Introdução à Teoria da Computação

Simão Melo de Sousa

1. Aviso Prévio

- A redacção deste documento baseou-se fortemente na bibliografia indicada. Parece-nos então óbvio que a leitura e a aprendizagem directa pelas obras originais é recomendada, e mesmo essencial à compreensão profunda das noções aqui apresentadas;
- O português não é a língua materna do autor e o presente documento encontra-se em fase de elaboração pelo que se agradece e até se incentiva qualquer sugestão ou correcção.

2. Bibliografia

Consultar [1, 2, 3, 4, 5, 6]

3. Objectivos

Existem limites à capacidade de resolução de problemas por um computador, mesmo na hipótese "idealista" de ausência de restrição em termos de tempo (de execução) e de espaço (memória).

Para delinear esse limites, visaremos:

- perceber a capacidade de computação das máquinas, assim como os seus limites teóricos. Precisaremos de definir formalmente o que é e o que não é um programa, um algoritmo, ou mais genericamente o que é um tratamento efectivo[1];
- perceber os conceitos que fundamentam as linguagens de programação. Precisaremos de determinar e estudar formalmente as construções que determinam a expressividade (ou capacidade de computação) das linguagens de programação assim como o comportamento dos programas.

[1] Consideraremos neste texto os termos "algoritmo", " método", "estratégia" e " tratamento efectivo" como sinónimos.

4. Tratamentos efectivos

De forma abstracta, o conceito de tratamento efectivo designa as transformações que uma máquina é capaz de submeter aos dados que recebe em entrada tendo como finalidade obter os resultados esperados. Outra definição é a de método sistemático, definido passo a passo, para a resolução sem falhas de determinados problemas.

Se queremos determinar se um número n é primo ou não podemos optar pela estratégia seguinte:

Para um determinado número inteiro n maior do que 1

- tentar a divisão inteira de n por 2
  $\Longrightarrow$ Se o resto for 0 então n é divisível por 2, logo n não é primo
- Senão, experimentar a divisão por 3
  $\Longrightarrow$ Se o resto for 0 então n é divisível por 3, logo n não é primo
- etc... repetir o processo até $n-1$
  $\Longrightarrow$ Se o resto for 0 então n é divisível, logo n não é primo
- Se nenhuma divisão resultou então n é primo

Este algoritmo é muito elementar (existem métodos bem mais eficientes) mas, com tal método, estamos assegurados:

- de sempre saber o que é preciso fazer para obter uma resposta, quer seja positiva, quer seja negativa;

- de que funciona qualquer que seja o número inteiro positivo n; i.e. que este método responde a uma *classe* de problemas e não a uma instância particular (como "Será 1234 um número primo ?");
- de que, em particular, precisa de um número finito de cálculos para apresentar uma resposta (mais uma vez, quer seja positiva quer seja negativa);
- de obter a resposta correcta.

Este exemplo mostra também que um algoritmo é definido a custa de instruções elementares (e.g. dividir), de testes (i.e. verificação do resultado da divisão) e de uma estrutura definida pela ordem na qual esses últimos se sucedem.

Uma questão relevante é a do tempo de resposta. De facto, tê-la em tempo finito ou não, diferencia, como o veremos com mais detalhes mais além, os tratamentos efectivos que são totais dos que são parciais. No caso dos algoritmos totais, surge outro aspecto da mesma questão que não desenvolveremos aqui: a rapidez da resposta. Genericamente, tendo como certo que o tempo de resposta é finito, trata-se de estabelecer uma função entre o tamanho dos dados e o tempo de produção da resposta. Esta função representa a avaliação da complexidade em tempo do método proposto.

Os tratamentos efectivos foram definidos e utilizados bem antes da aparição da informática e dos computadores. Se olharmos para a história dos algoritmos, ou mais genericamente da informática, vemos que podemos dividir esse métodos em duas famílias:

1. os métodos lógicos que prefiguram os algoritmos, programas e linguagens de programação;

2. os métodos mecânicos que prefiguram as máquinas, autómatos e computadores.

Ambas são relevantes para perceber como e porque apareceram os diferentes modelos da computação. Damos assim a seguir uma curta introdução, a luz do refinamento da noção de algoritmo e de computador, aos primórdios da informática.

4.1. O nascer de uma disciplina

Pre-história da informática: Os primeiros algoritmos conhecidos foram desenvolvidos na babilónia por volta de 1800 A.C.. Muitos e famosos algoritmos foram descobertos na altura da Grécia antiga. Por exemplo, o crivo de Eratostene é um método para encontrar os números primos menores do que um determinado número. O algoritmo de Euclides (elaborado mais ou menos em 300 A.C.) é um método para determinar se dois números são primos entre eles.

Há registo também das "abaques" que são tábuas sobre as quais se deslocavam pedras ("calculi") para efectuar operações básicas como contagens.

O próprio nome, "algoritmo", é uma derivação do nome do matemático do império persa do século IX, Al-Khowarizmi. Este último é conhecido por ser o autor em 825 de um tratado de aritmética onde transmitia aos árabes métodos de cálculo conhecidos pelos índios.

Século XIII: concepção do "Ars Magna" por Raymond Lulle. Trata-se duma máquina lógica constituída por círculos concêntricos contendo palavras que, dispostas numa certa ordem, formavam perguntas e respostas;

1624: Wilhem Schickard inventa um relógio automático calculador em Heidelberg;

1642: Blaise Pascal cria na idade dos 19 anos a "Pascaline", uma calculadora mecânica baseada em engrenagens capaz de somar e subtrair. A linguagem Pascal foi assim chamada em honra do cientista francês;

1673: Leibniz, outro grande cientista, melhora a Pascaline de tal forma a que possa efectuar multiplicações e divisões;

1805: Jacquart cria as máquinas de tecer automáticas (teares) que utilizam cartões furados como meio de programar o motivo para tecer;

1822 e 1833: O engenheiro inglês Babbage planeia a "máquina de diferenças" (1822). Demasiada complexa para a tecnologia da época, só será construída no século XX. Em 1833 Babbage planeia uma máquina ainda mais complexa e eficiente, chamada "máquina analítica", que utiliza, como as máquinas de Jacquart, cartões furados como programas. Babbage tentará, a todo o custo (ficará arruinado) e sem sucesso, construí-la. Terá a ajuda de Lady Ada Lovelace, filha do poeta inglês Lord Byron, que escreverá os primeiros programas que a máquina analítica teria sido capaz de executar. A linguagem Ada foi assim chamada em honra dela;

1854: O lógico inglês George Boole publica o livro "The mathematical Analysis of logic" onde define os operadores lógicos baseados nos valores 0 (que significa falso) e 1 (verdade): os famosos operadores booleanos;

1884-1896: O engenheiro americano Hollerith inventa e comercializa calculadoras eléctricas na empresa que criou, a "Tabulation Machine Corporation", a futura IBM. A HP, empresa de material electrónico será criada em 1938;

1936: Alan Turing e Alonzo Church propõem os primeiros modelos da computação;

1939: John Atanassoff e Clifford Berry constroem um protótipo chamado "ABC", que é considerado como o primeiro computador digital;

1939-1945: durante a guerra

- Alan Turing trabalha nos serviços secretos na decifra das comunicações alemãs (cifradas pela máquina "enigma"). Para esse efeito inventou uma máquina, o "Colossus";
- John Von Neumann desenvolve, ao abrigo dum projecto militar, as suas primeiras ideias de arquitectura de uma máquina universal;

1945: John Von Neumann propõe finalmente o seu modelo de arquitectura interna de uma máquina universal (computador);

1946: Construção do ENIAC, último computador eléctrico programável;

1949: Construção do EDVAC, o primeiro computador baseado na arquitectura de Von Neumann.

A partir desta data os computadores e a informática são uma realidade bem concreta. Como esta pequena cronologia o mostra, as noções de algoritmos e de mecanismos de cálculos são bem anteriores a era que classicamente entendemos ser o da informática (o século XX). No entanto a necessidade de dar uma definição formal rigorosa ao conceito de tratamento efectivo só surgiu no anos 30 do século passado e com a finalidade de dar respostas a considerações bem mais gerais e fundamentais.

5. Os problemas de Hilbert

A história contemporânea dos algoritmos começa em 1900, mais particularmente com o congresso da sociedade matemática em Paris. O convidado de honra do congresso é um famoso e brilhante matemático alemão chamado David Hilbert. Para celebrar a data do congresso, Hilbert estabelece uma lista de problemas fundamentais da matemática em aberto que supostamente representariam (e, de facto, representaram) os desafios da investigação matemática do século XX: os famosos 23 problemas de Hilbert.

O décimo problema trata de equações diofantinas e enuncia-se da forma seguinte: " Existirá um método efectivo para resolver qualquer equação diofantina ?". Uma equação diofantina expressa-se na forma de um polinómio com coeficientes inteiros do qual se pretende encontrar as raízes inteiras.

Em alguns casos particulares sabia-se resolver este problema, mas no caso geral nenhum algoritmo era conhecido. Os matemáticos sabem demonstrar que um determinado método (como o teste de primalidade acima apresentado) é correcto (parcialmente ou totalmente[2]). Mas, infelizmente, não sabiam na altura demonstrar a falta de existência de métodos de resolução para um determinado problema, até porque a própria noção de algoritmo não era formal.

De facto, como ter a certeza de ter experimentado todos os métodos possíveis para resolver um problema? Como ter a certeza que o facto de nenhuma tentativa até ao momento ter resultado significa efectivamente que não existe um método de resolução? Numa situação semelhante, poderíamos argumentar que se não encontramos método é que não procuramos o suficiente.

[2] A correcção total acrescenta a correcção parcial a propriedade de terminação.

Resolver tal dilema necessitava considerar a própria noção de método como um objecto matemático que poderá, desta forma, ser alvo de raciocínios e de demonstrações.

Tal aconteceu na primeira metade do século XX em que grande parte da investigação em lógica se debruçou sobre a formalização da noção de algoritmo.

Em 1928, em Bolonha, David Hilbert é de novo convidado ao congresso internacional de matemática. Nesta ocasião Hilbert refina o seu programa de investigação. Hilbert esperava encontrar uma teoria que pudesse fundamentar as matemáticas que, nesta altura, atravessavam uma crise profunda (a chamada "crise dos fundamentos" devido a descoberta de paradoxos em teorias que serviam de alicerces à matemática da época). Hilbert esperava estabelecer que a lógica, na altura em plena expansão, era um formalismo adequado para fundamentar a matemática. Mas, para tal, era preciso assegurar que a lógica dispunha de boas propriedades: consistência (ausência de contradições), completude (qualquer proposição é verdade ou falsa de forma exclusiva) e decidível (existência dum método permitindo estabelecer se uma fórmula qualquer é verdade ou falsa). Esta última propriedade foi qualificada por Hilbert como o "entcheidungsproblem", ou "problema de decisão".

A resolução do entcheidungsproblem resultaria num algoritmo capaz de resolver enunciados (i.e. encontrar uma demonstração que confirmasse ou desconfirmasse a veracidade do enunciado em entrada). Para perceber a importância de tal algoritmo, basta referir que resolveria sem erro e automaticamente enunciados como a conjectura de Goldbach, que se formula da forma seguinte:

$$\forall k > 1. \exists p, q.\ \ 2k = p + q \ \wedge\ \acute{e}\_primo(p) \ \wedge\ \acute{e}\_primo(q)$$

Infelizmente para Hilbert, o lógico austríaco Kurt Gödel demonstra em 1929 e particularmente em 1931 que a lógica elementar é completa mas que qualquer teoria suficientemente expressiva para fundamentar a aritmética é necessariamente incompleta e que a sua consistência não é demonstrável. Quais são as consequências? O programa de Hilbert nunca será completado e ser matemático ainda continuará a ser uma profissão respeitável.

Resta o problema de decisão, ao qual Alonzo Church, Stephen Kleene e Alan Turing darão, de forma independente, uma resposta negativa em 1936. Para tal, ambos tiveram de formalizar a noção de tratamento efectivo e demonstrar que nenhum tratamento efectivo puderia resolver o problema de decisão.

5.1. Os primeiros modelos da computação

Em 1936, Alan Turing tem 24 anos. É um jovem e brilhante estudante de matemática em Cambridge. Incentivado pelo seu professor de lógica que assistiu ao congresso de Bolonha, Turing fascina-se pela capacidade humana de raciocinar e de calcular. Durante a segunda guerra mundial, nos serviços secretos ingleses, estará envolvido na construção de máquinas "quebra-códigos" e mais tarde tentará modelar computacionalmente o cérebro humano.

No seu artigo de 1936, Turing propõe de formalizar a noção de algoritmo a custa dum dispositivo abstracto que se passou a chamar "máquinas de Turing". Demonstrou, a seguir, que nenhum desses dispositivos era capaz de decidir se uma fórmula lógica *qualquer* era falsa ou verdadeira.

As máquinas de Turing prefiguraram o desenvolvimento das máquinas digitais e os computadores, enquanto a solução de Alonzo Church, o cálculo λ

(ler "lambda"), prefigura o aparecimento das linguagens de programação.

Alonzo Church, lógico americano, estava mais interessado nos fundamentos da matemática. Desenvolveu em 1936 uma teoria das funções que visava, igualmente, a mecanização da lógica e, por tabela, da matemática que se revelou inconsistente (facto demonstrado por S. Kleene). Do refinamento da teoria original resultou o cálculo lambda.

Stephen Kleene, por seu turno, propõe um modelo chamado teoria das funções recursivas. Demonstrou que essa teoria era, em termos de expressividade, equivalente ao cálculo lambda. Da mesma forma, ficou estabelecida a equivalência entre as máquinas de Turing e as funções recursivas.

Outros modelos da computação entretanto apareceram. Veremos, ao longo desta disciplina, alguns deles. Os três modelos citados têm uma importância histórica por serem os primeiros modelos nos quais resultados importantes da teoria da computação foram enunciados e demonstrados.

5.2. A tese de Church

Uma máquina de Turing, um termo λ ou uma função parcial recursiva podem ser visto como a concretização no modelo em questão de uma função matemática. Trata-se de facto de uma máquina, se nós nos concentramos no modelo de Turing, que aceita dados de entrada e que executa um processo de transformação bem determinado resultando na produção de dados de saída.

Church e Turing afirmaram que o que informalmente entendemos por algoritmo podia ser expresso em termos de uma máquina de Turing/termo λ /função parcial recursiva. Esta afirmação é conhecida por **Tese de Church**.

É importante insistir sobre o facto que essa afirmação é uma tese. Como

tal não tem o papel de um teorema com uma demonstração conhecida mas mais o papel de uma definição: Church e Turing propõem que se identifique a noção de algoritmo com a de máquina de Turing/termo λ /função parcial recursiva.

Como ficar convencido com tal afirmação? De facto, a tese de Church é aceite pela comunidade científica e vários argumentos vêm fortalecê-la:

Argumentos intuitivos: nunca foi exibido um exemplo de tratamento efectivo que não pudesse ser implementado por uma máquina de Turing;:

Enriquecimento dos modelos da computação: foram propostas várias extensões aos diferentes modelos citados. Para cada uma delas ficou demonstrado que não traziam poder expressivo suplementar em comparação com o modelo original. I.e.:

- podia-se simular as extensões propostas no próprio modelo original;
- nenhuma extensão podia realizar cálculos que o modelo original não pudesse.

O papel das extensões não actuavam assim ao nível da expressividade mas sim, eventualmente, ao nível da eficiência ou do "conforto" de utilização. Os modelos originais pareciam assim terem chegado ao patamar adequado de expressividade;

Equivalência dos modelos computacionais: já referimos a existência de vários formalismos que foram utilizados para caracterizar a noção de algoritmo. Para todos eles ficou demonstrado que tinham exactamente (nem *mais*, nem *menos*) o mesmo poder expressivo.

Eis alguns dos modelos computacionais que foram definido ao longo do século XX.

- Church (1936): cálculo λ (sem tipos) e funções lambda-definíveis;
- Turing (1936): máquinas de Turing;
- Gödel-Kleene (1936): funções recursivas parciais;
- Post (1943): Sistemas (formais) canónicos;
- Markov (1951): Sistemas canónicos determinísticos;
- Lambek e Minsky (1961): máquinas de registos;
- Scott (1969) funções monótonas e sistemas recursivos.

6. Indecidibilidade

Com os modelos apresentados, ficou formalizada a noção de algoritmo. Foi então possível demonstrar que havia problemas que nenhum algoritmo poderia resolver. São os problemas indecidíveis. O primeiro desses problemas é o problema de terminação (o famoso "halting problem"). Foi com a ajuda desse problema que Church e Turing demonstraram a impossibilidade do entcheidungsproblem.

O problema de terminação expressa-se da forma seguinte: Existirá um algoritmo capaz de avaliar em tempo finito se outro algoritmo, fornecido em entrada, termina ou não.

Surpreendentemente, a demonstração expressa-se *informalmente* de forma simples. Imaginemos que exista uma função `termina` que aceita um programa, digamos `p`, e que responderia em tempo finito `verdade` se o programa `p` termina e `falso` se o programa `p` não termina.

Consideremos agora o programa seguinte:

$$sem\_fim\ x \triangleq \text{se } x \text{ então } (sem\_fim\ x) \text{ senão } falso$$

`termina(sem_fim)` devolve então `falso`.

Consideremos também o programa seguinte:

$$teste \triangleq \text{se } (termina\ teste) \text{ então } teste \text{ senão } verdade$$

Que devolverá então `(termina teste)`?

Se `teste` não termina então `(termina teste)` devolve `falso` e o valor de `teste` é `verdade`, logo `teste` termina. Da mesma forma, se `teste` termina

então `(termina teste)` devolve `verdade` e `teste` é de novo avaliado (devido a recursão) e entra em ciclo. Temos aqui uma situação contraditória. A função `termina` não pode, infelizmente, existir: o problema da terminação é indecidível.

Desde o trabalho pioneiro dos grandes lógicos dos anos 30, uma grande variedade de problemas indecidíveis foram descobertos. O décimo problema de Hilbert que, relembremos, está na origem desta problemática toda, só encontrou a sua resolução em 1970 quando o matemático russo Matijasevic descobriu uma transformação que relacionava a resolução das equações diofantinas à resolução do problema de terminação. Matijasevic demonstrou que nenhum algoritmo era capaz de resolver as equações diofantinas de grau superior ou igual a 5.

É importante perceber que a indecidibilidade é uma barreira teórica inquebrável: se um problema é indecidível isso significa que nenhum algoritmo poderá resolver este problema, qualquer que seja a tecnologia e os recursos utilizados.

No entanto soluções parciais são possíveis:

- variantes do problema podem ser decidíveis (essas variantes podem ser simplificações). Por exemplo, o problema da satisfação em lógica de primeira ordem é indecidível, mas no entanto alguns subconjuntos são decidíveis como a lógica das claúsulas de Horn (em que a linguagem PROLOG está baseada) ou a lógica proposicional;

- o problema pode pertencer a classe dos problemas semi-decidíveis. Para tais problemas é possível encontrar algoritmos que sabem responder correctamente “sim” mas que não sabem responder “não”. A dificuldade no

uso de tais algoritmos é o facto de não obter resposta não significa que essa seja negativa: o tempo de computação de uma resposta positiva é arbitrário.

7. A herança dos primórdios da algoritmia

A definição dos primeiros modelos da computação foi o ponto de partida da disciplina chamada *Ciência da Computação*. Os seus primeiros objectivos foram a classificação dos problemas em função da facilidade da sua resolução. A primeira distinção fundamental é o caracter decidível ou não desses últimos. Já referimos que tal não é suficiente. É preciso igualmente conhecer a eficiência dos algoritmos de resolução. A definição de critérios de classificação pela eficiência deu lugar a teoria da complexidade computacional.

Para além dos aspectos técnicos, a mecanização dos cálculos e o controlo teórico que os modelos da computação possibilitaram levantou a pergunta fundamental da comparação da capacidade de cálculo duma máquina com a capacidade cálculo dum homem. Um forma sintética de reformular esta questão é: "Será o cérebro humano o equivalente de uma máquina de Turing?".

Uma primeira abordagem é perceber o quão mecânicos são as actividades do espírito, ou seja, será que as actividades de cálculo ou de raciocínio dos homens podem ser expressas em termos de algoritmos? Esta interrogação é a base de disciplinas como a *Filosofia do Espírito* ou as *Ciências Cognitivas* onde as operações mentais são formalizadas em termos de tratamento de informação.

Uma segunda abordagem consiste em averiguar se as máquinas podem (ou poderão) pensar. O próprio Turing contribui a resolução desta questão. Em 1950, numa revista de filosofia, publicou e argumentou que não havia razões de pensar que a resposta a tal questão fosse negativa. Para este propósito definiu o que se passou a chamar o teste de Turing e que tem a finalidade de determinar se uma máquina pensa ou não. Este teste consiste em colo-

car um "examinador" humano que enuncia perguntas a dois "interlocutores" anónimos em que um deles é uma máquina. Se o examinador é incapaz de determinar dos dois quem é a máquina então a máquina terá de ser considerada como inteligente. O projecto da disciplina *Inteligência Artificial*[3] é exactamente esse. Fornecer algoritmos, arquitecturas, ou de forma genérica, meios para que as máquinas possam realizar operações que até agora eram o privilégio dos homens: raciocínio lógico, aprendizagem, compreensão da linguagem, etc... Como nota final a esta abordagem, assinalemos que o próprio Alan Turing tinha previsto que as máquinas venceriam o seu teste no inicio deste novo milénio. É obvio dizer que longe estamos de tal meta. Aliás, filósofos, cientistas da cognição e neurólogos começam a contestar a pertinência dos modelos apresentado por Turing para explicar o funcionamento do espírito humano.

Outro problema, também ligado a questão inicial, é a capacidade de raciocínio das máquinas ou a mecanização da matemática. Sabemos que os resultados de completude, de decidibilidade garantem que as máquinas não são capaz de abranger a matemática na sua totalidade. No entanto partes importantes da matemática e do raciocínio matemática podem ser automatizada (no caso de serem decidíveis) ou pelo menos sistematizada (no caso de ser semi-decidíveis). Tal tarefa tem sido o alvo de disciplinas como a lógica computacional ou a teoria da demonstração.

[3] A disciplina foi criada em 1950. O próprio termo de "inteligência artificial" data de 1956.

8. Perceber e confiar nos programas e nas linguagens de programação

Em muitas áreas da engenharia, o processo de produção incluí uma fase rigorosa de validação/conformidade do produto. Por exemplo, não se constrói uma ponte sem que o respectivo projecto contemple uma fase *bem sucedida* de validação (cálculos de resistência do material, etc...).

Apesar da sua maciça difusão no dia a dia de qualquer cidadão, os sistemas informáticos (SIs), ou mais genericamente a informática, tem vindo a ser desenvolvidos, de forma surpreendente, sem uma disciplina rigorosa de validação que contempla por exemplo a correcção do software ou a garantia de determinados comportamentos dos programas (como os comportamentos ligados a segurança dum sistema informático)..

As ligações entre os modelos matemáticos da computação (e o cálculo λ em particular) e a informática que sublinhamos até agora não têm só implicações teóricas e fundamentais, tem também implicações bem pragmáticas: contribuíram a construção da sugerida disciplina.

Nessa vertente, podemos classificar as várias abordagens que daí resultaram em duas famílias:

1. estabelecer linguagens de programação cujos componentes de base sejam suficientemente bem fundados para garantir uma programação fiável;

2. fornecer meios para perceber, modelar e justificar formalmente linguagens e programas já existentes.

Podemo-nos questionar sobre o interesse de tal esforço. Assim, com o

intuito de ilustrar a relevância de tal disciplina e de justificar a escolha do cálculo λ como modelo de computação central nesta disciplina, apresentaremos a seguir uma breve justificação dos objectivos de cada uma das referidas famílias.

8.1. Um passeio a volta da noção de linguagens de programação

Recentemente, a constituição duma lista das linguagens de programação permitiu recensear mais de duas mil linguagens concebidas em apenas cinquenta anos (desde a aparição da linguagem Fortran, concebida em 1954 pela equipa de John Backus). Mesmo se não podemos por todas essas linguagens ao mesmo nível, a informática não deixa de ter a imagem de uma torre de Babel pouca acolhedora.

Esta proliferação custa cara à industria do software porque:

- torna difícil a realização de sistemas integrando módulos previamente desenvolvidos em linguagens diferentes;
- obriga a construção de interfaces, módulos de ligação;
- torna complicada a manutenção desses sistemas híbridos.

Na primeira linha dos lesados, o ministério da defesa americano lançou em 1978 um concurso com a finalidade de escolher uma linguagem suficientemente bem concebida para ser universal e de impô-la como linguagem de desenvolvimento de todo o software que aí venha a ser produzido. Este concurso, vencido

no ano seguinte pela equipa de Jean Ichbiah, permitiu a criação da linguagem ADA que é, hoje em dia, bastante popular, mas que falhou na sua missão de universalidade.

No entanto, no domínio restrito das linguagens de programação, este sonho de linguagem universal não parece ser totalmente utópico. A grande maioria dos músicos utilizam a mesma linguagem para escrever partituras. De forma semelhante, os matemáticos de todo o mundo utilizam uma linguagem comum. Obviamente, a matemática não é uma linguagem rígida, antes pelo contrário, e cada matemático tem o seu estilo próprio. Se, no domínio das linguagens de programação, não se conseguiu tal unidade, é provavelmente porque ainda não percebemos completamente o que é um programa ou uma linguagem de programação.

Já referimos que podemos ver um programa como um meio de especificar quais transformações é preciso efectuar sobre os dados de entrada para obter um determinado resultado. Assim, numa visão genérica, um programa não é nada mais do que uma função e uma linguagem de programação nada mais do que uma linguagem de definição de funções.

Então, se a actividade de programar se pode resumir à definição de funções apropriadas, porquê criar novas linguagens se a própria linguagem matemática tem vindo a desempenhar perfeitamente este papel? As primeiras respostas a esta questão vieram mais uma vez dos esforços dos pioneiros da ciência da computação.

Se nos debruçarmos mais particularmente sobre os mecanismos de definição de funções na matemática veremos que podem assumir a forma seguinte:

- estabelecer o domínio da função por definir;

- definir um conjunto de equações que especificam as propriedades que a função deve verificar;
- demonstrar que existe uma e uma só função que verifica a propriedade em questão;
- atribuir um nome a função que as equações representam.

Em termos formais, os matemáticos utilizam o "operador de descrição" $fun \triangleq [f \mid P(f)]$[4] para representar definições de funções. Neste caso $[f \mid P(f)]$ é a única função f verificando a propriedade P e $fun \triangleq [f \mid P(f)]$ significa fun é definida como sendo esta função.

Por exemplo

$$x^n \triangleq [f \mid (\forall x.\ f(x,0) = 1) \wedge (\forall x, n.\ f(x, n+1) = x \times f(x,n))]$$

Qual é a vantagem duma tal abordagem? Se nos debruçarmos sobre a definição de x^n, a função resultante é uma função da qual podemos ter a certeza que calcula x elevado a n porque tivemos de demonstrar a sua existência e a sua correcção em relação a propriedade $P \triangleq (\forall x.\ f(x,0) = 1) \wedge (\forall x, n.\ f(x, n+1) = x \times f(x,n))$. Utilizar o operador de descrição para definir programas é programar sem bugs... o Eldorado da programação!

Infelizmente é preciso saber executar tais programas, e já sabemos que a crise dos fundamentos da matemática estabeleceu fronteiras firmes entre a matemática e os algoritmos. A consequência, neste caso concreto, é que as

[4] O símbolo $\triangleq$ representa uma definição. Assim $a \triangleq b$ significa que o símbolo a é definido como sendo o objecto matemático b.

máquinas não são capazes de abranger o operador de descrição na sua globalidade[5]. Assim uma linguagem de programação é um compromisso entre a expressividade desejada do operador e a capacidade das funções definidas em serem executadas. As linguagem de programação, qualquer que sejam elas (C, Pascal, Java, Prolog, ADA, etc...), disponibilizam então ao programador versões restritas do operador de descrição. Citemos no entanto algumas linguagens qualificadas de funcionais: Haskell (em honra do lógico Haskell Curry, que desenvolveu investigações fundamentais e pioneira na área do cálculo lambda) e OCaml. Essas linguagens de programação são baseadas sobre extensões do cálculo λ. Falaremos também, mais abaixo, de outras extensões deste cálculo que têm, ao contrário do Haskell e do OCaml, ligações directas com a matemática construtiva.

Um dos objectivos da presente disciplina será estudar o operador de descrição, em particular a sua relação com os modelos da computação. Por exemplo, os CPOs, a teoria dos pontos fixos conjuntamente com a teoria das funções monótonas (e contínuas) fornecem um fundamento unificador aos ciclos e a recursividade.

Ao início do século XX, uma escola de matemáticos, chamados intuicionistas ou construtivistas, criticaram fortemente e recusaram o uso do axioma do absurdo (e a suas formas equivalentes, como o terço excluído $P \vee \neg P$) que era no entanto utilizado em muitas demonstrações cruciais da matemática. A matemática resultante, a matemática construtiva, tem a propriedade interessante de ter uma relação muito forte entre demonstração e tratamento efectivo. De facto, uma demonstração é uma construção, isto é, um método para construir

[5]Este facto é uma consequência directa da indecidibilidade do problema de decisão.

um objecto matemático que testemunha que a propriedade é válida. A Luitzen Egbertus Jan Brower, líder da escola intuicionista, juntaram-se matemáticos famosos como Hermann Weyl ou Andreï Kolmogorov.

Stephen Kleene demonstrou que quando o operador de descrição é restringido a matemática construtiva (i.e. quando se abandona o terço excluído), as funções que se podem expressar são exactamente as funções computáveis. Dito de outra forma, quando o operador de descrição é restrito à matemática construtiva, é possível encontrar regras de cálculo que permitam a execução de qualquer função definida. Acontece que variantes do cálculo λ original que vamos estudar nesta disciplina são candidatos adequados para definir mecanismos de definição de funções e as sugeridas regras de cálculo.

Vemos assim surgir uma nova metodologia de programação: começa-se por especificar a função f que queremos programar definindo a propriedade P que deverá verificar. Constroí-se a seguir uma demonstração construtiva p de que existe uma e uma só função verificando P. Todas as componentes estão então presentes para definir $[f \mid P(f)]$. A execução do programa consiste em interpretar as regras de cálculo tendo em conta $[f \mid P(f)]$ e a demonstração construtiva p. A vantagem é que obtemos um programa "zero-defeito".

Em termos concretos, grande avanços teóricos permitiram o aparecimento de várias linguagens ou sistemas seguindo esta abordagem. Essas investigações permitiram descobrir que extensões ao cálculo lambda serviam de forma adequada de base a este tipo de programação. Citemos por exemplo o lógico alemão Gerhard Gentzen que propôs em 1934 as primeiras regras de execução num formalismo construtivo. O lógico francês Jean-Yves Girard propôs em 1971 novas regras de execução num cálculo mais geral e mais poderoso. Este passo foi essencial para permitir extensões ao cálculo λ que permitisse

abranger cada vez mais a matemática construtiva na sua globalidade. Assim o lógico sueco Per Martin-Löf propôs em 1973 a teoria dos tipos intuicionista e os lógicos franceses Gérard Huet e Thierry Coquand extenderam-na em 1985 com o cálculos das construções. Baseados nesses formalismos, linguagens ou sistemas como o AF2 (criado por Jean-Louis Krivine e Michel Parigot em 1987) ou Coq (desenvolvido por Gérard Huet, Thierry Coquand, Gilles Dowek, Christine Paulin a partir de 1985) foram desenvolvidos.

Em conclusão, contentar-se em propor uma visão unificadora das linguagens de programação não é razoavelmente suficiente. Qual é o estatuto das linguagens e programas existentes? Não podemos esquecê-los, obviamente. Até porque muitos programas e muitas linguagens têm e terão um papel crucial. Esta problemática é longe de ser estéril, antes pelo contrário. De facto as consequências não tiveram só implicações na informática, também tiveram na matemática.

8.2. Demonstração matemática e programa

Em 1998, Thomas Hales, matemático da universidade de Pittsburgh, entregou uma demonstração da conjectura de Kepler baseada num programa de computador que verificava extensivamente todas as configurações próprias ao problema. Esta demonstração junta-se a mesma família da demonstração do teorema das quatros cores (também realizada com a ajuda de computador). No entanto, retomou-se com este acontecimento o velho debate *"o que é uma demonstração?"*.

Classicamente em matemática, uma demonstração é uma argumentação considerada válida se resistir a *inspecção* da comunidade científica. Dito por

outra palavras, se houver consenso pela parte dos matemáticos então a demonstração é considerada válida. Neste sentido, Hales submeteu a demonstração para uma prestigiada revista matemática que achou bom que este tipo de trabalho, uma demonstração por computador, fosse analisado por um número invulgarmente grande de avaliadores. O problema aqui para os avaliadores não era analisar uma demonstração de papel e caneta composta de argumentos matemáticos expressos em língua natural (inglês, neste caso) mas sim analisar uma demonstração em grande parte constituída por um programa. Aqui, o processo social de validação tem de contemplar a correcção dos princípios matemáticos que levaram à construção do programa e a completude e correcção do próprio programa (o programa faz exactamente o que se pretende, nem mais nem menos). É importante aqui realçar que por programa entende-se todo o sistema que gera a demonstração: o programa fonte, o compilador, o programa máquina e a arquitectura física onde se executa o programa. Um bug em qualquer um desses pontos condena a demonstração.

Um matemático pode convencer-se (ou não) naturalmente com argumentos da primeira natureza, e alias, é formado para isso, mas ainda não prática comum nem consenso ter de se convencer por linhas de código máquina.

A avaliação, que chegou quatro anos mais tarde, ilustrou a polémica que levantou a última parte do processo de validação:

> "The news from the referees is bad from my perspective. They have not been able to certify the correctness of the proof, and will not be able to certify it in the future, because they have run out of energy to devote to the problem. This is not what I had hoped for."

(Comunicação de McPherson, editor de Annals of Matematics para Thomas Hales)

Desta anedota podemos extrair duas lições que o próprio Hales contemplou no seu programa de investigação. É relevante :

1. fundamentar matematicamente os sistemas informáticos e mais particularmente, no que nos diz respeito, as linguagens de programação. Tal é o objectivo da disciplina de "semântica das linguagens de programação";

2. ter uma forma fiável de construir e verificar demonstrações por computador. Várias disciplinas, algumas delas já referidas, partilham este objectivo: lógica computacional, matemática construtiva, demonstração automática e assistida por computador.

O cálculo λ tem um papel central nestas disciplinas. Um exemplo que já referimos é a descoberta importante em matemática construtiva do Isomorfismo de Curry-Howard que liga um termo λ a uma demonstração construtiva: programar no cálculo λ é igual a demonstrar! No entanto, iremo-nos concentrar mais na questão do lugar do cálculo λ no estudo das linguagens de programação.

Mais uma vez aqui, o cálculo λ tem um papel importante:

- porque serve de base ao estudo e à implementação de mecanismos incluídos em tais linguagens de programação;

- porque é utilizado como linguagem subjacente a várias semânticas de linguagens de programação e vários sistemas de métodos formais .

8.3. Construção de linguagens de programação avançadas

De facto, o cálculo λ é uma linguagem de programação simples mas dispõe da riqueza expressiva necessária para servir de cobaia privilegiada aos cientistas da programação. Actualmente os maiores avanços nas linguagens de programação tiveram origem em extensões do cálculo λ.

8.4. Modelação e verificação formal de programas

Contextualizamos com a demonstração da conjectura de Kepler a relevância da disciplina da semântica das linguagens como disciplina de modelação formal das linguagens de programação: trata-se aqui de ter um controlo formal, matemático das linguagens e dos programas. Embora tenhamos argumentado este facto com uma ilustração de foro matemático, existem inúmeras justificações a modelação e verificação formal de programas. Por exemplo, o NIST (instituto norte americano para as novas tecnologias) avaliou o custo dos bugs dos sistemas informáticos na economia americana em 59.5 biliões de dólares. Mas um erro num SI pode implicar mais do que um negativo impacto económico (vidas humanas, segurança informática, confiança dos clientes, etc...). A falta de uma disciplina rigorosa de construção de SI levou W. Gibbs a declarar em 1994 num artigo da revista *Scientific American* intitulado "Trends in Computing: Software's Chronic Crisis"

> "Despite 50 years of progress, the software industry remains years – perhaps decades – short of the mature engineering discipline needed to meet the needs of an information-age society."

Os métodos formais, nos quais podemos integrar as semânticas das linguagens, são uma resposta ao referido problema de falta de maturidade.

Para concluir podemos citar o exemplo de ADA e de OCaml como linguagens que foram definidos com a preocupação de terem fundamentos rigorosos que lhes permitam, por exemplo, um controlo matemático. Ambas dispõem duma semântica clara e completa.

Referências

[1] G. Dowek. *La logique*. Coll. Dominos. Flammarion, 1995.

[2] G. Dowek. Le langage mathématique et les langages de programmation. Voir, entendre, raisonner, calculer. Cité des sciences et de l'industrie, La Villette, Paris, 1997. `http://www.lix.polytechnique.fr/~dowek/Vulg/langagelangages.ps.gz`

[3] D. R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, New York, 1979.

[4] W. Kneale and M. Kneale. *O Desenvolvimento da Lógica*. Fundação Calouste Gulbenkian, (1962) - 1991. Terceira Edição.

[5] A. Pitts. Lecture notes on computation theory. University of Cambridge Computer Laboratory, 2002. `http://www.cl.cam.ac.uk/Teaching/2002/CompTheory`

[6] Isabelle Tellier. Introduction à l'informatique. `http://www.grappa.univ-lille3.fr/polys/intro-info/index.html`, 2001.