componentes pelos «cantos» dos vértices relativos a v, seguindo a ordem escolhida $a_1, \ldots, a_m$, juntando as arestas seguintes: $((a_1, v_6), (a_2, v_1))$, $((a_2, v_6), (a_3, v_1))$, *etc.* O resultado é um grafo com «cadeias» da forma



e cada «cadeia» representa a junção de arestas num certo vértice:

Seja v um vértice de G e $a_1, \ldots, a_m$ as arestas que incidem em v. As «componentes de teste» dessas arestas formam uma «cadeia».

Terceiro, a cada vértice v da cobertura de G acrescentamos um «vértice de selecção» α_v em G'.

Quarto, ligamos cada vértice α_v de G' ao princípio e fim de todas as «cadeias» definidas no segundo passo. Estas ligações são feitas por arestas

$$(\alpha_v, (a_1, u_1)), (\alpha_v, (a_m, u_6))$$

em que u e v são vértices de G e $a_1, \ldots, a_m$ são as arestas de G que se juntam em u.

Fica como **exercício** confirmar que dada uma instância (G, k) de $\mathcal{COBV}$, o grafo G' é assim definido em tempo polinomial (no número de arestas de G).

Agora é necessário mostrar que G tem uma cobertura com k vértices se, e só se, G' tem um circuito hamiltoniano.

Seja G um grafo no qual seleccionamos k vértices e usamos para definir G'. *Supondo que existe um circuito hamiltoniano em* G' — para concluir que os k vértices seleccionados formam uma cobertura de G — consideremos um segmento deste caminho que começa e acaba em «vértices de selecção», sem passar por mais nenhum «vértice de selecção».

Pela forma como as «componentes de teste» estão definidas e ligadas, entre si e aos «vértices de selecção», podemos concluir que

> *este segmento percorre a «cadeia» de «componentes de teste» associadas às arestas que incidem num certo vértice* v *de* G.

Portanto os k segmentos $\alpha_1 \to \alpha_2, \ldots, \alpha_k \to \alpha_1$ dividem o circuito hamiltoniano em k segmentos, com cada segmento associado a um certo vértice de G. Sejam $v_1, \ldots, v_k$ esses vértices.

Vejamos que estes vértices formam um cobertura de G. Seja $a = (a_1, a_2)$ uma aresta de G e T_a a sua «componente de teste» em G'. Os vértices de T_a são percorridos pelo circuito hamiltoniano, numa das três formas possíveis. Portanto cada lado (o «de cima», o «de baixo», ou

ambos) da «componente de teste» T_a é percorrido por um certo segmento do circuito hamiltoniano. Mas isso significa que a aresta a incide no vértice associado a esse segmento.

Portanto qualquer aresta de G incide num certo v_i, isto é, $v_1, \ldots, v_k$ é uma cobertura de vértices de G.

Reciprocamente, *supondo que os k vértices seleccionados,* $v_1, \ldots, v_k$, *são uma cobertura de* G — para concluir que G' tem um circuito hamiltoniano — consideremos os conjuntos de arestas:

A_1 (componentes) formado, para cada «componente de teste», pelas arestas que correspondem à forma (das três possíveis) de percorrer essa componente, conforme um dos vértices, ou ambos, pertencem à cobertura. Tem de acontecer exactamente um dos três casos pois ou um vértice ou ambos os vértices da aresta está na cobertura;

A_2 (cadeias) formado, para cada vértice na cobertura, pelas arestas que ligam a «cadeia» associada a esse vértice, isto é, pelas arestas que ligam o «canto de saída» desse vértice numa «componente de teste» ao «canto de entrada» na componente seguinte;

A_3 (entradas) formado, para cada vértice v_i na cobertura, pela aresta que liga α_{v_i} ao vértice «de entrada» na «cadeia» de componentes que corresponde ao vértice v_i;

A_4 (saídas) formado, para cada vértice v_{i+1} na cobertura, pela a aresta que liga α_{v_i} ao vértice «de saída» na «cadeia» de componentes que corresponde ao vértice v_i (fazendo $v_{k+1} = v_1$);

Usamos as arestas de $\bigcup_{i=1}^{4} A_i$ para construir um caminho em G':

1. começamos em α_{v_1}. Em A_3 há uma aresta que liga α_{v_1} ao princípio da «cadeia» associada a v_1. Juntamos o outro vértice desta aresta. Este vértice está na «entrada» de uma «componente de teste»;

2. as arestas em A_1 definem um caminho que liga este vértice à «saída» da «componente de teste». Juntamos esses vértices, seguindo a ordem dada por estas arestas;

3. o vértice de «saída» da «componente de teste» ou incide numa aresta (em A_2) que liga essa componente a outra, ou incide numa aresta (em A_4) que liga a componente ao «vértice de selecção» seguinte;

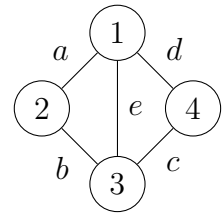
   (a) no primeiro caso juntamos o primeiro vértice da próxima componente da «cadeia» e voltamos ao ponto 2;

   (b) no segundo caso juntamos o «vértice de controlo» seguinte mais o primeiro vértice da «componente de controlo» da «cadeia» associada a esse vértice e voltamos ao ponto 2.

Pela forma como foi construído, este caminho é um circuito pois cada vértice está ligado ao seguinte por uma aresta de G' e o «vértice de selecção» α_{v_1} está ligado ao vértice de «saída» da última «componente de teste» da «cadeia» de v_k. Resta mostrar que este circuito percorre todos os vértices de G'.

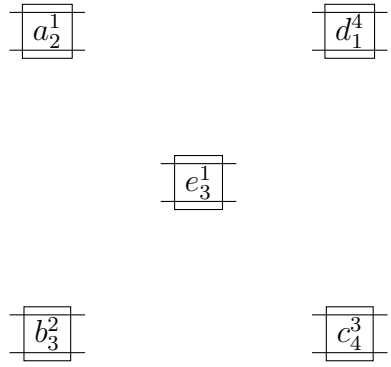
Os vértices de G' ou estão numa «componente de teste» ou são «vértices de selecção». Comecemos por α_{v_1}. Por construção este vértice está no circuito. Portanto todos os vértices que, em G', representam v_1 estão no circuito. Além disso, também passamos por todos os vértices de G' que representam vértices de G que (a) não estão na cobertura e (b) formam, em G, arestas com v_1. No fim deste percurso encontramos o «vértice de selecção» α_{v_2}. Agora podemos aplicar o mesmo raciocínio à «cadeia» que representa v_2. Indutivamente, percorremos todas as cadeias de todos os vértices da cobertura. Mas, precisamente por ser uma cobertura, todas as arestas de G foram visitadas, isto é, nenhuma «componente de teste» ficou com vértices por serem percorridos.

Consideremos um exemplo concreto. Seja G o grafo

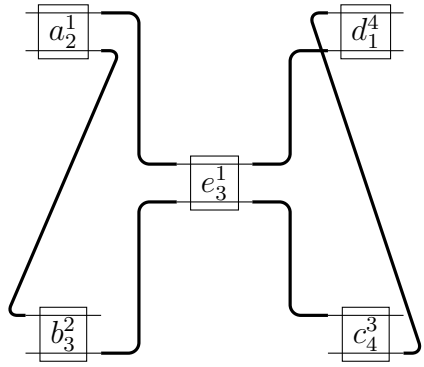


com vértices $1, 2, 3, 4$ e arestas a, b, c, d, e. Uma cobertura de vértices de tamanho 2 é $\{1, 3\}$. Seguindo a demonstração, primeiro definimos, para cada aresta $a = (u, v)$, uma «componente de teste» $T_{u,v}$, que representamos graficamente por

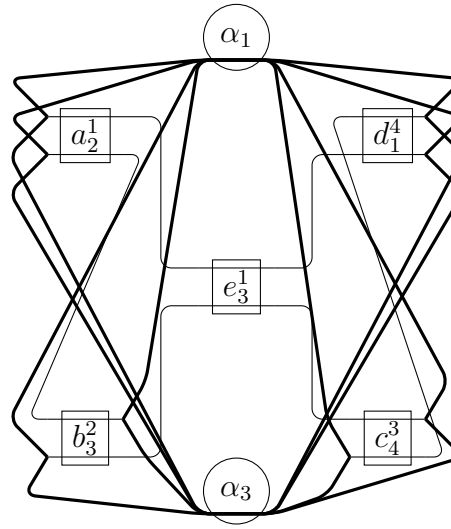
de forma a destacar o nome da aresta (a), os vértices (u, «de cima» e v, «de baixo») e ilustrar os pontos de «entrada» (à esquerda) e de «saída» (à direita). Assim, o grafo das componentes de teste é



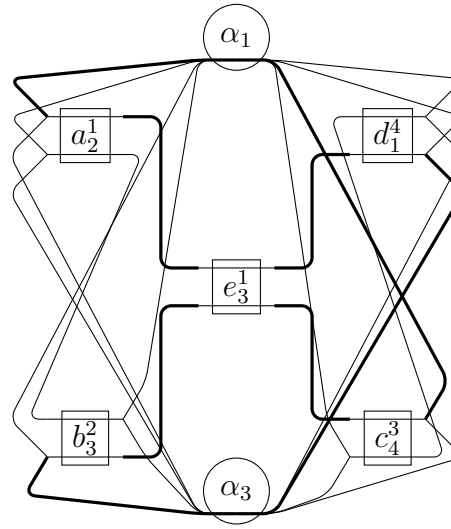
O segundo passo da construção de G' consiste em fazer as «cadeias» de cada vértice, ligando (por ordem arbitrária) as componentes que se juntam nesse vértice. Obtemos então quatro «cadeias»:



Os vértices da cobertura definem α_1 e α_3, que são ligados ao princípio e ao fim de cada «cadeia»:



Neste grafo podemos identificar o seguinte circuito hamiltoniano:



Neste circuito os vértices de cada «componente de teste» $T_{u,v}$ são percorridos por uma das três formas possíveis. Por exemplo, e_3^1 é percorrido pela primeira forma, a_2^1 e c_4^3 pela segunda forma e b_3^2 e d_1^4 pela terceira forma. Consideremos o segmento $\alpha_1 \to a_2^1 \to e_3^1 \to d_1^4 \to \alpha_3$; este segmento está associado ao vértice 1 de G e percorre as componentes de

teste a_2^1, e_3^1 e d_1^4. Portanto, podemos concluir que as arestas a, e e d incidem no vértice 1. Por sua vez, o segmento $\alpha_3 \to \cdots \to \alpha_1$ está associado ao vértice 3 e percorre as componentes de teste b_3^2, e_3^1 e c_4^3. Concluímos que as arestas b, e e c incidem no vértice 3.

O que acontece se o vértice 1 *for retirado da cobertura de* G*?* O «vértice de selecção» α_1 deixa de fazer parte de G'. *Nesse caso será possível construir um circuito hamiltoniano?*

Exercícios por Resolver

Exercício 5.5.2 *Mostre que* **P** *é fechado para uniões, intersecções e reduções polinomiais. Repita para* **NP**.

Exercício 5.5.3 *Mostre que se* f *for uma redução polinomial de* A *em* B *e* g *uma redução polinomial de* B *em* C*, então* $f \circ g$ *é uma redução polinomial de* A *em* C.

Exercício 5.5.4 *Mostre que se existir uma linguagem* **NP***-completa que pertença a* **P***, então* $\mathbf{P} = \mathbf{NP}$.

Exercício 5.5.5 *O* fecho de Kleene *de uma linguagem* L *é definido por*

$$L^* = \{w_1 \cdots w_k \mid k \geq 0 \wedge w_1, \ldots, w_k \in L\}.$$

Mostre que se $L \in \mathbf{P}$ *então* $L^* \in \mathbf{P}$. *Repita para* **NP** *(difícil!).*

Exercício 5.5.6 *O problema* $\mathcal{CAMG}$ *(caminho num grafo orientado) é definido por:* As instâncias são ternos (G, u, v) em que G é um grafo orientado e u, v são vértices de G. Uma instância é positiva se existir um caminho de u para v.

O problema $\mathcal{VALF}$ *(valor de uma fórmula) é definido por:* As instâncias são tuplos $(\phi, x_1, \ldots, x_n, y)$ em que ϕ é uma fórmula booleana com n variáveis (digamos, $\alpha_1, \ldots, \alpha_n$), $x_1, \ldots, x_n \in \{\top, \bot\}$ e $y \in \{0, 1\}$. Uma instância é positiva se, com a valoração $v : \alpha_i \mapsto x_i$, fica $v(\phi) = y$.

Mostre que existe uma redução polinomial de $\mathcal{CAMG}$ *para* $\mathcal{VALF}$.

Outras classes de complexidade

Apesar de **P** e **NP** serem as classes mais importantes no estudo da complexidade, existem muitas outras que são mais ou menos relevantes[11]. Esta pequena secção aborda alguns exemplos que podem dirigir o leitor na satisfação da curiosidade e no aprofundar do seu conhecimento desta área.

Um dos conceitos que não estudámos foi o de **complemento** de um problema. Por exemplo, dado um problema de decisão X, co-X é o problema correspondente após trocar o «sim» e o «não» do problema X (assim, o complemento do complemento é o problema original, co-co-$X = X$). Este conceito pode ser estendido às classes de complexidade: dada a classe Y, co-Y corresponde a todos os complementos dos problemas em Y (reparar que não corresponde ao complemento de Y em relação a todos os problemas de decisão que existem, algo muito maior e com problemas muito mais difíceis). Se uma classe contiver o seu próprio complemento, diz-se fechada para o complemento. Por exemplo, sabe-se que **P** = co-**P** mas não se sabe se **NP** = co-**NP** (isto significa que se **P** = **NP** então **NP** = co-**NP**). A factorização (dados inteiros a e b, saber se a tem um factor entre 1 e b) está simultaneamente em **NP** e co-**NP** (logo **NP** e co-**NP** não são conjuntos disjuntos)[12]. Esta classe, **NP** ∩ co-**NP**, é composta por problemas em que, qualquer argumento, ou tem um testemunho (polinomial) a confirmar a resposta «sim» ou tem um testemunho (polinomial) a confirmar a resposta «não». Deste modo, **P** ⊆ **NP** ∩ co-**NP**.

Existe outro conceito que surge nas obras sobre a complexidade computacional e referido pela palavra «difícil» (em inglês, «*hard*»). Dada uma classe de complexidade X, X-**difícil** (ou X-***hard***) corresponde, informalmente, aos problemas pelo menos tão difíceis como os mais difíceis de X. Mais formalmente, um problema Y é X-difícil se existir um

[11] Uma página com imensa informação sobre classes de complexidade é o *Complexity Zoo* em `http://qwiki.stanford.edu/wiki/Complexity_Zoo`.

[12] Sabe-se, desde 2002, que o problema de determinar se um número é primo pertence a **P**.

problema Y_1 que é X-completo e existir uma redução (polinomial) de Y_1 para Y. Por exemplo, um problema **NP**-difícil ou é **NP**-completo ou existe uma redução de um problema **NP**-completo para ele próprio. Isto significa que **NP**-completo $\subseteq$ **NP**-difícil. Se a versão sim/não de um problema de optimização é **NP**-completa, então a optimização do problema é **NP**-difícil, porque não existe um algoritmo rápido para determinar se um testemunho é o mais pequeno possível. Por exemplo, a cobertura de vértices, $\mathcal{COBV}$, dados (G, k) é **NP**-completa, como vimos anteriormente. Saber qual é a cobertura mínima (qual o menor k) para o grafo G é **NP**-difícil.

Apesar de nos termos focado na complexidade temporal (o número de passos de uma computação) referimos a complexidade espacial (a memória necessária para executar a computação). Não é de surpreender que existam classes que reflictam este conceito. Entre as mais relevantes (assumindo que os argumentos do problema são representados por n bits):

- **L** – corresponde ao conjunto de problemas cujo decisor necessita apenas de uma quantidade logarítmica de memória em relação a n (exceptuando os n bits necessários para armazenar o próprio argumento). Problemas nesta classe precisam apenas de uma muito modesta quantidade de memória para serem resolvidos. Um decisor com complexidade espacial $\mathcal{O}(\log(n))$ tem $2^{\mathcal{O}(log(n))} = n^{\mathcal{O}(1)}$ configurações possíveis, o que implica que os problemas em **L** estão obrigatoriamente em **P**, logo **L** $\subseteq$ **P**. Existe igualmente a classe **NL** que corresponde aos problemas que têm um verificador que necessita de uma quantidade logarítmica de memória. Entre alguns exemplos, o problema $2\mathcal{SAT}$ é **NL**-completo bem como o problema de, dado um grafo orientado G, saber se é possível ir de um vértice A até um outro B. De referir ainda que as questões **L** = **NL** e **L** = **P** ainda não foram resolvidas. Não saber que **L** = **P** significa que não está provado que não existam métodos muito eficientes, em termos de memória, para resolver qualquer problema polinomial (apesar dos estudiosos acreditarem, um pouco como na questão **P** = **NP**, que esta igualdade é extremamente improvável).

- **PSPACE** – corresponde aos problemas cujo decisor precisa de uma quantidade polinomial de memória. A restrição de memória nada exige sobre o tempo da computação, o que significa que, à partida, esta classe deverá ser mais ampla que **P**. E assim é, pois é possível realizar um programa que gera todas as combinações de candidatos a testemunhos usando sempre o mesmo espaço de memória, e para cada tentativa, testar se esta verifica o problema (se todos os candidatos falharem, não existe testemunho de dimensão polinomial). Como a memória disponível é polinomial, **PSPACE** inclui todos os problemas de **NP** (cujos testemunhos, pela definição, são de tamanho polinomial), ou seja, **NP** $\subseteq$ **PSPACE**. Esta classe é realmente muito grande, incluindo até problemas de decisão relativos a jogos de tabuleiro, como decidir o vencedor no Othello, no Hex ou no 5-em-linha (para ser exacto, são problemas **PSPACE**-completos).

- **EXPSPACE** – nestes problemas, o decisor necessita de uma quantidade exponencial de memória em relação à dimensão inicial dos argumentos. Esta classe é estritamente mais ampla que as classes **P**, **NP** e **PSPACE**, i.e., **P** $\subset$ **EXPSPACE**, **NP** $\subset$ **EXPSPACE**, **PSPACE** $\subset$ **EXPSPACE**. A decisão de vitória de vários jogos, como as Damas, o Xadrez ou o Go, são problemas **EXPSPACE**-completos.

Em resumo, **L** $\subseteq$ **NL** $\subseteq$ **P** $\subseteq$ **NP** $\subseteq$ **PSPACE** $\subset$ **EXPSPACE**.

Por fim, uma referência ao paralelismo e à classe **P-completa**. Um problema X é **P**-completo se estiver em **P** e se, para qualquer outro problema X_1 de **P**, existe uma redução em espaço logarítmico para X. O problema mais conhecido que é **P**-completo é a avaliação de um circuito lógico (ou de uma fórmula booleana, $\mathcal{VALF}$): dado um circuito lógico (composto por portas $\wedge$ e $\vee$) e uma valoração das entradas, determinar o resultado da saída.

Uma classe relacionada é **NC**, que corresponde aos problemas resolvidos por algoritmos em tempo $\mathcal{O}\left(\mathsf{log}\,(n)^k\right)$ usando $\mathcal{O}\left(n^k\right)$ processadores.

Se **P** é considerada a classe dos problemas acessíveis num computador sequencial, a classe **NC** representa os problemas resolvidos eficientemente num computador paralelo. Como uma máquina paralela pode ser simulada numa sequencial, $\mathbf{NC} \subseteq \mathbf{P}$ mas não se sabe se $\mathbf{NC} = \mathbf{P}$ embora os estudiosos acharem que esta igualdade é provavelmente falsa. A implicação é que nem todos os problemas de complexidade polinomial têm uma solução paralela eficiente. Como os problemas **P**-completos são os mais difíceis na classe **P**, a classe **P**-completa é considerada como a classe dos problemas para os quais não é possível obter uma versão paralela «eficiente» (num número polinomial de processadores) sendo, por isso, intrinsecamente sequenciais.

Anexo A

Revisões

A.1 Matemática

Conjuntos. A matemática assenta na noção de **conjunto**, que não se define, mas é regida pelas seguintes regras intuitivas (e mais algumas, apenas de importância técnica):

1. *Os conjuntos têm elementos*: $x \in A$ significa que x é **elemento** de A. Também se escreve $y \notin A$ para indicar que y *não* é elemento de A;
2. *Os conjuntos contêm subconjuntos*: $B \subset A$ lê-se «B está **contido** em A» e significa que todos os elementos de B também são elementos de A. Em particular, $A \subset A$;
3. *Existe um conjunto* vazio: $\emptyset$ é o conjunto sem elementos. Isto é, para qualquer objecto x, $x \in \emptyset$ é sempre falso e, para qualquer conjunto A, $\emptyset \subset A$ é sempre verdade;
4. *Dois conjuntos são **iguais** se, e só se, têm os mesmos elementos*: o conjunto A é igual ao conjunto B, $A = B$, se todo o elemento $b \in B$ também é elemento de A e, vice-versa, se todo o elemento $a \in A$ também $a \in B$; Outra forma de afirmar que A e B são o mesmo conjunto é verificando que $A \subset B$ e $B \subset A$;

A partir de alguns conjuntos é possível formarem-se outros, através das operações

intersecção $A \cap B$ é o conjunto dos elementos que estão em A e também estão em B. Se $x \in A \cap B$ então $x \in A$ e, também, $x \in B$;

união $A \cup B$ é o conjunto dos elementos que estão em A ou em B ou em ambos. Se $x \in A \cup B$ então $x \in A$ ou $x \in B$ (x pode estar num dos dois conjuntos, ou até em ambos, mas não se sabe em qual);

produto $A \times B$ é o conjunto dos **pares ordenados** de A e B. Os elementos de $A \times B$ são da forma (x, y) em que $x \in A$ e $y \in B$;

potência A^n é a n-ésima potência de A. Os elementos de A^n são **sequências**, todas de igual *dimensão*, n, da forma $(a_1, \ldots, a_n)$, em que cada $a_i \in A$. Em particular, $A^0 = \{()\}$ (um conjunto com uma única sequência, de dimensão zero), $A^1 = A$, $A^2 = A \times A$ e, em geral, $A^{n+1} = A^n \times A$.

fecho (1) $A^+ = \bigcup_{n>0} A^n$ é o conjunto de todas as sequências finitas e não vazias de elementos de A. Os elementos de A^+ são sequências, de dimensões distintas, mas maiores que 0.

fecho (2) $A^* = \bigcup_{n\geq 0} A^n$ é o conjunto de todas as sequências finitas de elementos de A. Podemos também definir $A^* = \{()\} \cup A^+$;

conjunto das partes $\mathcal{P}(A)$ é o conjunto de todos os subconjuntos de A. Isto é, $x \in \mathcal{P}(A)$ significa que $x \subset A$;

complemento $A \setminus B$ é o conjunto dos elementos de A que não estão em B. Isto é, $x \in A \setminus B$ quer dizer que $x \in A$ mas $x \notin B$.

Exercício A.1.1 *Seja U um conjunto não vazio e A, B subconjuntos de U. Mostre que*

1. $U \setminus (A \cup B) = (U \setminus A) \cap (U \setminus B)$;
2. $U \setminus (U \setminus A) = A$;
3. $U = A \cup (U \setminus A)$;
4. $A^* \subset U^*$;
5. *para cada* $n \in \mathbb{N}$, $A^n \subset U^n$;

6. $A \times B \subset U^2$.

Relações. As relações são a ferramenta matemática usada para distinguir certos elementos dentro de um conjunto: uma **relação** R em A é um subconjunto de A, $R \subset A$. O caso particular em que $A = B \times C$ é especialmente importante. Se $R \subset B \times C$ diz-se que R é uma **relação binária** e se $(b, c) \in R$ escreve-se bRc ou $R\,(b, c)$ e diz-se que «b está em relação (R) com c». Mais em particular, também nos interessa quando $B = C$. Seja então $R \subset B \times B$ uma relação binária em B. Dizemos que R é uma relação

reflexiva quando para cada $x \in B$, xRx;
simétrica quando para cada $x, y \in B$, se xRy então yRx;
anti-simétrica quando para cada $x, y \in B$, se xRy e yRx então $x = y$;
transitiva quando para cada $x, y, z \in B$, se xRy e yRz então xRz;
de equivalência se for reflexiva, simétrica e transitiva;
de ordem se for reflexiva, transitiva e anti-simétrica.

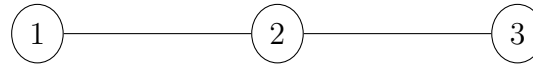
Grafos. Os grafos ilustram casos particulares de relações em conjuntos finitos. Um **grafo** G é definido por um conjunto de **vértices** (ou **nós**) V e uma relação simétrica em V, as **arestas** (ou **arcos**) A. Normalmente são ignorados os **lacetes**, isto é arestas da forma (x, x). Portanto o conjunto das arestas é, por norma, subconjunto de

$$V'' = \{(x, y) \mid x, y \in V, x \neq y\}$$

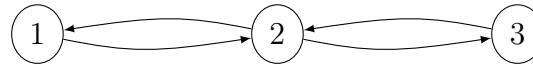
Os grafos podem ser representados esquematicamente por diagramas. Por exemplo, se

$$G_1 = (\{1, 2, 3\}, \{(1, 2), (2, 1), (3, 2), (2, 3)\})$$

os vértices são $V = \{1, 2, 3\}$ e as arestas $A = \{(1, 2), (2, 1), (3, 2), (2, 3)\}$ (note-se que A define uma relação simétrica). Este grafo pode ser representado por



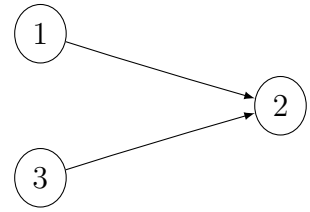
Outra outra forma de representar G_1 é usando setas, em vez de linhas,



o que corresponde a considerar cada par $(i, j) \in A$ como uma aresta **orientada**. Nesse caso ignora-se a condição de simetria nas arestas e diz-se que o grafo está **orientado**, ou que é um **digrafo**. Por exemplo,

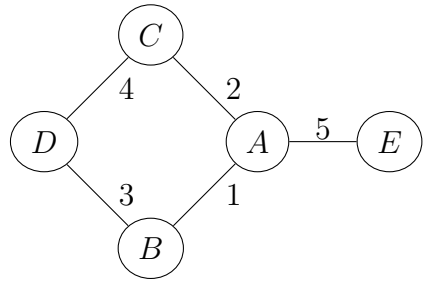
$$D_1 = (\{1, 2, 3\}, \{(1, 2), (3, 2)\})$$

pode ser representado por

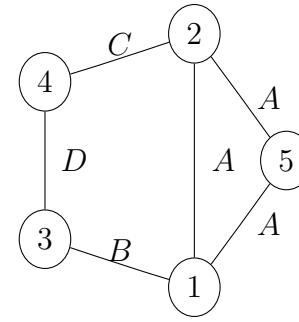


Podemos definir alguns conceitos interessantes sobre grafos. Seja $G = (V, A)$ um grafo. Então:

o grafo dual $G' = (V', A')$ do grafo $G = (V, A)$ define-se por $V' = A$ e $(x, y) \in A'$ se, e só se existe um vértice comum às arestas x e y em G;



um grafo G



e o seu dual G'

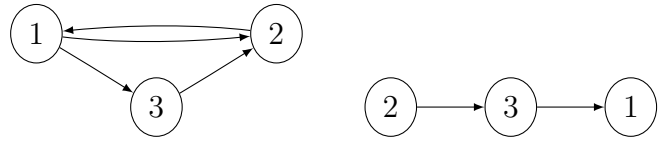
o grafo complementar é $G^c = (V, V'' \setminus A)$. Por exemplo, se

$$G = (\{1,2,3\}, \{(1,2),(2,1),(3,2),(1,3)\})$$

então

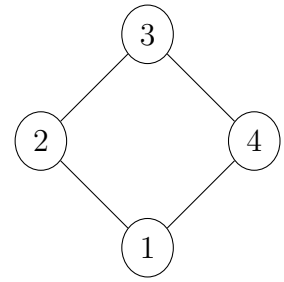
$$G^c = (\{1,2,3\}, \{(3,1),(2,3)\})$$

ou, graficamente:



um grafo G e o seu complementar G^c

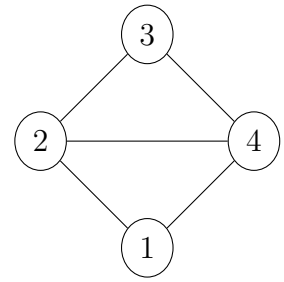
uma cobertura de vértices é um subconjunto de vértices $U \subseteq V$ tal que qualquer aresta de G tem, pelo menos, um vértice em U. Por exemplo:



1. os conjuntos $\{1,3\}$ e $\{2,4\}$ são coberturas de vértices, de tamanho dois;
2. no entanto $\{1,2\}$ não cobre todas as arestas ($3-4$ não está coberta);
3. qualquer subconjunto de três vértices é uma cobertura de G; nenhum subconjunto de um vértice é uma cobertura de G;

um conjunto independente é um subconjunto de vértices $I \subseteq V$ tal que nenhum par $i, j \in I$ forma uma aresta de G (**exercício:** determine todos os conjuntos independentes do grafo anterior.);

um clique é um subconjunto de vértices $C \subseteq V$ se para cada par de vértices $a, b \in C$, $(a, b) \in A$;



O conjunto $\{1, 2, 4\}$ é um clique: $(1, 2)$, $(1, 4)$ e $(2, 4)$ são arestas. Mas $\{1, 2, 3\}$ não, porque $(1, 3)$ não é uma aresta.

Teorema A.1.2

Seja $G = (V, A)$ um grafo, $U \subseteq V$. As seguintes afirmações são equivalentes:

1. *U é uma cobertura de vértices de G;*
2. *$V \setminus U$ é um conjunto independente de G;*
3. *$V \setminus U$ é um clique de G^c;*

Demonstração. Vamos demonstrar a cadeia de implicações $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$:

Para mostrar que $1 \Rightarrow 2$, sejam $a, b \in V \setminus U$. Supondo que $(a, b) \in A$, como U é uma cobertura de vértices, um destes dois vértices está em U, o que é uma contradição;

Vejamos agora que $2 \Rightarrow 3$. Se $a, b \in V \setminus U$ então $(a, b) \notin A$ portanto $(a, b) \in A^c$;

Finalmente provamos que $3 \Rightarrow 1$. Seja $(a, b) \in A$ uma aresta de G. Então ou a ou b não está em $V \setminus U$. Portanto $U \cap \{a, b\} \neq \emptyset$ □

Outras noções importantes sobre grafos envolvem a noção de **caminho**: uma sequência $v_1, \ldots, v_n$ de vértices tal que $(v_i, v_{i+1}), i = 1, \ldots, n-1$ é uma aresta. Se $v_1 = v_n$, diz-se que o caminho é **fechado** ou que é um **ciclo**. Um **caminho hamiltoniano** passa exactamente

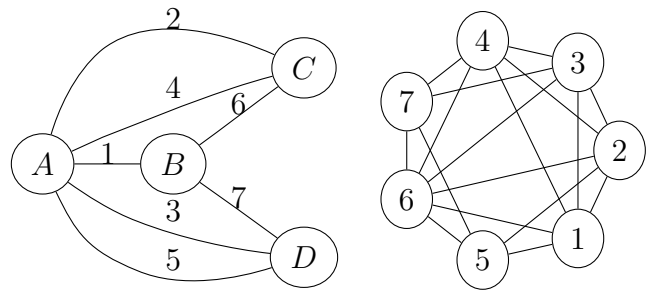


Figura A.1: Um exemplo de um grafo que não tem um **caminho hamiltoniano** é inspirado nas **sete pontes de *Königsberg***: na figura da esquerda, os vértices representam as margens e ilhas num rio e as arestas pontes. *Será que algum caminho passa por todas as pontes apenas uma vez?* Note-se que o grafo ilustrado tem um caminho hamiltoniano (por exemplo, A, C, B, D) mas o problema está enunciado sobre as arestas! De facto, o problema refere-se ao grafo **dual** (ilustrado à direita).

uma vez por cada vértice; se o caminho for fechado, temos um **ciclo hamiltoniano**.

Um dos problemas mais conhecidos sobre grafos envolve precisamente a noção de caminho (ou ciclo) hamiltoniano, o **problema do caixeiro viajante**, que pode ser enunciado da seguinte forma: *Dado um grafo em que os vértices representam as cidades de uma certa região, e as arestas estradas que ligam estas cidades, juntamente com informação sobre os comprimentos das estradas, pretende-se descobrir qual é o percurso mais curto que passa por todas as cidades.*

Funções. As funções associam elementos de um conjunto a elementos de outro conjunto. Uma **função** $f : A \to B$ é uma relação $f \subset A \times B$ em que cada elemento $a \in A$ está relacionado (por f) *quanto muito* com um elemento $y \in B$. Nesse caso escreve-se $y = f(x)$ ou $x \mapsto y$ por f (em vez de se escrever xfy ou $(x, y) \in f$, como nas relações) e diz-se que y é **imagem** (por f) de x. Além disso, A é **conjunto de partida** e B o **conjunto de chegada** de f.

Figura A.2: **Leonhard Euler** (*n.* 15 de Abril de 1707, *f.* 18 Setembro 1783) é um dos mais influentes e prolíferos matemáticos de sempre. Deve-se a ele a fórmula $e^{i\pi} + 1 = 0$ que relaciona não só os cinco números mais importantes como também exprime a essência da trigonometria. Fundou a teoria dos grafos, inspirado pelo problema das **sete pontes de Königsberg**.

Dizemos que $f : A \to B$ é uma função

sobrejectiva quanto cada elemento $y \in B$ é imagem de algum elemento $x \in A$;

injectiva quando a elementos diferentes de A correspondem imagens diferentes em B. Isto é, quando para cada $x_1, x_2 \in A$, se $x_1 \neq x_2$ então $f(x_1) \neq f(x_2)$ — ou, o que é equivalente, quando para cada $x_1, x_2 \in A$, se $f(x_1) = f(x_2)$ então $x_1 = x_2$;

bijectiva se for sobrejectiva e injectiva;

total quando cada elemento $x \in A$ tem imagem $f(x) \in B$;

parcial quando não é total.

O subconjunto dos elementos de A que têm imagem chama-se o **domínio** de f e representa-se por $\mathsf{dom}\,(f)$: $x \in \mathsf{dom}\,(f)$ se, e só se $f\,(x) \in B$. Por outro lado, o subconjunto das imagens em B chama-se **imagem** de f e representa-se por $\mathsf{im}\,(f)$: $y \in \mathsf{im}\,(f)$ se, e só se existe algum $x \in \mathsf{dom}\,(f)$ tal que $y = f\,(x)$.

Uma operação entre funções especialmente importante é a **composição**. Sejam $f : A \to B$ e $g : B \to C$ duas funções. Se $\mathsf{im}\,(f) \subset \mathsf{dom}\,(g)$ então pode-se definir uma nova função $g \circ f$ pela *composição* de f com g:

$$\begin{array}{rccc} g \circ f : & A & \to & C \\ & x & \mapsto & g\,(f\,(x)) \end{array} \tag{A.1}$$

Vejamos agora algumas noções sobre funções frequentemente usadas:

- A **identidade** em A é a função $f : A \to A, x \mapsto x$. Normalmente esta função representa-se por id_A;

- Seja $f : A \to B$ uma função qualquer. Se existir uma outra função $g : B \to A$ tal que

  1. $f \circ g = \mathsf{id}_B$, diz-se que g é a **inversa direita** de f;
  2. $g \circ f = \mathsf{id}_A$, diz-se que g é a **inversa esquerda** de f;
  3. seja inversa direita e esquerda, diz-se que g é a **inversa** de f e escreve-se $g = f^{-1}$.

- As **operações** são um caso particular das funções. Formalmente, dado um conjunto A, uma operação n-ária em A é uma função $f : A^n \to A$.

A soma e a multiplicação usuais são dois casos comuns de operações. Por exemplo, a soma de números reais é a operação

$$\begin{array}{rccc} \mathsf{soma} : & \mathbb{R}^2 & \to & \mathbb{R} \\ & (x, y) & \mapsto & x + y \end{array} .$$

É necessária alguma cautela com certas «operações» comuns: a divisão de números reais *não* é uma operação, pois não se pode definir

$$\begin{array}{rccc} \mathsf{divisão}: & \mathbb{R}^2 & \to & \mathbb{R} \\ & (x,y) & \mapsto & \frac{x}{y} \end{array}$$

uma vez que $\frac{x}{y}$ *não está definido* quando $y = 0$. Este problema pode ser parcialmente remediado fazendo uma operação em $\mathbb{R} \setminus \{0\}$,

$$\begin{array}{rccc} \mathsf{divisão}: & (\mathbb{R} \setminus \{0\})^2 & \to & \mathbb{R} \setminus \{0\} \\ & (x,y) & \mapsto & \frac{x}{y} \end{array}$$

mas, neste caso, por exemplo $\frac{0}{3}$ não está contemplado por esta operação...

A.2 Lógica

Historicamente, a **lógica** dedicou-se ao estudo da noção que intuitivamente se designa por *verdade*. Em particular procurou descobrir *como é que se podem produzir afirmações verdadeiras*. Mais recentemente essa investigação mostrou que a escrita tem um papel importante na pesquisa da verdade. Para representar *rigorosamente* afirmações — de que se pretende inquirir a verdade — uma parte significativa da lógica é dedicada à forma *como devem ser escritas as proposições* — *i.e.* expressões formais que poderão ser verdadeiras ou falsas.

Proposições. Uma das noções elementares da lógica é a de **proposição**, *i.e.* uma expressão de que podemos dizer que é verdadeira, ou falsa — por exemplo, podemos dizer que «de noite todos os gatos são pardos» é verdade ou falso. Mas já não diremos que «(o) gato» é verdade, nem falso. A partir de proposições dadas podemos construir outras, mais complexas: sendo P, Q duas proposições, também são proposições a **conjunção** $P \wedge Q$; a **disjunção** $P \vee Q$; a **implicação** $P \Rightarrow Q$ e a **negação** $\neg P$. Os símbolos $\wedge, \vee, \Rightarrow, \neg$, usados para construir expressões lógicas, também representam as **operações lógicas** mais comuns.

Exercício A.2.1 *Diga se são proposições:*

1. $2 + 2$;
2. $2 \times 3 = 5$;
3. $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$;
4. $\sin^2 \alpha + \cos^2 \alpha$;
5. *«O primeiro ministro de Portugal é alentejano.»*;
6. *«O primeiro ministro de Portugal.»*;
7. *«O preço da gasolina é muito elevado.»*;
8. *«A gasolina vai custar 2 euros por litro.»*;
9. $2\pi - \int_0^\infty \frac{1}{x^2} dx$;
10. $2\pi - \int_0^\infty \frac{1}{x^2} dx > 0$;
11. $(2 + 2) \wedge 3$;
12. $((2 + 2) > 3) \wedge (0 = 1)$;
13. $\neg$*«Amanhã vai chover.»* $\Rightarrow$ *«Amanhã vai fazer Sol.»*;
14. *«Amanhã vai chover.»* $\vee$ *«Aquela cadeira.»*;

Importa saber calcular os possíveis valores de verdade/falso de fórmulas formadas pelas operações lógicas. Esse cálculo resulta da tabela seguinte, em que $\top, \bot$ abreviam, respectivamente «verdade» e «falso»:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$

(A.2)

Exercício A.2.2 *Calcular o valor verdade/falso de algumas proposições;*

Analogamente ao cálculo aritmético, onde usamos expressões como $3 \times x - 2 = 19$ para designar certas contas, ou equações, também no cálculo lógico temos operações e incógnitas.

Definição A.2.3 (Fórmulas booleanas e Valorações)
Usamos fórmulas booleanas e valorações para fazer o cálculo de expressões da lógica.

1. *Uma* ***fórmula booleana*** *é*
   (a) *uma* ***variável booleana*** $u_1, u_2, \ldots$ *ou*
   (b) $\bot$ (***falso***, 0, *ou* bottom) *ou* $\top$ (***verdade***, 1, *ou* top) *ou*
   (c) $(\neg a)$, *se* a *for uma fórmula booleana ou*
   (d) $(a \wedge b)$ *ou* $(a \vee b)$ *se* a *e* b *forem ambas fórmulas booleanas e*
   (e) *nada mais;*
2. *o* ***comprimento*** *de uma fórmula booleana* a, *representado por* $|a|$, *define-se indutivamente por:*
   (a) *se* a *é uma variável,* $|a| = 1$;
   (b) *se* $a = \neg b$, $|a| = |b| + 1$;
   (c) *se* $a = b \wedge c$ *ou* $a = b \vee c$, $|a| = |b| + |c|$
3. *Escreve-se* $a \Rightarrow b$ *para abreviar* $\neg a \vee b$;
4. *Se* u *for uma variável,* u *é uma* ***literal positiva*** *e* $\neg u$ *uma* ***literal negativa***;
5. *Se* a *for uma fórmula,* $\mathrm{Var}(a)$ *é o conjunto das variáveis que ocorrem em* a;
6. *Uma* ***valoração booleana*** *é uma função* v *que associa a cada variável um valor em* $\{0, 1\}$;
7. *Uma* ***extensão*** *de uma valoração* v *é uma função* $\hat{v}$ *definida para cada fórmula* a *por:*
   (a) *Se* a *for uma variável,* $\hat{v}(a) = v(a)$;
   (b) *Se* $a = \bot$, $\hat{v}(a) = \hat{v}(\bot) = 0$;
   (c) *Se* $a = \top$, $\hat{v}(a) = \hat{v}(\top) = 1$;
   (d) *Se* $a = \neg b$, $\hat{v}(a) = 1 - \hat{v}(b)$;
   (e) *Se* $a = (b \wedge c)$, $\hat{v}(a) = \mathsf{min}\{\hat{v}(b), \hat{v}(c)\}$;

(f) Se $a = (b \vee c)$, $\hat{v}(a) = \mathsf{max}\{\hat{v}(b), \hat{v}(c)\}$;

8. *Uma fórmula* a *é* ***satisfeita*** *por uma valoração* v*, ou* v *é um* modelo *de* a*, se* $\hat{v}(a) = 1$*, e escreve-se* $v \models a$*;*
9. *Uma fórmula* a *é* ***satisfazível*** *se existir uma valoração* v *tal que* $v \models a$*;*
10. *Uma* ***interpretação*** *de uma fórmula* a *é uma valoração* v *tal que, para cada variável* $u \notin \mathrm{Var}(a)$*,* $v(u) = 1$*;*
11. *Duas fórmulas* a *e* b *são* ***equivalentes*** *se, para qualquer valoração* v*,* $\hat{v}(a) = \hat{v}(b)$ *e, nesse caso, escreve-se* $a \equiv b$*;*

Na escrita corrente de fórmulas omitimos os parenteses, se daí não resultar uma leitura ambígua, dando a seguinte prioridade aos conectivos: $\neg, \wedge, \vee$*;*

Exemplo A.2.4 (valoração, interpretação, equivalência, *etc.*) *A função*

$$\begin{array}{rccc} v: & \{a,b\} & \to & \{0,1\} \\ & a & & 1 \\ & b & \mapsto & 0 \end{array}$$

que representamos abreviadamente por

$$v = \begin{pmatrix} a & b \\ 1 & 0 \end{pmatrix}$$

é uma valoração, definida para as variáveis booleanas a *e* b*;*

Enquanto não podemos escrever, por exemplo, $v(\neg a \vee b)$*, porque a fórmula* $\neg a \vee b$ *não está no domínio de* v*, já podemos usar a extensão de* v *para calcular*

$$\hat{v}(\neg a \vee b) = \mathsf{max}\{\hat{v}(\neg a), v(b)\} = \mathsf{max}\{1 - v(a), 0\} = \mathsf{max}\{0,0\} = 0;$$

Neste caso $v \not\models \neg a \vee b$*. Mas, por exemplo,* v *satisfaz* $a \wedge \neg b$ *porque*

$$\hat{v}(a \wedge \neg b) = \mathsf{min}\{v(a), 1 - v(b)\} = \mathsf{min}\{1,1\} = 1;$$

Supondo agora que estamos a trabalhar com as variáveis a, b, c, d*, a valoração*

$$u = \begin{pmatrix} a & b & c & d \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

é uma interpretação de $a \vee \neg b$ *e, também, de* $c \vee b$*. Mas já não é uma interpretação de* $c \vee a$ *porque* $b \notin \mathrm{Var}\,(c \vee a) = \{a, c\}$ *e* $u\,(b) = 0$*;*

Algumas equivalências de fórmulas booleanas:

$$\begin{array}{lll} & \neg\neg a \equiv a & \\ \neg\bot \equiv \top & a \wedge a \equiv a & a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c) \\ \neg\top \equiv \bot & a \vee a \equiv a & a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c) \\ \neg\,(a \vee b) \equiv \neg a \wedge \neg b & & \neg\,(a \wedge b) \equiv \neg a \vee \neg b \end{array}$$

As duas últimas equivalências são as ***leis de*** **de Morgan***;*

Por aplicação sucessiva destas equivalências, para cada fórmula a dada, consegue-se chegar a uma outra fórmula a' equivalente a a mas em que todas as negações são aplicadas apenas a variáveis ou às constantes $\bot$ e $\top$. Dizemos então que a' está na **forma normal**.

As cláusulas e os conjuntos finitos de cláusulas fornecem uma representação alternativa das fórmulas booleanas.

Definição A.2.5 (Conjuntos finitos de cláusulas)

1. *Uma* ***cláusula*** *é um conjunto finito de literais (positivas ou negativas);*

2. *Se* c *for uma cláusula e* $\hat{v}$ *uma extensão da valoração* v*,*

$$\hat{v}\,(c) = \begin{cases} 1 & \text{se existe } x \in c : \hat{v}\,(x) = 1 \\ 0 & \text{se para cada } x \in c, \hat{v}\,(x) = 0 \end{cases}$$

3. *Se* C *for um* ***conjunto finito de cláusulas*** *(abreviadamente:* ***cfc****),*

$$\hat{v}\,(C) = \begin{cases} 1 & \text{se para cada } c \in C, \hat{v}\,(x) = 1 \\ 0 & \text{se existe } c \in C : \hat{v}\,(c) = 0 \end{cases}$$

4. *Um conjunto finito de cláusulas é* ***satisfazível*** *se existir alguma (extensão de uma) valoração v tal que $\hat{v}(C) = 1$.*

A equivalência entre a satisfação de fórmulas e a satisfação de cfc's resulta do seguinte

Teorema A.2.6 (Forma normal conjuntiva) *Seja b uma fórmula e $\mathrm{Var}(b) = \{u_1, \ldots, u_n\}$ o conjunto das variáveis que ocorrem em b. Existe uma fórmula b', equivalente a b e tal que*

$$b' = \left(u_1^1 \vee \cdots \vee u_n^1\right) \wedge \cdots \wedge \left(u_1^m \vee \cdots \vee u_n^m\right)$$

onde cada u_i^j é u_i ou $\neg u_i$ ou não ocorre em b'.

Demonstração. Obtemos a forma normal conjuntiva de uma fórmula dada fazendo as seguintes substituições:

cada ocorrência de	por
$\neg(a \wedge b)$	$\neg a \vee \neg b$
$\neg(a \vee b)$	$\neg a \wedge \neg b$
$\neg\neg a$	a
$a \vee (b \wedge c)$	$(a \vee b) \wedge (a \vee c)$
$(a \wedge b) \vee c$	$(a \vee c) \wedge (b \vee c)$
$a \vee a$	a
$a \wedge a$	a

Cada uma destas substituições transforma a fórmula inicial noutra equivalente. Portanto, a fórmula final é equivalente à inicial. □

A fórmula a' do teorema anterior chama-se **uma forma normal conjuntiva** de a e diz-se que é uma **fórmula conjuntiva**. A equivalência entre a satisfação de fórmulas e de cfc's pode agora ser estabelecida de imediato pois a cada subfórmula $\left(u_1^j \vee \cdots \vee u_n^j\right)$ faz-se corresponder uma cláusula $c_j = \left\{u_1^j, \ldots, u_n^j\right\}$ e, à fórmula a', o cfc $C_a = \{c_1, \ldots, c_m\}$.

Isto é,

Teorema A.2.7 *Para cada fórmula booleana a existe um cfc C_a tal que a é satisfazível se, e só se, C_a é satisfazível.*

Demonstração. Dada uma fórmula a e a sua forma normal conjuntiva a', para se obter C_a basta percorrer a' e substituir cada conectivo $\wedge$ ou $\vee$ por uma vírgula e cada parêntese por uma chaveta, de acordo com a regra

$$\begin{array}{rcccc} \text{substituir} & \wedge & \vee & (&) \\ \text{por} & , & , & \{ & \} \end{array}$$

□

Exemplo A.2.8 *Usando a demonstração dos dois teoremas anteriores, temos*

$$\begin{aligned} a &= \neg\left(u_1 \wedge \left(\neg u_2 \vee u_1\right)\right) \\ &\equiv \neg u_1 \vee \neg\left(\neg u_2 \vee u_1\right) \\ &\equiv \neg u_1 \vee \left(\neg\neg u_2 \wedge \neg u_1\right) \\ &\equiv \neg u_1 \vee \left(u_2 \wedge \neg u_1\right) \\ &\equiv \left(\neg u_1 \vee u_2\right) \wedge \left(\neg u_1 \vee \neg u_1\right) \\ &\equiv \left(\neg u_1 \vee u_2\right) \wedge \left(\neg u_1\right) \\ &\mapsto C_a = \left\{\left\{\neg u_1, u_2\right\}, \left\{\neg u_1\right\}\right\} \end{aligned}$$

Predicados. A lógica das proposições não é suficientemente elaborada para representar muitas situações importantes. Por exemplo, não é possível analisar a proposição «de noite todos os gatos, excepto o Tareco, são pardos» sem ter em conta o que é o gato «Tareco», se o estado do mundo «é de noite», a propriedade «é pardo», *etc.* O problema é que nas proposições não é possível identificar-se *de quem* se está a falar, nem *o que* se está a afirmar! A solução para este problema consiste em *estruturar* as proposições, identificando *termos* (de quem se está a falar) e *predicados* (as qualidades desses termos).

Supomos que os **termos** pertencem a um certo conjunto não vazio, digamos A, a que chamamos **domínio**, onde escolhemos algumas relações, as **relações atómicas** (que podem ser unárias, binárias, *etc.*). Sendo a um termo, x uma variável e P uma relação (sem perda de generalidade, unária), são **predicados**: a **substituição** $P\left(a\right)$, as expressões

que obtemos pelas operações lógicas (por exemplo $P(a) \wedge \neg (Q(b) \Rightarrow R(a))$), a **quantificação universal** $\forall x\, P(x)$ e a **quantificação existencial** $\exists x\, P(x)$. Um predicado de uso muito comum é a **igualdade** $x = y$ (*i.e.* a relação binária $\{(a,a) \mid x \in A\} \subset A \times A$).

Exercício A.2.9 *Classifique as expressões seguintes como* predicado, termo *ou* erro*:*

1. $\forall x$ maior $(x, 0)$, $\exists x\, \forall y\, x = y$;
2. $\exists y$, $\forall\, \exists a$;
3. 12, $\sqrt{24}$, 2^x, $3 + 12$, $2 > 3$, $x \leq 1$;
4. gato, tareco *(supondo que é um gato)*, gato (tareco);
5. par (2), par (3), par (x), par;

De novo, é necessário indicar como é que os valores de verdade dos predicados mais simples influenciam os valores de verdade das fórmulas onde ocorrem:

1. $\forall x\, P(x)$ é verdade se e só se $P(a)$ é verdade para cada $a \in A$ (*i.e.* se $P = A$);
2. $\exists x\, P(x)$ é verdade se e só se $P(a)$ é verdade para algum $a \in A$ (*i.e.* se $P \neq \emptyset$);

Exemplo A.2.10 *O* domínio *é* $A = \{1,2,3,4,5,6,7,8,9\}$*, onde escolhemos as relações unárias «*é par*»:* par $= \{2,4,6,8\} \subset A$ *e «*é potência de 2*»*[1]*:* pot2 $= \{2,4,8\} \subset A$ *e a relação binária «*é metade de*»:* metade $= \{(1,2),(2,4),(3,6),(4,8)\} \subset A \times A$.

A partir destes predicados podemos definir outros, pela substituição, quantificação e operações lógicas:

1. ímpar $(x) \equiv \neg$par (x);
2. metade-par $(x) \equiv \exists y$ metade $(x, y) \wedge$ par (y);

[1] Se, neste exemplo, pretendêssemos ser aritmeticamente correctos teríamos de incluir $1 = 2^0$ em pot2.

3. $\mathsf{dobro}\,(x, y) \equiv \mathsf{metade}\,(y, x)$;
4. $\mathsf{ímpar\text{-}dobro}\,(x) \equiv \mathsf{ímpar}\,(x) \wedge \exists y\, \mathsf{dobro}\,(x, y)$

Como encontrar os valores de verdade/falso nalguns predicados:

1. $\mathsf{par}\,(2)$ *é verdade porque* $2 \in \mathsf{par}$, $\mathsf{par}\,(1)$ *é falso porque* $1 \notin \mathsf{par}$;
2. $\forall x\, \mathsf{pot2} \Rightarrow \mathsf{par}\,(x)$ *é verdade: consideremos todos os termos que a variável* x *pode representar* ($x = 1, x = 2, \ldots, x = 9$). *Vejamos, termo a termo, se o predicado* $\mathsf{pot2}\,(x) \Rightarrow \mathsf{par}\,(x)$ *é verdadeiro.*

   $x = 1$**:** $\mathsf{pot2}\,(1) \Rightarrow \mathsf{par}\,(1)$ *será verdade? Como* $\mathsf{pot2}\,(1)$ *é falso, a implicação* $\mathsf{pot2}\,(1) \Rightarrow \mathsf{par}\,(1)$ *é verdadeira;*
   $x = 2$**:** $\mathsf{pot2}\,(2) \Rightarrow \mathsf{par}\,(2)$ *será verdade? Como* $\mathsf{pot2}\,(2)$ *é verdade e* $\mathsf{par}\,(2)$ *também, a implicação* $\mathsf{pot2}\,(2) \Rightarrow \mathsf{par}\,(2)$ *é verdadeira;*
   $x = \cdots$**:** etc.

   Portanto, na fórmula $\mathsf{pot2}\,(x) \Rightarrow \mathsf{par}\,(x)$ *para qualquer possível substituição da variável* x *por qualquer um dos termos* $1, 2, \ldots, 9$, *resulta um predicado verdadeiro. Isto é,* $\mathsf{pot2}\,(a) \Rightarrow \mathsf{par}\,(a)$ *é verdadeiro para cada* $a \in A$;
3. $\exists x\, \mathsf{par}\,(x) \wedge \mathsf{pot2}\,(x)$ *é verdade: basta encontrar um termo* $1, 2, \ldots, 9$ *que, substituindo a variável* x *em* $\mathsf{par}\,(x) \wedge \mathsf{pot2}\,(x)$, *produza um predicado verdadeiro. Ora,* $\mathsf{par}\,(4) \wedge \mathsf{pot2}\,(4)$ *é verdadeiro.*

A.3 Aritmética

Números naturais, inteiros, racionais, reais e complexos. Os **números naturais** usam-se para contar: «tenho 5 dedos em cada mão»; «depois das obras fiquei sem dinheiro (tenho 0 euros)», *etc.* O conjunto dos números naturais representa-se por $\mathbb{N}$. Note-se que 0 é uma quantidade, logo é um número natural. Alguns autores consideram que os números naturais começam em 1, mas isso não é correcto.

As soluções de certos problemas são números naturais. Por exemplo «O café custa 1 euro. Que troco tenho de receber se paguei com uma

moeda de 2 euros?». Mas nem todos os problemas deste tipo encontram solução entre os números naturais: «Um caracol andou um metro para cima e deslizou dois para baixo. Quantos metros subiu esse caracol?». Este problema requer outra classe de números, os **números inteiros**, representada por $\mathbb{Z}$. Os números inteiros permitem resolver todas as equações (em x) da forma

$$x + a \;=\; b$$

para quaisquer valores (inteiros) de a e b. Entre os naturais seria necessário exigir que na equação anterior fosse $b \geq a$.

Mas essa equação é, ainda, muito simples. Uma forma imediata de a tornar mais geral é considerando a multiplicações no lado esquerdo:

$$x \times a + b = c$$

Certos casos particulares desta equação têm solução inteira:

$$-4 \times x = 2$$

$(a = -4, b = 0, c = 2)$ tem solução $x = -2$. Mas, de novo, algumas equações deste tipo necessitam de outros números, além dos inteiros:

$$x \times 2 = 1.$$

A nova classe de números, que resolve todas as equações do tipo $x \times a + b = c$, representa-se por $\mathbb{Q}$ e os seus elementos são os **números racionais** (do latim *ratio*, divisão). Formalmente, podemos escrever

$$\mathbb{Q} = \left\{ \frac{p}{q} \mid p, q \in \mathbb{Z} \wedge q \neq 0 \right\}$$

mas é importante lembrar que, desta forma, os números racionais podem ser escritos como várias fracções (de facto, cada número racional pode ser escrito como uma infinidade de fracções). Por exemplo,

$$\frac{6}{12} = \frac{2}{4} = \frac{3}{6} = \frac{1}{2} = \frac{50}{100} = \cdots$$

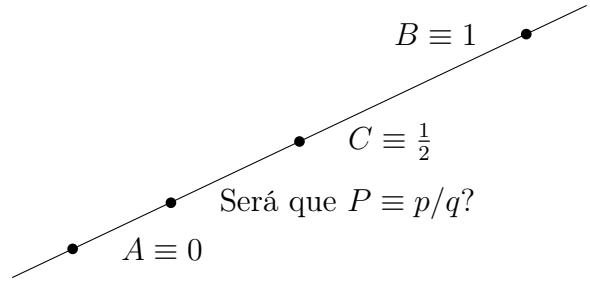
De todas estas formas de representar o mesmo número racional, uma em particular é especialmente importante: quando p e q não têm divisores comuns diz-se que a fracção está **reduzida**. Por exemplo o número racional 0.25 pode ser escrito como a fracção $\frac{25}{100}$ mas esta não é reduzida porque 25 e 100 têm factores comuns (5 e 25). Outra forma de representar $0,25$ é pela fracção reduzida $\frac{1}{4}$. Uma forma de reduzir uma fracção $\frac{p}{q}$ é dividindo o numerador p e o denominador q pelo maior factor comum. Assim, *qualquer número racional pode ser escrito como uma única fracção reduzida, com denominador positivo.* Exigimos o denominador positivo porque, por exemplo $0.25 = \frac{1}{4} = \frac{-1}{-4}$ tem duas fracções reduzidas, mas esta questão pode ser resolvida exigindo que o denominador seja sempre positivo, descartando a fracção $\frac{-1}{-4}$. Resumindo,

> Todo o número racional pode ser escrito de uma só forma como uma fracção reduzida com o denominador positivo.

Exercício A.3.1 *Escreva os seguintes números racionais como fracções reduzidas:*

$$\frac{6}{9}; \frac{-35}{14}; \frac{136}{-578}; \frac{-1024}{-512}; \frac{2^7}{3^7}; 0.312; 0.9999\cdots; 0.151515\cdots$$

A classe $\mathbb{Q}$ *parece preencher completamente toda a linha recta.* O que significa isto? Consideremos a recta $\overline{AB}$, que passa nos pontos A e B. Associemos ao ponto A o número 0 e ao ponto B o número 1. O ponto médio do segmento $[AB]$ vem então associado ao número racional $\frac{1}{2}$. Podemos perguntar: «será que cada ponto P da recta está associado a um número racional p/q?».



Os números racionais até têm uma propriedade que parece indicar que sim, que qualquer ponto da recta estará associado a um número racional. A **propriedade arquimediana** diz que «*dados dois números racionais diferentes, $a < b$, existe um terceiro número racional, c, entre ambos: $a < c < b$.*» Basta fazer $c = \frac{a+b}{2}$ e verificar que, de facto, $a < c$ e $c < b$ (claro, desde que $a < b$). Resulta desta propriedade que os números racionais formam um conjunto que parece «**denso**», sem buracos entre dois números, ao contrário do que acontecia com os inteiros. Entre 0 e 1 não existe nenhum número inteiro, mas, graças à propriedade arquimediana, existem infinitos números racionais. Mais, por muito próximos que estejam dois racionais, entre eles existem *infinitos* outros racionais.

Voltando à questão dos pontos da recta e dos números racionais, a propriedade arquimediana parece indicar que cada ponto está associado a algum número racional. Mas isso não é verdade. Na grécia clássica já se sabia que *o comprimento da diagonal do quadrado de lado* 1, $\sqrt{2}$, não é um número racional (diz a lenda que, na altura, esta «descoberta» foi um escândalo enorme, e alvo de censura. A existência de números **irracionais** — nos dois sentidos da palavra, números que não são nem fracções nem raciocináveis — era um segredo reservado a certas elites). Portanto os números racionais não chegam para esgotar as soluções de

$$ax^2 + bx + c = 0$$

pois $\sqrt{2}$ é uma solução do caso particular $x^2 - 2 = 0$. Para incluir os números que estão associados aos pontos das rectas é necessária uma nova classe de números, $\mathbb{R}$, cujos elementos são os **números reais**.

Mas a questão das soluções de equações não se esgota com os números reais. É que

$$x^2 + 1 = 0$$

não tem soluções entre os números reais. Estes «esgotam» os pontos das rectas, mas não chegam para as soluções de certas equações do 2º grau. Para esgotar todas as equações polinomiais, e os pontos das rectas, é preciso considerar um outro tipo de números, os **números complexos**, cujo conjunto é representado por $\mathbb{C}$ e pode ser formalmente definido por

$$\mathbb{C} = \{a + bi \mid a, b \in \mathbb{R}\}$$

sendo $i = \sqrt{-1}$ uma das soluções (existem duas, e a outra é $-i$) da equação $x^2 + 1 = 0$.

Certos subconjuntos dos naturais, inteiros e reais são muito comuns:

- $\mathbb{N}^+ = \mathbb{Z}^+$ é o conjunto dos **inteiros positivos**: $\{1, 2, 3, \ldots\}$. Para incluir o número 0 escreve-se $\mathbb{Z}_0^+$ (note-se que $\mathbb{Z}_0^+ = \mathbb{N}$);
- $\mathbb{R}^+$ é o conjunto dos **reais positivos**: $x \in \mathbb{R}^+$ se e só se $x \in \mathbb{R}$ e $x > 0$. Para incluir o número 0 escreve-se $\mathbb{R}_0^+$;

Números primos. Os números **primos** são os números inteiros positivos que têm exactamente dois divisores positivos. Por exemplo, 5 é primo porque os únicos divisores positivos de 5 são 1 e 5. Mas 8 já não é primo porque tem os seguintes divisores positivos: $1, 2, 4, 8$.

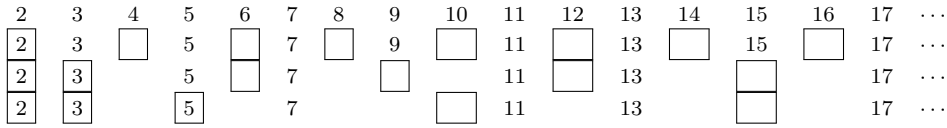


Figura A.3: O **filtro de Eratóstenes** serve para descobrir os números primos. Escrevem-se os números naturais a partir do 2: 2, 3, 4, 5, ... Assinala-se o 2 e, saltando de dois em dois, eliminam-se os números em que se cai: 4, 6, 8, 10, ... Depois, assinala-se o primeiro número que não foi eliminado. Neste caso, é o 3. Então, saltando de três em três, vão-se eliminado os números que restaram e em que se cai: 9, 15, 18, ... O próximo número que resta é o 5. Saltando de cinco em cinco, elimina-se o 25, 35, ... Seguindo este processo, os números que sobram, $2, 3, 5, 7, 11, \cdots$, são os primos.

Exercício A.3.2 *Qual é o primeiro número que se elimina quando se assinala o 5? Note-se que não é o 10, porque este foi eliminado quando se tinha assinalado o 2... Também não é o 15, porque 15 foi eliminado na «vez» do 3. Agora repita esta questão substituindo 5 por 7, 10 por 14 e 15 por 21. Encontra algum padrão? Consegue formular uma regra geral?*

Uma das propriedades mais importantes dos números primos é que

Teorema A.3.3 *Cada número natural n pode ser factorizado (de forma «única») como produto de potências de números primos.*

Vejamos a factorização dos primeiros inteiros positivos:

$$\begin{array}{rclrcl} 1 &=& 2^0 3^0 5^0 7^0 11^0 & 7 &=& 2^0 3^0 5^0 7^1 11^0 \\ 2 &=& 2^1 3^0 5^0 7^0 11^0 & 8 &=& 2^3 3^0 5^0 7^0 11^0 \\ 3 &=& 2^0 3^1 5^0 7^0 11^0 & 9 &=& 2^0 3^2 5^0 7^0 11^0 \\ 4 &=& 2^2 3^0 5^0 7^0 11^0 & 10 &=& 2^1 3^0 5^1 7^0 11^0 \\ 5 &=& 2^0 3^0 5^1 7^0 11^0 & 11 &=& 2^0 3^0 5^0 7^0 11^1 \\ 6 &=& 2^1 3^1 5^0 7^0 11^0 & 12 &=& 2^2 3^1 5^0 7^0 11^0 \end{array}$$

Exercício A.3.4 *Factorize* $43, 67, 89, 102, 135, 144$*;*

Os números primos são infinitos. Escrevemos $p_1, p_2, \ldots$ para representar a sucessão dos números primos. Por exemplo

$$p_1 = 2, p_2 = 3, p_3 = 5, \ldots, p_6 = 13, \ldots$$

Não existe nenhuma fórmula «normal» (no sentido em que, por exemplo, a fórmula $h^2 = a^2 + b^2$ é «normal») para descrever a sucessão p_n. Usando a sucessão dos números primos, temos a **expansão prima** de um inteiro. Por exemplo,

$$43560 = 2^3 \times 3^2 \times 5 \times 11^2 = p_1^3\, p_2^2\, p_3^1\, p_4^0\, p_5^2.$$

Os expoentes nestes produtos de primos podem ser obtidos com a notação «$(x)_n$», que indica qual é *o expoente do n-ésimo primo na factorização de* x ou seja, o número de vezes que o primo p_n divide x. Usando

o exemplo anterior,

$$\begin{aligned}(43560)_1 &= 3\\ (43560)_2 &= 2\\ (43560)_3 &= 1\\ (43560)_4 &= 0\\ (43560)_5 &= 2\\ (43560)_6 &= 0\\ &\vdots\end{aligned}$$

Exercício A.3.5 *Calcule*

1. $(145)_1, (145)_2, (145)_2, (145)_3, (145)_4, (145)_5, (145)_{123}$;
2. $(325)_1, (325)_2, (325)_2, (325)_3, (325)_4, (325)_5, (325)_{123}$;
3. $(1024)_1, (1024)_2, (1024)_2, (1024)_3, (1024)_4, (1024)_5, (1024)_{123}$;
4. $(2000)_1, (2000)_2, (2000)_2, (2000)_3, (2000)_4, (2000)_5, (2000)_{123}$;

Bases numéricas e Expansões. Quando escrevemos um número inteiro, por exemplo «1432», coexistem dois aspectos distintos, o número em si (mil quatrocentos e trinta e dois) e o **numeral**, *i.e.*, a sequência de símbolos que representam esse número (o texto «1432»). O nosso hábito é utilizar o **sistema de numeração árabe** (que «conquistou» a matemática nos últimos séculos pela sua flexibilidade) mas poderíamos escolher outras notações:

romana MCDXXXII

português mil quatrocentos e trinta e dois

binário 10110011000

unário $\underbrace{\bullet\bullet\bullet\ldots\bullet\bullet}_{1432 \text{ contas}}$

Existe uma grande diferença entre os sistemas **unários** e **posicionais**. Nos sistemas unários, cada número é representado por uma extensa sequência de contas iguais, e usam-se abreviaturas para sequências de contas, como na notação romana onde «M» abrevia mil contas. Estes sistemas tornam-se demasiado complicados para representar números grandes ou números que não sejam inteiros.

Os **sistemas posicionais** utilizam um método diferente: o numeral é constituído por um conjunto de símbolos em que *a posição relativa de cada símbolo determina o seu valor para o cálculo final do valor*. Normalmente, cada posição representa um acréscimo baseado em potências de um número, a **base**.

O sistema posicional que aprendemos na escola tem base dez (daí o nome de sistema decimal) e utiliza dez símbolos distintos (0, 1, 2, ..., 8, 9). Quando se escreve «1432» informamos que o número representado por este numeral é o resultado da expressão $1\times1000+4\times100+3\times10+2$, ou seja

$$1 \times 10^3 + 4 \times 10^2 + 3 \times 10^1 + 2 \times 10^0.$$

Uma das consequências do uso deste sistema é que tem de existir um termo que indique quando uma posição não contribui para o cálculo do número. Por outras palavras, na escrita posicional é necessário um símbolo para o zero.

Mas como é que «mudamos a base»? Se mudarmos para uma base arbitrária $b > 1$, necessitamos de b símbolos e temos de refazer a expressão que calcula o número. Quando $b \leq 10$ usamos como símbolos os dígitos de «0» a «$b-1$» e quando $b > 10$ usam-se os dígitos de «0» a «9» juntamente com as letras do alfabeto (*i.e.*, «A», «B», «C»...). Para indicarmos que estamos a usar a base b para escrever o numeral n escrevemos n_b. Como fazemos habitualmente, quando estamos a escrever numerais, se não indicarmos a base (com o índice «$_b$»), é porque estamos usar a base dez: $13552 = 13552_{10}$.

Vejamos algumas representações de mil quatrocentos e trinta e dois:

- Em base 12: $1432 = 9B4_{12} = 9 \times 12^2 + 11 \times 12 + 4$;
- Em base 11: $1432 = 1092_{11} = 1 \times 11^3 + 0 \times 11^2 + 9 \times 11 + 2$;

- Em base 9: $1432 = 1861_9 = 1 \times 9^3 + 8 \times 9^2 + 6 \times 9 + 1$;
- Em base 8: $1432 = 2630_8 = 2 \times 8^3 + 6 \times 8^2 + 3 \times 8 + 0$;
- Em base 7: $1432 = 4114_7 = 4 \times 7^3 + 1 \times 7^2 + 1 \times 7 + 4$;
- Em base 6: $1432 = 10344_6 = 1 \times 6^4 + 0 \times 6^3 + 3 \times 6^2 + 4 \times 6 + 4$;
- Em base 5: $1432 = 21212_5 = 2 \times 5^4 + 1 \times 5^3 + 2 \times 5^1 + 1 \times 5 + 2$.

Exercício A.3.6 *Como é o numeral de n em base n? E de n^2? E de n^k?*

Na ciência da computação, a **base binária** ($b = 2$) é amplamente usada (que se reflecte na arquitectura dos nossos computadores), bem como a base **octal**($b = 8$) e a **hexadecimal** ($b = 16$).

Como calcular a representação de n numa dada base b? É possível obtê-la por sucessivas divisões inteiras, de acordo com o seguinte algoritmo:

```
enquanto n ≠ 0 :
    q ← cociente da divisão inteira n/b
    r ← resto da divisão inteira n/b
    numeral ← r-ésimo dígito ⊕ numeral
    n ← q
retorna numeral
```
(A.3)

onde *numeral* é uma sequência de símbolos (que começa vazia) e o símbolo $\oplus$ representa a operação que junta o primeiro argumento à esquerda do segundo.

Para converter 1432 na base $b = 9$:

$1432/9$	$q = 159$	$r = 1$	$numeral = 1$
$159/9$	$q = 17$	$r = 6$	$numeral = 61$
$17/9$	$q = 1$	$r = 8$	$numeral = 861$
$1/9$	$q = 0$	$r = 1$	$numeral = 1861$

São possíveis outros sistemas posicionais sem uma base fixa, por exemplo, a **expansão prima** de n, como vimos anteriormente, na parte

sobre números primos. Por exemplo, $13552 = 2^4 \times 7^1 \times 11^2$. Podemos escrever $13552 = 2^4 \times 3^0 \times 5^0 \times 7^1 \times 11^2 \times 13^0 \times 17^0 \ldots$ Exprimindo somente os expoentes do primeiro primo até ao maior expoente diferente de zero, obtém-se a lista $[4, 0, 0, 1, 2]$, que é a expansão prima de 13552.

Demonstrações por absurdo. Uma das técnicas mais elegantes de provar resultados consiste na chamada **demonstração por absurdo**. A ideia é a seguinte: supondo que sabemos que $A \Rightarrow B$. Se, por outro lado, chegarmos à conclusão que B é falso, temos de concluir que também A é falso. Porquê? Porque, se A fosse verdade, como $A \Rightarrow B$, também B teria de ser verdade. Mas sabemos que B é falso. Assim, A não pode ser verdade, portanto A é falso.

Vejamos algumas aplicações desta técnica:

Teorema A.3.7 *Se n^2 for um número natural par, então n também é um número natural par.*

Demonstração. Vamos supor que n^2 é par mas n é ímpar. Se n for ímpar, então existe um certo número inteiro k tal que $n = 2k + 1$. Mas então

$$\begin{aligned} n^2 &= (2k+1)^2 \\ &= (2k+1)(2k+1) \\ &= 2k(2k+1) + (2k+1) \\ &= (2k \times 2k + 2k) + 2k + 1 \\ &= 4k^2 + 2k + 2k + 1 \\ &= 4k^2 + 4k + 1 \end{aligned}$$

Agora, consideremos a igualdade que obtemos:

$$n^2 = 4k^2 + 4k + 1.$$

Do lado esquerdo temos n^2, que sabemos que é um número par. Do lado direito temos $4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$ que tem de ser ímpar. Isto é, n^2 teria de ser, simultaneamente, par e ímpar. Mas isso é impossível!

Esta impossibilidade resultou de termos suposto que n poderia ser ímpar. Como n não pode ser ímpar, tem de ser par. □

Para vermos uma aplicação mais profunda das demonstrações por absurdo, vamos provar que $\sqrt{2}$ não é um número racional. Antes de fazermos essa demonstração, lembremos que os números racionais podem ser escritos como fracções $\frac{p}{q}$ em que $p, q \in \mathbb{Z}, q \neq 0$.

Vejamos agora o que tanto preocupou os gregos da escola de Pitágoras:

Teorema A.3.8 *A raiz quadrada de* 2 *não é um número racional.*

Demonstração. Vamos supor que $\sqrt{2}$ *é um número racional.* Então $\sqrt{2}$ pode ser escrito como uma fracção reduzida (com denominador positivo):

$$\sqrt{2} = \frac{p}{q}$$

Mas $\sqrt{2}$ é solução de $x^2 - 2 = 0$, ou seja, é solução de $x^2 = 2$ e portanto $\frac{p}{q}$ verifica a igualdade

$$\left(\frac{p}{q}\right)^2 = 2$$

Desenvolvendo o quadrado do lado esquerdo, ficamos com

$$\frac{p^2}{q^2} = 2$$

logo

$$p^2 = 2q^2 \tag{A.4}$$

Mas esta igualdade implica que p^2 é um número par. Portanto (pelo teorema A.3.7), também p *tem de ser um número par*, isto é, $p = 2r$. Substituindo p por $2r$ na equação A.4, ficamos com

$$(2r)^2 = 2q^2$$

Agora podemos desenvolver o lado esquerdo e obtemos a igualdade

$$4r^2 = 2q^2$$

que pode ser simplificada:

$$2r^2 = q^2$$

Desta última igualdade podemos concluir que q^2 é um número par. Mas, de novo graças ao teorema A.3.7, também *q é um número par.*

Mas agora já temos uma contradicção: Por um lado a fracção $\frac{p}{q}$ está reduzida: *não podem haver divisores comuns a p e a q.* Por outro lado p e q são ambos números pares: *ambos podem ser divididos por* 2!

Esta contradicção resulta de termos suposto que $\sqrt{2}$ poderia ser um número racional. Portanto $\sqrt{2}$ não pode ser um número racional. □

Uma bijecção entre $\mathbb{N}^2$ e $\mathbb{N}$. A função

$$\begin{array}{rccl} \Pi: & \mathbb{N}^2 & \to & \mathbb{N} \\ & (n,m) & \mapsto & 2^n(2m+1)-1 \end{array} \tag{A.5}$$

é uma bijecção. A imagem inversa do natural $x \in \mathbb{N}$ é calculada à custa de duas funções $\Pi_1, \Pi_2 : \mathbb{N} \to \mathbb{N}$ (as *inversas parciais* de Π) que devem respeitar a equação

$$\Pi\left(\Pi_1(x), \Pi_2(x)\right) = x$$

e, também, sendo $x = \Pi(n,m)$,

$$\left\{ \begin{array}{rl} n = & \Pi_1(x) \\ m = & \Pi_2(x) \end{array} \right. .$$

A questão que se põe agora é: dado x, como calcular n e m, isto é, como definir as inversas parciais Π_1 e Π_2?

Ora, se $x = 2^n(2m+1) - 1$, então $x + 1 = 2^n(2m+1)$ portanto n é o expoente de 2 na factorização prima de $x+1$, *i.e.* $n = (x+1)_1$. Pode ser calculado dividindo sucessivamente $x+1$ por 2 até se obter um ímpar

e contanto as divisões. Então n é o número de divisões sucessivas. Por outro lado, supondo dado x e que n já é conhecido, $m = \frac{1}{2}\left(\frac{x+1}{2^n} - 1\right)$.

Resumindo:

$$\Pi_1(x) = (x+1)_1 \tag{A.6}$$

$$\Pi_2(x) = \frac{1}{2}\left(\frac{x+1}{2^{\Pi_1(x)}} - 1\right) \tag{A.7}$$

A relação entre n, m e x pode ser parcialmente representada por uma tabela

n	m	x	n	m	x
0	0	0	1	0	1
0	1	2	2	0	3
0	2	4	1	1	5
0	3	6	3	0	7
0	4	8	1	2	9

ou, graficamente,

m					$\cdots$	$x = \Pi(n,m)$
$\vdots$	$\ddots$					$\vdots$
4	8	$\ddots$				
3	6	13	$\ddots$			
2	4	9	19	$\ddots$		
1	2	5	11	23	$\ddots$	
0	0	1	3	7		
	0	1	2	3	$\cdots$	n

Classes de crescimento de funções reais de variável real. No caso particular das funções $f : \mathbb{R}_0^+ \to \mathbb{R}_0^+$, por vezes é importante considerar **classes de crescimento**. Sejam f e g duas funções $\mathbb{R}_0^+ \to \mathbb{R}_0^+$:

1. Diz-se que g **domina** f, e escreve-se

$$f \prec g \tag{A.8}$$

se, a partir de certa altura, g é maior que f, isto é, se existe um x_o tal que para qualquer $x > x_o$, $f(x) < g(x)$.

2. A **ordem** da função g,

$$\mathcal{O}(g) = \left\{ f : \mathbb{R}_0^+ \to \mathbb{R}_0^+ \mid \exists c > 0 : f \prec cg \right\} \tag{A.9}$$

é o conjunto de funções dominadas por g, a menos do produto por uma constante positiva.

3. $f \in \Omega(g)$ se existem $x_0 \in \mathbb{R}_0^+$ e $c \in \mathbb{R}^+$ tais que para cada $x \geq x_0$,

$$cg(x) \leq f(x);$$

4. $f \in \Theta(g)$ se existem $x_0 \in \mathbb{R}_0^+$ e $c_1, c_2 \in \mathbb{R}^+$ tais que para cada $x \geq x_0$,

$$c_1 g(x) \leq f(x) \leq c_2 g(x);$$

Os infinitos e o argumento da diagonal. Um dos casos mais espectaculares da aplicação das demonstrações por absurdo é o chamado **argumento da diagonal**, usado pelo matemático Georg Cantor para mostrar que o conjunto dos números reais não é enumerável. Isto é, que $\mathbb{R}$ e $\mathbb{N}$, embora sendo ambos conjuntos infinitos, são *infinitos diferentes*! O *infinito* de $\mathbb{R}$ é maior do que o de $\mathbb{N}$.

Vejamos então o que se passa com o argumento da diagonal. O problema que se procura resolver é o seguinte: sabemos que o conjunto dos números naturais, $\mathbb{N}$, tem infinitos elementos. Como $\mathbb{N} \subset \mathbb{R}$, também o conjunto dos números reais tem infinitos elementos. Mas,

> *Será que o infinito de $\mathbb{N}$ é «igual» ao infinito de $\mathbb{R}$? Só há um infinito, ou existirão vários?*

Para responder a esta questão é necessário partir das noções intuitivas «mesmo número de elementos» e «infinito» e dar-lhes um sentido rigoroso:

Figura A.4: **Georg Cantor** (*n.* 3 de Março de 1845, *f.* 6 de Janeiro de 1918) é o pai da moderna teoria dos conjuntos e marcou decisivamente a matemática e o pensamento contemporâneos. Deve-se a ele, por exemplo, a noção de **conjunto enumerável** e a notação $\mathbb{R}$ para representar o conjunto dos números reais. Dele disse David Hilbert: *Ninguém nos poderá expulsar do Paraíso que Cantor criou.*

1. Dois conjuntos A e B **têm o mesmo número de elementos** se existir uma função bijectiva $f : A \to B$;

2. Um conjunto X é **infinito** se existe um subconjunto próprio $Y \subset X$, $Y \neq X$ com o mesmo número de elementos que X;

Por exemplo os conjuntos $A = \{1, 2, 3\}$ e $B = \{4, 5, 6\}$ têm o mesmo número de elementos porque a função

$$\begin{array}{rccc} f: & A & \to & B \\ & x & \mapsto & x+3 \end{array}$$

é injectiva e sobrejectiva (**exercício:** verifique).

Outros dois conjuntos com o mesmo número de elementos são $\mathbb{N}$ e $\mathbb{Z}$. Para justificar esta afirmação basta verificar (**exercício!**) que a função

$$\begin{array}{rccl} f: & \mathbb{N} & \to & \mathbb{Z} \\ & n & \mapsto & \begin{cases} \frac{n}{2} & \text{se } n \text{ é par;} \\ -\frac{n+1}{2} & \text{se } n \text{ é ímpar;} \end{cases} \end{array}$$

é uma bijecção entre $\mathbb{N}$ e $\mathbb{Z}$. Também já vimos que $\mathbb{N}$ e $\mathbb{N}^2$ têm o mesmo número de elementos.

Sobre os conjuntos infinitos, consideremos de novo o conjunto dos números naturais e o subconjunto $P = \{n \in \mathbb{N} \mid n \text{ é par.}\}$. Ora, $P \subset \mathbb{N}$ mas $P \neq \mathbb{N}$ porque, por exemplo, $1 \in \mathbb{N}$ mas $1 \notin P$. Mas, além disso, P e N têm o mesmo número de elementos:

$$\begin{array}{rccl} f: & \mathbb{N} & \to & P \\ & n & \mapsto & 2n \end{array}$$

é uma bijecção entre $\mathbb{N}$ e P. Concluímos que $\mathbb{N}$ tem um subconjunto próprio (P) com o mesmo número de elemento que $\mathbb{N}$, portanto $\mathbb{N}$ é um conjunto infinito.

Exercício A.3.9 *Mostre que o conjunto $A = \{1, 2, 3\}$ não é infinito (sugestão: considere todos os subconjuntos próprios de A).*

Neste momento já vimos, com algum rigor, o que quer dizer «conjunto infinito» e podemos ter uma noção mais informada sobre como responder à questão de existirem «infinitos diferentes»: basta encontrar dois conjuntos «infinitos» que não tenham «o mesmo número de elementos» para ficarmos a saber que existem infinitos diferentes. Ora, o problema que se pôs no tempo de Cantor foi que já se sabia que $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{N}^2$, o conjunto dos primos, dos pares, dos negativos, *etc.* têm, todos, o mesmo número de elementos. E suspeitava-se que $\mathbb{R}$ teria «outro tipo de infinito». Também já se sabia que $\mathbb{R}$ e, por exemplo, o intervalo $]0, 1[$ têm o mesmo número de elementos. Mas, como provar que $\mathbb{R}$, ou $]0, 1[$, têm, de facto, «mais elementos» do que $\mathbb{N}$?

Esta questão ficou resolvida com uma demonstração genial de Cantor:

Demonstração. Supomos, com vista a um absurdo, que $]0,1[$ e $\mathbb{N}$ têm o mesmo número de elementos. Então existe uma bijecção $f : \mathbb{N} \to]0,1[$. Vamos descrever esta função da seguinte forma: consideramos uma «tabela» infinita, e nessa tabela escrevemos, na primeira coluna $f(0)$, na segunda coluna $f(1)$, *etc.*:

$$\left[\begin{array}{cccccc} f(0) & f(1) & \cdots & f(n) & \cdots \end{array}\right].$$

Agora, como cada $f(n) \in]0,1[$, na forma decimal temos $f(n) = 0.d_1d_2d_3\cdots$ em que os d_i são os dígitos de $f(n)$. Ilustremos: se $f(0) = 0.7253$ então $d_1 = 7; d_2 = 2; d_3 = 5; d_4 = 3$ e $d_i = 0$ quando $i \geq 5$.

Todos os números no intervalo $]0,1[$ podem ser escritos desta forma[2]. Agora desenvolvemos as colunas na tabela anterior:

$$\left[\begin{array}{ccccc} a_{0;1} & a_{1;1} & \cdots & a_{n;1} & \cdots \\ a_{0;2} & a_{1;2} & \cdots & a_{n;2} & \cdots \\ \vdots & \vdots & \ddots & \vdots & \\ a_{0;n+1} & a_{1;n+1} & \cdots & a_{n;n+1} & \cdots \\ \vdots & \vdots & & \vdots & \end{array}\right]$$

onde $a_{n;j}$ é o j-ésimo dígito de $f(n)$. Continuamos a ter $f(n)$ na $n+1$-ésima coluna da tabela, mas agora estamos a «olhar» para os dígitos desse número.

Agora vamos aplicar a técnica fabulosa que Cantor introduziu. Consideremos o número real b, definido da seguinte forma:

$$b = 0.b_1b_2\cdots b_n\cdots$$

em que $b_1, b_2, \ldots, b_n, \ldots$ são os dígitos de b, definidos por:

[2]Aqui há um pequeno detalhe a considerar. É que certos números podem ser escritos de duas maneiras diferentes. Por exemplo, 0.1 é exactamente igual a $0.099\cdots$. Mas então, como deve ser escrito esse número? Como 0.1 ou como $0.99\cdots$? Podemos supor que nestes casos se escolhe a única forma mais curta (***exercício:*** *porque é única?* sugestão: considere que um número tem duas escritas mais curtas; onde diferem?).

1. se $a_{0;1} = 9$ então $b_1 = 0$; caso contrário, $b_1 = a_{0;1} + 1$;
2. se $a_{1;2} = 9$ então $b_2 = 0$; caso contrário, $b_2 = a_{1;2} + 1$;
3. ...
4. se $a_{n-1;n} = 9$ então $b_n = 0$; caso contrário, $b_n = a_{n-1;n} + 1$;
5. ...

Isto é, estamos a mudar os dígitos sublinhados, na *diagonal da tabela*

$$\begin{bmatrix} \underline{a_{0;1}} & a_{1;1} & \cdots & a_{n;1} & \cdots \\ a_{0;2} & \underline{a_{1;2}} & \cdots & a_{n;2} & \cdots \\ \vdots & \vdots & \ddots & \vdots & \\ a_{0;n+1} & a_{1;n+1} & \cdots & \underline{a_{n;n+1}} & \cdots \\ \vdots & \vdots & & \vdots & \end{bmatrix}$$

acrescentando 1, e voltando a «0» quando o dígito sublinhado é «9». Por exemplo

$$\begin{bmatrix} \underline{1} & 7 & \cdots & 0 & \cdots \\ 5 & \underline{9} & \cdots & 3 & \cdots \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 2 & \cdots & \underline{4} & \cdots \\ \vdots & \vdots & & \vdots & \end{bmatrix} \rightarrow \begin{bmatrix} \underline{2} & 7 & \cdots & 0 & \cdots \\ 5 & \underline{0} & \cdots & 3 & \cdots \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 2 & \cdots & \underline{5} & \cdots \\ \vdots & \vdots & & \vdots & \end{bmatrix}$$

iríamos obter

$$b = 0.20 \cdots 5 \cdots$$

Tentemos localizar b na tabela inicial. Ora, b não pode ser

1. $f(0)$ porque $b_1 \neq a_{0;1}$;
2. $f(1)$ porque $b_2 \neq a_{1;2}$;
3. ...

4. $f(n)$ porque $b_{n+1} \neq a_{n;n+1}$;

5. ...

Conclusão: afinal, b não está na tabela. Mas tinha de estar! Recapitulemos o que fizemos: a função f é uma bijecção entre $\mathbb{N}$ e o intervalo $]0,1[$. Isto implica que todos os números reais desse intervalo então na tabela. Cada número real (do intervalo $]0,1[$) está numa certa coluna, não interessa qual. Portanto b teria de estar em alguma coluna da tabela. Mas não está porque, olhando para os seus dígitos, concluímos que é diferente de qualquer número que esteja numa coluna.

Esta contradição mostra que não podemos supor que exista uma bijecção como f: os conjuntos $\mathbb{N}$ e $]0,1[$ não podem ter o mesmo número de elementos. Portanto, há infinitos maiores que outros. □

O golpe de génio de Cantor, nesta demonstração, consiste na técnica de percorrer a diagonal de uma tabela e fazer uma pequena alteração de forma a especificar um elemento que não constava anteriormente na tabela mas que, não obstante, devia lá estar. Esta técnica, muito geral e extremamente poderosa, foi mais tarde usada por Kurt Gödel para demonstrar o seu influente **Teorema da Incompletude** e também por Alan Turing para demonstrar a existência de predicados algoritmicamente indecidíveis.

Anexo B

Complementos

B.1 Computabilidade do Índice da Composição

Uma das operações importantes na computação em geral é a **composição** de programas. No caso das funções parciais recursivas, a composição é uma das regras que produzem novas funções. No capítulo 3, onde se demonstra que a composição de funções computáveis é computável (teorema 3.1.3), definimos um esquema geral para a composição de programas URM. Agora estamos interessados no caso mais simples, a composição de dois programas URM: supondo que F e G são programas URM, por composição definimos um outro programa

$$C = [F\,[1, \ldots, n \to 1]\,, G\,[1 \to 1]]\,.$$

A questão que vamos resolver aqui é a seguinte:

> Dados os códigos x e y de F e G, respectivamente, será que podemos computar o código z de C?

Mostraremos que sim, que a função

$$\begin{array}{rccc} \mathsf{comp}: & \mathbb{N}\times\mathbb{N} & \to & \mathbb{N} \\ & (x,y) & \mapsto & z \end{array}$$

é computável. Vamos enunciar esta tarefa com um pouco mais de cuidado: *Queremos determinar uma função parcial recursiva que, dados dois naturais x e y, compute o código do programa obtido pela composição de* $\mathsf{dec}\,(x)$ *com* $\mathsf{dec}\,(y)$, isto é, o código $\mathsf{cod}\,(C)$ do programa[1]

$$C = \left[\begin{array}{l|l} 1. & \mathsf{dec}\,(x)\,[1,\ldots,N \to 1] \\ 2. & \mathsf{dec}\,(y)\,[1 \to 1] \end{array}\right].$$

Esta tarefa, que acaba por ser revelar um pouco complexa, será dividida em vários passos:

1. Resolução URM: definimos as instruções do programa C;
2. Bases: analisamos as condições necessárias para calcular $\mathsf{cod}\,(C)$;
3. Funções Auxiliares: mostramos a computabilidade de certas funções auxiliares;
4. Juntando as peças: usamos as funções anteriores para calcular $\mathsf{cod}\,(C)$;

Resolução URM Supondo que $\mathsf{dec}\,(x) = [X_1,\ldots,X_n]$, $\mathsf{dec}\,(y) = [Y_1,\ldots,Y_m]$ e que $q = \mathsf{rho}\,(y)$, podemos definir, por exemplo,

$$C = \left[\begin{array}{r|c} 1. & X_1 \\ \vdots & \vdots \\ n. & X_n \\ n+1. & Z\,(2) \\ \vdots & \vdots \\ n+q-1. & Z\,(q) \\ n+q. & Y_1^{n+q-1} \\ \vdots & \vdots \\ n+q+m-1. & Y_m^{n+q-1} \end{array}\right]$$

[1]Para ser formalmente correcto, seria o código de algum programa D que, dados os mesmos argumentos que forem dados a C, termine com o mesmo resultado que C: $D\,(x) \downarrow b$ se, e só se, $C\,(x) \downarrow b$.

onde Y_j^k resulta de ajustar Y_j substituindo as instruções $\mathsf{J}(a,b,q)$ por $\mathsf{J}(a,b,q+k)$.

Bases Assim, para determinarmos $\mathsf{cod}(C)$, necessitamos de calcular:

1. *o espaço de trabalho de um programa*, dado o seu código n, isto é, a função rho;

2. *o número de instruções num programa*, dado o seu código $n = 2^{b_1} + \cdots + 2^{b_1+\cdots+b_k+k-1} - 1$, isto é, a função $\mathsf{numi} : n \mapsto k$;

3. *o código da i-ésima instrução*, dado o código $n = 2^{b_1} + \cdots + 2^{b_1+\cdots+b_k+k-1} - 1$ de um programa, isto é, a função $\mathsf{codi} : (n,i) \mapsto b_i$;

4. *o ajuste por k das instruções de tipo salto*, isto é, sendo b o código de uma instrução, $\mathsf{ajuste}(b,k)$ faz esse ajuste;

5. a partir dos códigos das instruções de C, o código $\mathsf{cod}(C)$.

Funções auxiliares Abreviemos algumas funções já conhecidas:

- $\mathsf{S}(x)$ e $\mathsf{soma}(x,y)$, por $x+1$ e $x+y$;
- $\mathsf{pred}(x)$ e $\mathsf{monus}(x,y)$, por $x \dot{-} 1$ e $x \dot{-} y$;
- $\mathsf{prod}(x,y)$, por $x \times y$

A função ajuste. Para fazermos o ajuste das instruções J, notemos que estas instruções, quando codificadas por β, deixam resto 3 se divididas por 4. Como

$$\beta(\mathsf{J}(n,m,q)) = 4\Pi(\Pi(n-1,m-1),q-1)+3,$$

se $b = \beta(\mathsf{J}(n,m,q))$, tem-se $\Pi(n-1,m-1) = \Pi_1\left(\frac{b-3}{4}\right)$ e $q = \Pi_2\left(\frac{b-3}{4}\right)+1$.

Portanto, para ajustarmos uma instrução de código b, de forma a «saltar» mais k linhas, temos

$$\mathsf{ajuste}\,(b,k) = \begin{cases} b & \text{se } \mathsf{mod}\,(b,4) \neq 3 \\ 4\Pi(\Pi_1(\mathsf{qt}(4, b \dot{-} 3))\,, \Pi_2(\mathsf{qt}(4, b \dot{-} 3) + 1) + k) + 3 & \text{c.c.} \end{cases}$$

As funções numi *e* codi. Por outro lado, dado o código n de um programa, as funções numi e codi podem ser definidas por minimização limitada, composição e recursão:

$$\mathsf{numi}\,(n) = \mu_{m \leq n}\,[\mathsf{icauda}\,(n,m)]$$

e

$$\mathsf{codi}\,(n,i) = \mathsf{cprim}\,(\mathsf{icauda}\,(n,i))$$

em que

$$\begin{cases} \mathsf{icauda}\,(n,0) & = & n \\ \mathsf{icauda}\,(n,y+1) & = & \mathsf{cauda}\,(\mathsf{icauda}\,(n,y)) \end{cases}$$

A função cauda determina, dado o código n de $[I_1, I_2, \ldots, I_p]$, o código do programa que resulta se for removida a primeira instrução, $[I_2, \ldots, I_p]$ enquanto que a função icauda usa mais um argumento, y e computa o código do programa que resulta se forem retiradas as primeiras y instruções. Além disso, cprim determina, dado o código n de um programa, o código da sua primeira instrução:

$$\begin{array}{rcl} \mathsf{cprim}\,(n) & = & \mathsf{me2}\,(n+1) \\ \mathsf{cauda}\,(n) & = & \mathsf{qt}\,(\mathsf{exp2}\,(\mathsf{cprim}\,(n))\,, n+1) \dot{-} 1 \end{array}$$

Aritmética. Nas funções anteriores usámos várias funções aritméticas:

- a potência de base 2, $\mathsf{exp2} : x \mapsto 2^x$;
- o maior expoente de 2, isto é, a função $\mathsf{me2} : x \mapsto k$ em que k é o maior inteiro que 2^k divide x;

- o cociente inteiro $\mathsf{qt} : x, y \mapsto z$ em que z é o maior inteiro que $x \times z \leq y$;
- o resto da divisão inteira $\mathsf{rm} : x, y \mapsto x - y \times \mathsf{qt}\,(y, x)$;
- a bijecção $\Pi : \mathbb{N}^2 \to \mathbb{N}$ e as respectivas projecções Π_1 e Π_2.

Todas estas funções são primitivas recursivas, como verificámos nos capítulos 2 e 3. Também o é o operador $\wedge_h$, semelhante ao somatório Σ_h e produtório Π_h de funções primitivas recursivas, de que iremos necessitar: dada uma função primitiva recursiva h, definimos recursivamente $\wedge_h$ por

$$\begin{cases} \wedge_h (\boldsymbol{x}, 0) & = & h\,(\boldsymbol{x}, 0) \\ \wedge_h (\boldsymbol{x}, y+1) & = & \mathsf{max}\,(h\,(\boldsymbol{x}, y+1)\,, \wedge_h (\boldsymbol{x}, y)) \end{cases}$$

isto é, $\wedge_h$ é a função $\boldsymbol{x}, y \mapsto \mathsf{max}\,\{h\,(\boldsymbol{x}, i) \mid i \leq y\}$. A dimensão do argumento $\boldsymbol{x}$ é finita, mas arbitrária[2]; estamos a considerar $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{N}^n$ e $n \in \mathbb{N}$.

Ora, o máximo e o mínimo de dois argumentos podem ser definidos por

$$\begin{array}{rcl} \mathsf{max}\,(x, y) & = & (x + y) \dot{-} \mathsf{min}\,(x, y) \\ \mathsf{min}\,(x, y) & = & x \dot{-} (x \dot{-} y) \end{array}$$

A função rho. Agora podemos definir rho:

$$\mathsf{rho}\,(n) = \wedge_{\mathsf{mr}}\,(n, \mathsf{numi}\,(n))$$

em que

$$\mathsf{mr}\,(n, i) = \mathsf{maxreg}\,(\mathsf{codi}\,(n, i))$$

e maxreg determina, dado o código b de uma instrução, o maior índice de registo usado por esta instrução (convém ter presente a definição da codificação β):

[2]Rigorosamente, para cada aridade $n+1$ da função h, deveríamos fazer uma definição apropriada para $\wedge_h^{(n)}$.

$$\mathsf{maxreg}\,(b) = \begin{cases} \mathsf{qt}\,(4,b)+1 & \text{se } \mathsf{rm}\,(b,4)=0 \\ \mathsf{qt}\,(4,b\ominus 1)+1 & \text{se } \mathsf{rm}\,(b,4)=1 \\ \mathsf{max}\,(\Pi_1\,(\mathsf{qt}\,(4,b\ominus 2))\,,\Pi_2\,(\mathsf{qt}\,(4,b\ominus 2)))+1 & \text{se } \mathsf{rm}\,(b,4)=2 \\ \mathsf{max}\,(\Pi_1\,(\Pi_1\,(\mathsf{qt}\,(4,b\ominus 3)))\,,\Pi_2\,(\Pi_1\,(\mathsf{qt}\,(4,b\ominus 3))))+1 & \text{se } \mathsf{rm}\,(b,4)=3 \end{cases}$$

Juntando as peças. Nesta fase definimos

$$\begin{aligned} x_i &= \beta\,(X_i) \\ &= \mathsf{codi}\,(x,i) \end{aligned}$$

$$\begin{aligned} y_i &= \beta\,(\mathsf{ajuste}\,(\beta\,(Y_i)\,,(n+q)\ominus 1)) \\ &= \mathsf{ajuste}\,(\mathsf{codi}\,(y,i)\,,(\mathsf{numi}\,(x)+\mathsf{rho}\,(y))\ominus 1)\,, \end{aligned}$$

e notamos que o «-1» que costuma ser somado na codificação de programas está, já, incluído em x. Então,

$$\begin{array}{rc|l} & \mathsf{cod}\,(C)=\mathsf{comp}\,(x,y) & \\ = & x & [X_1,\ldots,X_n] \\ + & 2^{a+4+1}+\cdots+2^{a+4+\cdots+4(q-1)+(q-1)} & [Z\,(2)\,,\ldots,Z\,(q)] \\ + & 2^{c+y_1+1}+\cdots+2^{c+y_1+\cdots+y_m+m} & \left[Y_1^{n+q-1},\ldots,Y_m^{n+q-1}\right] \end{array}$$

onde as parcelas dos expoentes são obtidas de

$$\begin{aligned} a = a(x) &= x_1 + \cdots + x_n + n \dot{-} 1 \\ &= \left(\sum_{i=1}^{\mathsf{numi}(x)} \mathsf{codi}(x,i) \right) + \mathsf{numi}(x) \dot{-} 1 \end{aligned}$$

$$\begin{aligned} b(x,j) &= a(x) + j + 4 \sum_{i=1}^{j} i \\ &= a(x) + j + 2 \times (j \times (j+1)) \end{aligned}$$

$$c = c(x,y) = b(x, \mathsf{rho}(y) - 1)$$

$$\begin{aligned} d(x,y,j) &= c(x,y) + j + \sum_{i=1}^{j} y_i \\ &= c(x,y) + j + \sum_{i=1}^{j} \mathsf{ajuste}(\mathsf{codi}(y,i), (\mathsf{numi}(x) + \mathsf{rho}(y)) \dot{-} 1) \end{aligned}$$

Finalmente, temos

$$\mathsf{comp}(x,y) = x + \sum_{i=1}^{\mathsf{rho}(y)} 2^{b(x,i)} + \sum_{i=1}^{\mathsf{numi}(y)} 2^{d(x,y,i)}$$

e podemos então garantir que

Teorema B.1.1 (Computabilidade do índice da composição) *Existe uma função parcial recursiva*

$$\mathsf{comp} : \mathbb{N}^2 \to \mathbb{N}$$

que, dados códigos x e y de programas URM,

$$\mathsf{comp}(x,y)$$

é o código do programa URM que resulta da composição dos programas dados.

Numa última observação, *estamos a supor que o programa codificado por y usa um único argumento*. Mas isso tem mesmo de acontecer na composição de funções primitivas recursivas: se f e g forem funções parciais recursivas, computadas, respectivamente, por $\mathsf{dec}\,(x)$ e $\mathsf{dec}\,(y)$, então a composição $g \circ f$ é computada pelo programa $\mathsf{dec}\,(\mathsf{comp}\,(x, y))$. Ora, como a assinatura das funções parciais recursivas é $\mathbb{N}^n \to \mathbb{N}$, para que esta composição seja possível, temos, obrigatoriamente, $g : \mathbb{N} \to \mathbb{N}$.

Índice