Maribel Fernández

Models of Computation

An Introduction to Computability Theory

Springer

1
Introduction

This book is concerned with abstract models of computation. Several new models of computation have emerged in the last few years (e.g., chemical machines, bio-computing, quantum computing, etc.). Also, many developments in traditional computational models have been proposed with the aim of taking into account the new demands of computer system users and the new capabilities of computation engines. A new model of computation, or a new feature in a traditional one, usually is reflected in a new family of programming languages and new paradigms of software development. Thus, an understanding of the traditional and emergent models of computation facilitates the use of modern programming languages and software development tools, informs the choice of the correct language for a given application, and is essential for the design of new programming languages.

But what exactly is a "model of computation"? To understand what is meant by a model of computation, we briefly recall a little history. The notions of computability and computable functions go back a long time. The ancient Greeks and the Egyptians, for instance, had a good understanding of computation "methods". The Persian scientist Al-Khwarizmi in 825 wrote a book entitled "On the Calculation with Hindu Numerals", which contained the description of several procedures that could now be called algorithms. His name appears to be the origin of the word "algorithm": When his book was translated into Latin, its title was changed to "Algoritmi de Numero Indorum". The word "algorithm" was later used to name the class of computation procedures described in the book. Roughly, an *algorithm* is:

– a finite description of a computation in terms of well-defined elementary operations (or instructions);

– a deterministic procedure: the next step is uniquely defined, if there is one;

– a method that always produces a result, no matter what the input is (that is, the computation described by an algorithm always terminates).

The modern computability theory has its roots in the work done at the beginning of the twentieth century to formalise the concept of an "algorithm" without referring to a specific programming language or physical computational device. A computation model abstracts away from the material details of the device we are using to make the calculations, be it an abacus, pen and paper, or our favourite programming language and processor.

In the 1930s, logicians (in particular Alan Turing and Alonzo Church) studied the meaning of computation as an abstract mental process and started to design theoretical devices to model the process of computation, which could be used to express algorithms and also non-terminating computations.

The notion of a *partial function* generalises the notion of an algorithm described above by considering computation processes that do not always lead to a result. Indeed, some expressions do not have a value:

1. $True + 4$ is not defined (we cannot add a number and a Boolean).
2. $10/0$ is not defined.
3. The expression $\mathsf{factorial}(-1)$ does not have a value if $\mathsf{factorial}$ is a recursive function defined as follows:

$$\begin{array}{lcl} \mathsf{factorial}(0) & = & 1 \\ \mathsf{factorial}(n) & = & n * \mathsf{factorial}(n-1) \end{array}$$

The first is a type error since addition is a function from numbers to numbers: For any pair of natural numbers, the result of the addition is defined. We say that addition is a *total function* on the natural numbers.

The second is a different kind of problem: 10 and 0 are numbers, but division by 0 is not defined. We say that division is a *partial function* on the natural numbers.

There is another case in which an expression may not have a value: The computation may loop, as in the third example above. We will say that $\mathsf{factorial}$ is a *partial function* on the integers.

The notion of a partial function is so essential in computability theory that it deserves to be our first definition.

Definition 1.1 (Partial function)

Let A and B be sets. We denote their Cartesian product by $A \times B$; that is, $A \times B$ denotes the set of all the pairs where the first element is in A and the second in B. We use the symbol $\in$ to denote membership; i.e., we write $a \in A$ to indicate that the element a is in the set A.

A *partial function* f from A to B (abbreviated as $f : A \to B$) is a subset of $A \times B$ such that if $(x, y) \in f$ and $(x, z) \in f$, then $y = z$. In other words, a partial function from A to B associates to each element of A at most one element of B.

If $(x, y) \in f$, we write $f(x) = y$ and say that y is the *image* of x. The elements of A that have an image in B are in the *domain* of f.

In the study of computability, we are often interested only in functions whose domain and co-domain are the set of integer numbers. In some cases, this is even restricted to natural numbers; that is, integers that are positive or zero.

The notion of a partial function is also important in modern programming techniques. From an abstract point of view, we can say that *each program defines a partial function*. In practice, we are interested in more than the function that the program computes; we also want to know how the function is computed, how efficient the computation is, how much memory space we will need, etc. However, in this book we will concentrate on whether a problem has a computable solution or not, and how the actual computation mechanism is expressed, without trying to obtain the most efficient computation.

1.1 Models of computation

Some mathematical functions are computable and some are not: There are problems for which no computer program can provide a solution even assuming that the amount of time and space available to carry out the computation is infinite. Complexity theory studies the "practical" aspects of computability; that is, for a computable function, it answers the question: How much time and space will be needed for the computation? We will not cover complexity theory in this book but instead will concentrate on computability.

First we need to define precisely the notion of a computable function. This is a difficult task and is still the subject of research. We will first give an intuitive definition.

Definition 1.2 (Computable function)

All the functions on the natural numbers that can be effectively computed in an ideal world, where time and space are unlimited, are called *partial recursive functions* or *computable functions*.

The definition of a computable function above does not say what our notion of "effective" computation is: Which programming language is used to define the function? What kind of device is used to compute it? We need a *model of computation* to abstract away from the material details of the programming language and the processor we are using. In fact, computability was studied as a branch of mathematical logic well before programming languages and computers were built. Three well-studied abstract models of computation dating from the 1930s are

- *Turing machines*, designed by Alan Turing to provide a formalisation of the concept of an algorithm;
- the *Lambda calculus*, designed by Alonzo Church with the aim of providing a foundation for mathematics based on the notion of a function; and
- the *theory of recursive functions*, first outlined by Kurt Gödel and further developed by Stephen Kleene.

These three models of computation are equivalent in that they can all express the same class of functions. Indeed, *Church's Thesis* says that they compute all the so-called computable functions. More generally, Church's Thesis says that the *same* class of functions on the integers can be computed in any sequential, universal model of computation that satisfies basic postulates about determinism and the effectiveness of elementary computation steps. This class of *computable functions* is the set of partial recursive functions.

We say that a programming language is *Turing complete* if any computable function can be written in this language. All general-purpose programming languages available nowadays are complete in this sense. Turing completeness is usually proved through an encoding in the programming language of a standard universal computation model.

1.2 Some non-computable functions

Since the 1930s, it has been known that certain basic problems cannot be solved by computation. The typical example is the Halting problem discussed below,

which was proved to be non-computable by Church and Turing. Other examples of non-computable problems are:

– Hilbert's 10th problem: solving Diophantine equations.

  Diophantine equations are equations of the form

$$P(x_1, \ldots, x_n) = Q(x_1, \ldots, x_n)$$

  where P and Q are polynomials with integer coefficients. A polynomial is a sum of monomials, each monomial being a product of variables with a coefficient. The coefficients are constants; for example, $x^2 + 2x + 1$ is a polynomial on one variable, x.

  The mathematician David Hilbert asked for an algorithm to solve Diophantine equations; that is, an algorithm that takes a Diophantine equation as input and determines whether this equation has integer solutions or not. This problem was posed by Hilbert in 1900 in a list of open problems presented at the International Congress of Mathematicians, and it became known as Hilbert's 10th problem. It is important to note that the coefficients of the polynomials are integers and the solution requested is an assignment of integer numbers to the variables in the equation.

  Hilbert's 10th problem remained open until 1970, when it was shown to be undecidable in general by Yuri Matijasevič, Julia Robinson, Martin Davis, and Hilary Putnam.

– Hilbert's *decision problem*: the *Entscheidungsproblem*.

  This problem was also posed by Hilbert in 1900. Briefly, the problem requires writing an algorithm to decide whether any given mathematical assertion in the functional calculus is provable.

  Hilbert thought that this problem was computable, but his conjecture was proved wrong by Church and Turing, who showed that an algorithm to solve this problem could also solve the Halting problem.

These are examples of *undecidable problems*. We end this introduction with a description of the Halting problem.

The Halting problem. Intuitively, to solve the Halting problem, we need an algorithm that can check whether a given program will stop or not on a given input. More precisely, the problem is formulated as follows:

Write an algorithm H such that given

– *the description of an algorithm A (which requires one input) and*

– *an input* I*,*

H *will return* 1 *if* A *stops with the input* I *and* 0 *if* A *does not stop on* I*.*

We can see the algorithm H as a function: $H(A, I) = 1$ if the program A stops when the input I is provided, and $H(A, I) = 0$ otherwise.

In the quest for a solution to this problem, Turing and Church constructed two abstract models of computation that later became the basis of the modern theory of computing: Turing machines and the Lambda calculus.

In fact, Church and Turing proved that there is no algorithm H such that, for any pair (A, I) as described above, H produces the required output. Its proof, which follows, is short and elegant.

Proof

If there were such an H, we could use it to define the following program C:

> C *takes as input an algorithm* A *and computes* $H(A, A)$*. If the result is* 0*, then it answers* 1 *and stops; otherwise it loops forever.*

Below we will use the notation $A(I)\uparrow$, where A is a program and I is its input, to represent the fact that the program A does not stop on the input I. Using the program C, for any program A, the following properties hold:

– If $H(A, A) = 1$, then $C(A)\uparrow$ and $A(A)$ stops.

– If $H(A, A) = 0$, then $C(A)$ stops and $A(A)\uparrow$.

In other words, $C(A)$ stops if and only if $A(A)$ does not stop.

Since A is arbitrary, it could be C itself, and then we obtain a contradiction:

$$C(C) \text{ stops if and only if } C(C) \text{ does not stop.}$$

Therefore H cannot exist.

□

The proofs of undecidability of Hilbert's decision problem or Diophantine equations are more involved and we will not show them in this book, but it is important to highlight that these results, obtained with the help of abstract models of computation, still apply to current computers.

Since the class of computable functions is the same for all the traditional computation models, we deduce that imperative or functional languages (which are based on Turing machines and the Lambda calculus, respectively) can describe exactly the same class of computable functions. Several other models

of computation, or idealised computers, have been proposed, some of them inspired by advances in physics, chemistry, and biology. There is hope that some of these new models might solve some outstanding non-feasible problems (i.e., problems that cannot be solved on a realistic timescale in traditional models).

1.3 Further reading

Readers interested in algorithms can find more information in Harel and Feldman's book [22]. Further information on partial functions and computability in general can be found in [47, 49] and in the chapter on computability in Mitchell's book [36]. Additional references are provided in the following chapters.

1.4 Exercises

1. Give more examples of total and partial functions on natural numbers.
2. To test whether a number is even or odd, a student has designed the following function:

   ```
   test(x) def=  if x = 0 then "even"
                 else if x = 1 then "odd" else test(x-2)
   ```

   Is this a total function on the set of integer numbers? Is it total on the natural numbers?
3. Consider the following variant of the Halting problem:

   > *Write an algorithm H such that, given the description of an algorithm A that requires one input, H will return 1 if A stops for any input I and H will return 0 if there is at least one input I for which A does not stop.*

   In other words, the algorithm H should read the description of A and decide whether it stops for all its possible inputs or there is at least one input for which A does not stop.

   Show that this version of the Halting problem is also undecidable.