Chapter 3
Foundations

In this chapter, we will not be concerned with programming languages but with the limits of the programs that we can write, asking whether there exist problems that no program can solve. A motivation for this research is the question that we asked at the end of Sect. 2.3: that is, is it possible to construct a static semantic analyser which can verify constraints imposed by the programming language's definition? We will soon discover, however, that the answer to the question is rather more general and is, in reality, a kind of absolute limit to what can (and cannot) be done with a computer. Although the material in this chapter can appear abstract, our treatment is wholly elementary.

3.1 The Halting Problem

In Sect. 2.3, we asked if there exists a static semantic analyser able to determine whether a program can generate a division by zero error during execution. Instead of tackling this problem, we will examine a larger problem, one that is also more interesting. We will ask whether there exists a static analyser able to discover whether a program, when provided with certain input data, will loop. There is no need to emphasise the usefulness of a check of this kind. If we know, prior to execution, that a given program will loop when presented with some input data, we will have a way of showing automatically that it definitely contains an error.

A static analyser is nothing more than a program used as a subprogram of a compiler. Let us, therefore, fix a programming language, $\mathscr{L}$, in which we will write programs. Given a program, P, and an input x, we write $P(x)$ to denote the result of the computing P on x. We should note that, in general, the program P might not terminate on x because of a loop or of an infinite recursion. Writing $P(x)$, therefore, we do not necessarily indicate a computation that terminates. Without loss of generality, we can now reformulate the question as:

> Does there exist a program, H, that, having been given as input a program P (in the language $\mathscr{L}$) and its input data x, will terminate and print "yes" if $P(x)$ terminates, and terminate and print "no" if $P(x)$ loops?

We emphasise that H has two inputs. It must always terminate (we do not want a compiler that loops!) and it must function on every program, P, written in $\mathscr{L}$ and every input x. We can then assume, again without loss of generality, that H is written in the language $\mathscr{L}$, since $\mathscr{L}$, to be interesting, must be in a language in which it is possible to write all possible programs.[1]

We now show our hand. We want to show that there exists no program H with the behaviour we have just discussed. We will argue by contradiction, assuming we can have such an H, and from this we will derive a contradiction. The reasoning might seem a little contorted at first sight but requires no advanced knowledge. The argument, however, is subtle and elegant; the reader should read it more than once and be sure of understanding the critical role of self application.

1. Let us assume, by contradiction, that we have a program H with the properties stated above.
2. Using H, we can write a program K, with only one input (which will be another program), with the following specification:

   > The program K, given P as input, terminates printing "yes" if $H(P, P)$ prints "no"; it goes into a loop if $H(P, P)$ prints "yes".

   Writing the program K, given that H is available, is simple. We read the input P, call H as a subprogram, passing P as first and second parameter. We wait until H terminates (this will certainly happen, given H's specification). If $H(P, P)$ terminates and prints "no" (we can assume that we can intercept this printing and stop it from appearing on the output), K prints "yes". If, on the other hand, $H(P, P)$ prints "yes", then K goes into an infinite loop programmed for the purpose.

   If we recall the specification of H, we can summarise the semantics of K as:

   $$K(P) = \begin{cases} \text{"yes"} & \text{if } P(P) \text{ does not terminate,} \\ \text{does not terminate} & \text{if } P(P) \text{ does terminate.} \end{cases} \tag{3.1}$$

   At first sight the application of P to itself seems strange. There is no miracle. P will receive an input consisting of a string representing the text of P.[2]
3. Let us now execute K on its own text. That is, we focus on $K(K)$. What is its behaviour? If we substitute K for P in (3.1), we obtain:

   $$K(P) = \begin{cases} \text{"yes"} & \text{if } K(K) \text{ does not terminate,} \\ \text{does not terminate} & \text{if } K(K) \text{ does terminate.} \end{cases} \tag{3.2}$$

4. Now let us observe that (3.2) is absurd! It says that $K(K)$ terminates (printing "yes") when $K(K)$ fails to terminate and that it does not terminate when $K(K)$ terminates.

[1] Clarifying the sense of "all possible programs" is actually one of the aims of this chapter.

[2] If we had assumed that the input to P was a number, the string comprising the text of P can read as a number: a number i, whose bits denote the individual characters in the text of P.

5. Where does this contradiction come from? Not from K itself, which, as we have seen, is a simple program which uses H. The contradiction arises from having to assume the existence of a program with the properties of H. Therefore, H cannot exist.

We have therefore proved that there exists no *decision procedure* (in the language $\mathscr{L}$) capable of determining whether any another program in $\mathscr{L}$ terminates on an arbitrary input. We use here decision procedure in the technical sense of a program that: (i) works for arbitrary arguments; (ii) always terminates and (iii) determines (by responding "yes" or "no") those arguments which are solutions to the problem and those which are not.

This result, one of fundamental importance in computing, is called the *undecidability of the halting problem*. Many other interesting problems are undecidable in the same way. We will discuss some in the next section after first having discussed the characteristics of $\mathscr{L}$ from which the result derives. We can thus tackle the problem of the expressive power of programming languages.

3.2 Expressiveness of Programming Languages

At first sight, the result we obtained in the last section can appear fairly limited. If we take a language different from $\mathscr{L}$, perhaps the program H can exist without generating a contradiction.

Upon reflection, though, we have not assumed much about $\mathscr{L}$. We have used $\mathscr{L}$ in an implicit way to define the program K, given H (that is, at Step 2 of the proof). To be able truly to write K, the following conditions must be satisfied:

1. There must be some form of conditional available in $\mathscr{L}$, so that the cases in the definition of K can be distinguished;
2. It must be possible to define functions which do not terminate in $\mathscr{L}$ (we must therefore have at our disposal some form of iteration or recursion).

At this level of detail, $\mathscr{L}$ is nothing special. What programming language does not provide these constructs? If a language provides these constructs, it can be used in place of $\mathscr{L}$ in the proof, showing that a program like H exists in *no* programming language worth its salt.

The undecidability of the halting problem is not, therefore, a contingent fact, related to any particular programming language, nor is it the expression of our inability as programmers. On the contrary, it is a limitation that is in some way absolute, indissolubly linked to the intuitive concepts of program (algorithm) and with that of programming language. It is a principle of nature, similar to the conservation of energy or to the principles of thermodynamics. In nature, there exist more problems and functions than there are programs that we can write. There are problems to which there correspond no program that is anything but insignificant; the halting problem is definitely one of them.

Rather, as argued in Sect. 3.3, the problems for which there exists a program for their solution constitute only a tiny part of the set of all possible problems.

As with the halting problem, when we say that some problem is undecidable, we mean that there exists no program such that: (i) it accepts arbitrary arguments; (ii) it always terminates and (iii) it determines which arguments are solutions to the problem and which are not. We will stray too far if we begin a proof of the undecidability of any other important problems; the interested reader can consult any good text on the theory of computability for them. We can however list some undecidable problems without attempting to explain all the terms used or to undertake any proofs.

The following problems are undecidable:

- Determine whether a program computes a constant function;
- Determine whether two programs compute the same function;
- Determine whether a program terminates for every input;
- Determine whether a program diverges for every input;
- Determine whether a program, given an input, will produce an error during its execution;
- Determine whether a program will cause a type error (see the box on page 204).

Undecidability results tell us that they do not exist general software tools which will automatically establish significant properties of programs.

3.3 Formalisms for Computability

To make the discussion of $\mathscr{L}$ more precise, we need to fix a programming language and show that the reasoning of the previous section applies to it. Historically, the first language in which the impossibility of writing program H was shown was the language of the Turing Machine. The Turing Machine language is a notation which is, at first sight, highly rudimentary. It was introduced in the 1930s by the mathematician Alan M. Turing. It is summarised in the *Turing Machines* box. It is at first sight surprising that such a rudimentary formalism (there is no predefined arithmetic; everything turns on the positioning of a finite number of symbols on a tape) could be good enough to express computations as sophisticated as those needed to write K on the basis of H. More surprising is the fact that there exists a Turing Machine which can act as an interpreter for others; that is a machine which, once an input has been written on its own tape, as well as the (appropriately coded) description of a generic machine and an input for it, executes the computation which that machine would perform on that input. This interpreter is, to all intents and purposes, a (very simple) computer like those that we know today (program and data in memory, fundamental cycle which interprets program instructions).

A function is *computable in a language* $\mathscr{L}$ if there exists a program in $\mathscr{L}$ which computes it. More precisely, the (partial—see the box on page 12) function $f : A \to B$ is computable if there exists a program P with the following properties: for each element $a \in A$, whenever P is executed on an input a, which we write $P(a)$, it terminates providing as an output $f(a)$ if $f(a)$ is defined; the computation $P(a)$ does not terminate if f is undefined at a.

Turing Machines

A Turing machine is composed of an infinite tape, divided into cells, each of which stores a single symbol from a finite alphabet. A mobile head reads from and writes to the tape. At any time, the head is positioned over a single cell. The machine is controlled by a finite-state controller. At each step in the computation, the machine reads a symbol from the tape and, according to the state of the machine and symbol that it has read, the controller decides which symbol to substitute for it on the tape and if the head is to move to the left or to the right; the controller therefore enters another state (remember, there is a finite number of states). The controller of a Turing machine can be seen as its associated program.

We might expect there to be fewer functions computed by a Turing Machine than those which can be computed by a sophisticated, modern programming language. In investigating this question, there have been many suggestions since the 1930s formalisms for expressing algorithms (programs), amongst which there are the General Recursive Functions of Church-Gödel and Kleene (which make no reference to programming languages), the Lambda Calculus (which we will examine in Chap. 11), and then all current programming languages. Indeed, all these formalisms and languages can be simulated by each other. That is, it is possible in each of these formalisms to write an interpreter for any of the others. From this, it follows that they are all equivalent in terms of the functions that they compute. They all compute exactly the same functions as the Turing Machine. In principle, therefore, every algorithm is expressible in any programming language. This is frequently expressed by saying that *all programming languages are Turing complete* (or *Turing equivalent*). The undecidability results can also be expressed by saying that there exist functions which cannot be computed (with a Turing Machine, or, by the above, in any programming language).

If all languages are equivalent with respect to the functions they compute, it is clear that they are not equivalent as far as their flexibility of use, pragmatics, abstraction principles and so on, are concerned. And often this complex of properties is referred to as the *expressiveness* of a language.

While all these computability formalisms provide a definite result about equivalence in terms of the functions that can be expressed, these same formalisms are useless when discussing the expressiveness of languages. In fact, even now, there is no agreement on how formally to tackle these aspects.

3.4 There are More Functions than Algorithms

In Sect. 3.1, we gave a specific example of a uncomputable function. We can give another proof that there exist functions which cannot be computed using a simple cardinality argument (though, unlike in Sect. 3.1, we will not produce a specific example). That is, we show that, in nature, there are more functions than algorithms.

First of all, let us consider any formal system which allows us to express algorithms and which, for simplicity we can assume to be a programming language, $\mathscr{L}$, (this could, however, be generalised to a wider definition). It is easy enough to see that the set of all possible (finite) programs that can be written in $\mathscr{L}$ is *denumerable*, that is, they can be put into one-one correspondence with the natural numbers (which we denote by $\mathbb{N}$). We can, in fact, first consider the set P_1 containing all programs of length one (which contain a single character), then the set P_2 containing all programs of length two, and so on. Every set, P_i, is finite and can be ordered, for example lexicographically (first all programs which begin with a, then all those which begin with b, and so on by succeeding characters). It is clear that by doing this, by taking into account the ordering produced by the subscript in the sets $P_1, P_2, \ldots$, and then doing the same with the internal indices in each set, we can count (or enumerate) all possible programs and therefore put them into one-one correspondence with the natural numbers. In more formal terms, when this has been done, we can say that the cardinality of the set of all programs writable in $\mathscr{L}$ is equal to the cardinality of the naturals. Let us now consider the set $\mathscr{F}$ containing all the functions $\mathbb{N} \to \{0, 1\}$. An important theorem of Cantor states that this set is not denumerable but has a cardinality which is strictly greater than that of $\mathbb{N}$. Given that every program expresses a unique function, the set $\mathbb{N}$ is too small to contain all the functions in $\mathscr{F}$ as programs in $\mathscr{L}$.

Let us see a direct proof of the fact that $\mathscr{F}$ is not denumerable. Let us assume that $\mathscr{F}$ is denumerable, that is it is possible to write $\mathscr{F} = \{f_j\}_{j\in\mathbb{N}}$. Let us observe, first of all, that we can put $\mathscr{F}$ into one-one correspondence with the set $\mathscr{B}$ of all the infinite sequences of binary numbers. To each $f_j \in \mathscr{F}$, there corresponds the sequence $b_{j,1}, b_{j,2}, b_{j,3}, \ldots$, where $b_{j,i} = f_j(i)$, for $i, j \in \mathbb{N}$. Therefore if $\mathscr{F}$ is denumerable, so too is the set $\mathscr{B}$. Since $\mathscr{B}$ is denumerable, we can enumerate its elements one after the other, listing, for each element (for every sequence), the binary digits which comprise it. We can arrange such an enumeration in an infinite square matrix:

$$\begin{array}{llll}
b_{1,1}, & b_{1,2}, & b_{1,3}, & \ldots \\
b_{2,1}, & b_{2,2}, & b_{2,3}, & \ldots \\
b_{3,1}, & b_{3,2}, & b_{3,3}, & \ldots \\
\vdots & & &
\end{array}$$

where row j contains the sequence for the jth function. Writing $\overline{b}$ for the complement of the binary number b, let us now consider the sequence of binary numbers $\overline{b}_{1,1}, \overline{b}_{2,2}, \overline{b}_{3,3}, \ldots$. This sequence (since it is an infinite sequence of binary numbers) is certainly an element of $\mathscr{B}$, however it does *not* appear in our matrix (and therefore does not appear in our enumeration) for the reason that each line is different at *at least one* point. Along the diagonal, the sequence found in the matrix has elements $b_{j,j}$, while, by construction, the new sequence has the element $\overline{b}_{j,j}$ in position j.

We have therefore a contradiction: we had assumed that $\mathscr{F} = \{f_j\}_{j\in\mathbb{N}}$ was an enumeration of *all* functions (that is, of all sequences), while we have constructed

Church's Thesis

The proofs of equivalence of the various programming languages (and between the various computability formalisms) are genuine theorems. Given the languages $\mathscr{L}$ and $\mathscr{L}'$, write first in $\mathscr{L}$ the interpreter for $\mathscr{L}'$ and then write in $\mathscr{L}'$ the interpreter for $\mathscr{L}$. At this point $\mathscr{L}$ and $\mathscr{L}'$ are known to be equivalent. A proof of this type has been effectively given for all existing languages, so they are therefore provably equivalent. This argument would, in reality, leave open the door to the possibility that sooner or later someone will be in a position to find an *intuitively computable* function which is not associated with a program in any existing programming language. All equivalence results proved over more than the last 70 years, however, amount to convincing evidence that this is impossible. In the mid-1930s, Alonzo Church proposed a principle (which since then has become known as Church's, or the Church-Turing, Thesis) that states exactly this impossibility. We can formulate Church's Thesis as: every intuitively computable function is computed by a Turing Machine.

In contrast to the equivalence results, Church's Thesis is not a theorem because it refers to a concept (that of intuitive computability) which is not amenable to formal reasoning. It is, rather, a philosophical principle which the computer science community assumes with considerable strength to be true so that it will not even be discredited by new computational paradigms, for example quantum computing.

a function (a sequence) which did not belong to the enumeration. Therefore the cardinality of $\mathscr{F}$ is strictly greater than that of $\mathbb{N}$.

It can be shown that $\mathscr{F}$ has the cardinality of the real numbers. This fact indicates that the set of programs (which is denumerable) is much smaller than that of all possible functions, and therefore of all possible problems.

3.5 Chapter Summary

The phenomenon of computation on which Computer Science is founded has its roots in the theory of computability which studies the formalisms in which one can express algorithms and their limits. The chapter has only presented the main result of this theory. This is a fact of the greatest importance, one that every computer scientist should know. The principal concepts which were introduced are:

- *Undecidability*: there exist many important properties of programs which cannot be determined in a mechanical fashion by an algorithm; amongst these is the halting problem.
- *Computability*: a function is computable when there exists a program which computes it. The undecidability of the halting problem assures us that there exist functions which are not computable.

- *Partiality*: the functions expressed by a program can be undefined on some arguments, corresponding to those data for which the program will fail to terminate.
- *Turing Completeness*: every general-purpose programming language computes the same set of functions as those computed by a Turing Machine.

3.6 Bibliographical Notes

The original undecidability result is in the paper by A.M. Turing [3] which ought to be necessary reading for every computer scientist with an interest in theory. More can be found on the arguments of this chapter in any good textbook on computability theory, among which, let us recommend [1], which we cited in Chap. 2, and the classic [2] which after more than 40 years continues to be one of the most authoritative references.

3.7 Exercises

1. Proof that the restricted halting problem is undecidable. That is, determine whether a program terminates when it is applied to itself. (Suggestion: if the problem were decidable, the program which decides it would have to have the same property as program K, which can be derived by contradiction in the usual fashion.)
2. Show that the problem of verifying whether a program computes a constant function is undecidable. Hint: given a generic program P, consider the program Q_P, with a single input, specified as follows:

$$Q_P(y) = \begin{cases} 1 & \text{if } P(P) \text{ terminates,} \\ \text{does not terminate} & \text{otherwise.} \end{cases}$$

   (i) Write the program Q_P;
   (ii) assume now that P is a program such that $P(P)$ terminates. What is the behaviour of Q_P, as y varies?
   (iii) what is, on other hand, the behaviour of Q_P, as y varies, if $P(P)$ does not terminate?
   (iv) from (ii) and (iii), it can be obtained that Q_P computes the constant function *one* if and only if $P(P)$ terminates;
   (v) if it were now decidable whether a program computes a constant function, the restricted halting problem would also be decidable, given that the transformation that, given P, constructs Q_P is completely general.

References

1. J. E. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, 2001.
2. H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw Hill, New York, 1967.
3. A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc.*, 42:230–365, 1936. *A Correction*, ibidem, 43 (1937), 544–546.