

Chapter 2

Problems and Effective Procedures

What does computable mean? What problems do we consider?

To a computer science student the question “*what does computable mean?*” might appear frivolous. We use computers every day, so don’t they—obviously—“compute things” for us, although one might say, very often they mainly retrieve and display information, for instance when we are browsing web pages or watching videos. And “computable” means “being able to be computed”, so what is the point?

The ACM¹ Computing Curricula [11, Sect.2.1] states “computing to mean any goal-oriented activity requiring, benefiting from, or creating computers. Thus, computing includes . . . processing, structuring, and managing various kinds of information; finding and gathering information relevant to any particular purpose, and so on. The list is virtually endless, and the possibilities are vast.” This is uncontroversial but quite generic and does not really define what “computable” means.

It should be clear that we need to pin down what computable means precisely and formally if we want to explore the limits of computation in a scientific manner. The same issue arises with the definition of problems. Everybody has their own understanding of what a problem is: from not being able to pay the rent to finding the shortest path in a graph. Also in that respect we will have to restrict the definition in order to be able to apply formal reasoning so that we can prove results. It is also important to understand the difference between a problem and a program. Computable problems will be the ones for which there are programs that “solve” them. All these concepts will be carefully defined below.

We begin with a very short historical perspective (Sect. 2.1) introducing the notion of “effective procedure”. Sets and structures on sets, i.e. relations and functions,

¹The Association for Computing Machinery (ACM), “the world’s largest educational and scientific computing society, delivers resources that advance computing as a science and a profession” [1]. It was founded in 1947 and has its headquarters in New York.

together with their basic operations, are defined in Sect. 2.2 and some basic reasoning principles recalled. Finally, we define precisely what we mean by a “problem” in Sect. 2.3.

2.1 On Computability

In order to define the term “computable” we need to have a look at *what* is to be computed and *how* “computed” is actually defined. *What* is to be computed is generically called a problem. As computing in the 21st century is ubiquitous and microprocessors are not only in computers, but also in games consoles, mobile phones, music players and all kinds of consumer products, even washing machines, we need to restrict the problem domain since the “problem” in the context of washing will be very different from the “problem” in the context of game playing, or other areas.²

But first we address the “computable” question as the kind of problems we will look at depends on this definition.

2.1.1 Historical Remarks

Kleene wrote that the origin of algorithms³ goes back at least to Euclid⁴ ca. 330 B.C. according to [10] which provides an excellent historic overview. There have been many machines designed for calculation, from Gottfried Leibniz⁵ to Charles Babbage⁶ who wanted to automate calculations done in analysis.

The *Entscheidungsproblem*, the *decision problem for first order logic*, was raised in the 1920s by David Hilbert⁷ and was described in [6]. The problem is to give a decision procedure “that allows one to decide the validity (respectively satisfiability) of a given logical expression by a finite number of operations” [6, pp. 72–73]. For Hilbert this was a fundamental problem of mathematical logic and played an important part

²And every reader may have their own problems, i.e. their own idea of what a “problem” is.

³The name “algorithm” dates back to the name of the ninth century Persian mathematician Al-Khwarizmi.

⁴Euclid (ca. 300 BC) was a Greek mathematician often called the “Father of Geometry” who also worked in number theory. He is the inventor of the common divisor algorithm.

⁵Gottfried Wilhelm von Leibniz (July 1, 1646–November 14, 1716) was a German mathematician and philosopher, credited with the independent invention of differential and integral calculus. He also invented calculating machines.

⁶Charles Babbage, (December 26, 1791–October 18, 1871) was an English mathematician, philosopher, inventor and mechanical engineer (and a Fellow of the Royal Society). He is also known for originating the concept of a programmable calculating machine. The London Science Museum has constructed two of his machines where they are on display.

⁷David Hilbert (January 23, 1862–February 14, 1943) was a world-renowned German mathematician.

in his program of finding a finite axiomatisation of mathematics that is consistent, complete and decidable (in an automatic way).

Gödel⁸ then proved in 1931 that no axiomatic system of arithmetic can exist that is consistent and complete. This result proves a significant inherent limitation of mathematical logic and deductive systems. He gave the definition of *general recursive functions* on natural numbers based on previous work by Herbrand, Skolem, Hilbert, and Péter.

At the age of only 22 and still a student, Alan Turing . . .

. . . worked on the problem for the remainder of 1935 and submitted his solution to the incredulous Newman on April 15, 1936. Turing's monumental paper 1936 was distinguished because: (1) Turing analyzed an idealized human computing agent (a computer) which brought together the intuitive conceptions of a function produced by a mechanical procedure which had been evolving for more than two millenia from Euclid to Leibniz to Babbage and Hilbert; (2) Turing specified a remarkably simple formal device (Turing machine) and proved the equivalence of (1) and (2); (3) Turing proved the unsolvability of Hilberts *Entscheidungsproblem* which established mathematicians had been studying intently for some time; (4) Turing proposed a universal Turing machine, . . . an idea which was later to have great impact on the development of high speed digital computers and considerable theoretical importance. [10, Sect. 3]

Turing used the term *a-machine* for his theoretical computing device but we now call them Turing machines in his honour.

Independently, Alonzo Church⁹ proposed Church's Thesis "which asserts that the effectively calculable functions should be identified with the recursive functions" [10].¹⁰ Church had initially intended this to be the definition of "effectively computable". Nowadays one uses the term "*Church-Turing thesis*" which amalgamates both theses, identifying all the intuitive notions of computation and all the various formal definitions. What is to be subsumed under the notion of "intuitively computable", is obviously up to interpretation. There have been suggestions that there are computational models much more powerful than Turing machines, called *hypercomputation*, but this is currently hotly debated. We will discuss this in more detail in Chap. 11 dedicated to the Church-Turing thesis.

Despite general recursive functions and Turing machines being the first formal definitions of computability, this book will not use the former and only briefly look at the latter. The reason is that the former needs some mathematical background and the latter is tedious to program. We follow the idea of Neil Jones [7] and use a high-level programming language which will be introduced in the next chapter. Thus, we need to justify that our language qualifies as "intuitive notion of computability". We must therefore understand what is required for such an intuitive notion of computation.

⁸Kurt Friedrich Gödel (April 28, 1906–January 14, 1978) was an Austrian (and American) logician, mathematician, and philosopher and is considered one of the most significant logicians in history. He proved many important results, relevant here is the *Incompleteness Theorem*.

⁹Alonzo Church (June 14, 1903–August 11, 1995) was an important American mathematician and logician.

¹⁰Church's first version that the computable functions are those definable by λ -terms [3] was initially rejected.

2.1.2 *Effective Procedures*

Effective Procedures, or effective algorithms are the programs that we understand to perform computations. The naming goes back to *Alan Turing*: “A function is said to be effectively calculable if its values can be found by some purely mechanical process. . . . We may take this statement literally, understanding by a purely mechanical process one which could be carried out by a machine [12, p. 166]. Note that Turing uses the word “calculable” here. In the 1930s computations usually referred to mathematical calculations. The machines he suggested have been called Turing machines and we will look at them more carefully in Chap. 11.

So what is an effective procedure? Copeland gives the following definition in [4]:

‘Effective’ and its synonym ‘mechanical’ . . . do not carry their everyday meaning. A method, or procedure, M , for achieving some desired result is called ‘effective’ or ‘mechanical’ just in case

1. M is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);
2. M will, if carried out without error, always produce the desired result in a finite number of steps;
3. M can (in practice or in principle) be carried out by a human being unaided by any machinery save paper and pencil;
4. M demands no insight or ingenuity on the part of the human being carrying it out.

The instructions of an effective procedure must therefore be executable in a mechanical way. This means that instructions (or commands) in programs must not be “vague”. For instance “find a number that has property P ” which cannot be carried out effectively. How do we find the number? We need instructions that produce a number effectively such that it has the desired property. Therefore we cannot use oracles or choice axioms in our effective procedures. Moreover, we must be able to carry out the procedures in a finite amount of time. Infinite computations are by definition not effective. However, all notions of computation allow the definition of infinite computations as well. It will become clear in Chap. 8 why it is difficult to separate finite from infinite computations.

The exact meaning of “intuitive” computable is to a certain degree subject to interpretation. Some researchers insist that the mechanical computability by Turing machines does not include the so-called interactive computation, where humans (or other potentially non-computable oracles) interact with the program (see [5]). This appears to be equivalent to Turing’s o -machines, Turing machines with an “oracle tape”, an extra tape on which the Turing machine can write a word w and then ask the environment, the oracle, to answer whether w is in a certain set A which can be arbitrarily complicated (in particular it does not have to be decidable by an a -machine). The resulting definition of computability by o -machines is called *relative computability*. Relative computability will *not* be covered in this introductory book. Also we will not discuss computability of infinite objects (e.g. real number computation).

Turing’s machine model extends the concept of a finite state automaton with extra memory. This memory is organised as a tape on which symbols can be written and read sequentially by a head that moves along the tape and that is controlled by the finite state automaton. Programming Turing machines is therefore a tedious and error-prone undertaking. For this reason, we don’t want to use them to prove anything in this book, but rather use a programming language close to what we use on a daily basis. In Chap. 3, a more convenient notion of “effective procedure” will thus be presented. Turing machines will, for the sake of completeness and historical importance, be presented in detail in Sect. 11.3.

The following definition will be useful to compare languages later (for instance in Chap. 11 and Sect. 10.2).

2.2 Sets, Relations and Functions

Before we continue and define problems and solutions more formally, we recall some basic definitions that allow us to make formal statements throughout this book. Readers well familiar with those concepts can skip this section. We will discuss sets and structures on sets, namely relations and functions. We introduce operations on sets, fix notation, and recall some basic reasoning principles, which will be used throughout the book. A proper introduction to sets and logic for computing can be found e.g. in [8].

2.2.1 Sets

Sets are collections of objects. The collections can be finite or infinite. We will usually only consider *homogeneous* sets which means that the objects in a set are all of the same type.¹¹ For each element of this type one must be able to say whether the element is in the given set or not. An example of a set of natural numbers is the set S_{10} containing the numbers from 1 to 10. In this case, number 3 is in the set S_{10} but number 42 is not. It is important to observe that one does not care how many times the objects appears in the set as one would do in a list or an array. A set thus abstracts away from the number of occurrences. An object simply is either in or out. If we have such knowledge for all objects of the given underlying type we have uniquely defined a set.

Let us now fix some notation:

Definition 2.1 (*Sets*) A finite set containing n different objects e_1, e_2, \dots, e_n is written

$$\{e_1, e_2, \dots, e_n\}$$

¹¹This type may be a set again.

We call those objects contained in a set, the *elements* of this set. The *empty set* is the unique set that contains no elements at all and is usually denoted $\{\}$ or \emptyset .

Let A be a set of elements of type T . The *elementhood operation* is a statement

$$x \in A$$

stating that element x is in set A (“belongs to A ”, “is contained in A ”). If the set is infinite, we cannot write down all the elements. In this case we usually write the “law” that states which elements are in the set as follows (which can also be used to describe finite sets). If S is a type and $P(x)$ denotes a condition on variable x then

$$\{x \in S \mid P(x)\}$$

describes the set of all elements of type S that have property P . The type of all natural numbers is denoted \mathbb{N} (which contains 0), the integer numbers is denoted \mathbb{Z} and the real numbers is denoted \mathbb{R} . The type of Boolean values $\{\text{true}, \text{false}\}$ is denoted \mathbb{B} .

Example 2.1 Here are some examples of finite sets with objects (elements) in \mathbb{N} :

1. $\{1, 10, 100\}$: the set of natural numbers containing the three elements 1, 10 and 100.
2. \emptyset : the empty set containing no natural number.
3. $\{x \in \mathbb{N} \mid x \text{ is even}\}$: the infinite set of all even natural numbers, which is the set $\{0, 2, 4, 6, 8, 10, 12, \dots\}$. The notation with \dots followed by a closing $\}$ is sometimes used to indicate an infinite set when the condition P used to define it is clear from the context. Note that in this example the condition $P(x)$ is “ x is even”.
4. $\{x \in \mathbb{N} \mid x = 10^n, 0 \leq n \leq 2\}$: the finite set containing the first three powers of 10, namely $1 = 10^0$, $10 = 10^1$ and $100 = 10^2$. So in fact this set is equal to the first. More about equality of sets follows these examples.

Definition 2.2 (*Set equality and subsets*) Let S_1 and S_2 be two sets ranging over the same type T of objects. We say that two sets S_1 and S_2 are *equal*, short $S_1 = S_2$, if, and only if, they contain exactly the same elements. This confirms that it is enough to know which elements are in the set and which are not to uniquely define a set.

We say that a set S_1 is a *subset* of a set S_2 (or S_1 is contained in S_2) if, and only if, every element of S_1 is also an element of S_2 . More formally we can also write

$$\begin{aligned} S_1 \subseteq S_2 &\iff \forall x \in T. x \in S_1 \Rightarrow x \in S_2 \\ S_1 = S_2 &\iff \forall x \in T. x \in S_1 \Leftrightarrow x \in S_2 \end{aligned}$$

where \Rightarrow denotes implication and \Leftrightarrow denotes equivalence and $\forall x \in T. P$ denotes universal quantification over all elements of type T .

In the above definition we used the phrase “if, and only if” (in the formal version \Leftrightarrow) and not just “if” (formally \Leftarrow) for a good reason. For a definition, it is important to cover all cases exactly. Consider the following statement: “Sets A and B (over natural numbers) are equal if S and T are both the empty set.” This is obviously a correct statement about equality of sets A and B . But it is far from a definition of equality. The statement does not specify anything about the equality of non-empty sets. Clearly, its contraposition “if A and B are equal sets then A and B are both empty” is wrong. Thus the statement “sets A and B (over natural numbers) are equal if, and only if, S and T are both the empty set.” is equally wrong.

As explained above the use of phrase “if, and only if” is important and we will encounter it often throughout the book. Therefore, we sometimes abbreviate it and simply write “iff” instead of “if, and only if”.

In order to show equality of two sets, an important reasoning principle is often used:

Proposition 2.1 *Let S_1 and S_2 be sets of objects in T , then $S_1 = S_2$ if, and only if, $S_1 \subseteq S_2$ and $S_2 \subseteq S_1$. In other words, S_1 equals S_2 if, and only if, S_1 is a subset of S_2 and vice versa.*

Proof We need to show the two directions of the “if, and only if”. The “only if” (\Rightarrow) and the “if” (\Leftarrow) direction.

“ \Rightarrow ”: If $S_1 = S_2$ then by definition $S_1 \subseteq S_2$, as being equal is a special (degenerated) case of being a subset of. Analogously, $S_2 \subseteq S_1$.

“ \Leftarrow ”: Assume $S_1 \subseteq S_2$ and $S_2 \subseteq S_1$. To show that both sets are equal we must show that they contain exactly the same elements, i.e. for all $x \in T$ it must hold that $x \in S_1$ iff $x \in S_2$. Unfolding the meaning of “iff” we get two conditions for all $x \in T$, namely $x \in S_1 \Rightarrow x \in S_2$ and $x \in S_2 \Rightarrow x \in S_1$. We can move the quantifier $\forall x$ around both conditions separately without changing the meaning of the formula, so it suffices to show:

$$\begin{aligned} \forall x \in T. x \in S_1 \Rightarrow x \in S_2 \quad \text{and} \\ \forall x \in T. x \in S_2 \Rightarrow x \in S_1 \end{aligned}$$

and thus by Definition 2.2 that $S_1 \subseteq S_2$ and $S_2 \subseteq S_1$ which were our assumptions.

Definition 2.3 (*Set operations*) We will use the following standard operations on sets: union ($S_1 \cup S_2$) intersection ($S_1 \cap S_2$) and set difference ($S_1 \setminus S_2$). They are defined as follows:

$$\begin{aligned} x \in S_1 \cup S_2 &\Longleftrightarrow x \in S_1 \vee x \in S_2 \\ x \in S_1 \cap S_2 &\Longleftrightarrow x \in S_1 \wedge x \in S_2 \\ x \in S_1 \setminus S_2 &\Longleftrightarrow x \in S_1 \wedge \neg(x \in S_2) \end{aligned}$$

where \vee denotes logical disjunction (“or”), \wedge denotes logical conjunction (“and”), and \neg denotes logical negation (“not”). If x is not contained in A , we usually abbreviate $\neg(x \in A)$ by simply writing

$$x \notin A.$$

If S is a set of elements of type T , we call $T \setminus S$ the *complement* of S , which is sometimes also abbreviated \bar{S} .

Example 2.2 Here are some concrete examples of set operations and their results:

$$\begin{aligned}\{3, 5, 7\} \cup \{2, 4, 6, 8\} &= \{2, 3, 4, 5, 6, 7, 8\} \\ \{3, 5, 7\} \cap \{2, 4, 6, 8\} &= \{\} \\ \{3, 5, 7\} \setminus \{3, 5, 8, 16\} &= \{7\} \\ \mathbb{N} \setminus \{x \in \mathbb{N} \mid x \text{ is even}\} &= \{x \in \mathbb{N} \mid x \text{ is odd}\}\end{aligned}$$

Definition 2.4 (*Cartesian product*) Let S_1 and S_2 be sets. Then $S_1 \times S_2$, the *Cartesian product*¹² of S_1 and S_2 , is the set of pairs (i.e. tuples) (s, t) where $s \in S_1$ and $t \in S_2$. In other words:

$$S_1 \times S_2 = \{(s, t) \mid s \in S_1 \wedge t \in S_2\}$$

Based on the cartesian product one can also form sets of tuples of length k over a given set or type:

Definition 2.5 (*k-tuples*) Let S be a set. Then S^k , the *k-tuples* over S , is defined as follows:

$$S^k = \underbrace{S \times S \times S \dots \times S}_{k \text{ times}}$$

such that elements of S^k are tuples of length k , i.e. $(s_1, s_2, \dots, s_{k-1}, s_k)$ where $s_i \in S$ for all $1 \leq i \leq k$.

If we consider sets from a programming perspective as a kind of datatype then we would like to nest the set data type constructor. In other words, we would like to have set of sets, and so on.

Definition 2.6 (*Powerset*) Let S be a set. Then $Set(S)$, the *powerset* of S , denotes the set of all subsets of S , including the empty set, and the set S .

$$Set(S) = \{\text{set } s \mid s \subseteq S\}$$

Example 2.3

$$\begin{aligned}Set(\{1, 10, 100\}) &= \{\emptyset, \{1\}, \{10\}, \{100\}, \{1, 10\}, \{1, 100\}, \{10, 100\}, \{1, 10, 100\}\} \\ Set(\mathbb{N}) &= \{\text{set } s \mid \forall x \in s. x \in \mathbb{N}\}\end{aligned}$$

¹²The Cartesian product is named in honour of René Descartes (31 March 1596–11 February 1650), a French mathematician and philosopher, who spent most of his life in the Netherlands, and is famous for his saying “I think therefore I am” as well as for the development of (Cartesian) analytical geometry. He was invited to the court of Queen Christina of Sweden in 1649. “In Sweden—where, Descartes said, in winter men’s thoughts freeze like the water—the 22-year-old Christina perversely made the 53-year-old Descartes rise before 5:00 am to give her philosophy lessons, even though she knew of his habit of lying in bed until 11 o’clock in the morning” [9]. Consequently, Descartes caught pneumonia and died.

Another standard set used in this book is the set of words over a finite alphabet Σ as used for instance by *finite state automata*. These words are just finite strings of letters of the alphabet, including the empty string.

Definition 2.7 (*Set of words*) Let Σ be a finite alphabet of symbols (or letters). Then we define a new set Σ^* by providing the rules to generate elements of this set and state that these are the only rules to generate elements of the set. The rules are as follows:

$$\begin{aligned} \varepsilon &\in \Sigma^* \\ aw &\in \Sigma^* \quad \text{if } a \in \Sigma \wedge w \in \Sigma^* \end{aligned}$$

This is an *inductive* definition. The first rule states that the empty word ε is a word which provides the termination case of the induction. The second rule describes how to generate new elements from already generated ones.

Example 2.4 For alphabet $\Sigma = \{0, 1\}$ here are some examples of words in Σ^* :

- ε (the empty word)
- 0
- 01
- 1111001

Finally, for the definition of partial functions in Sect. 2.2.4 we need a way to extend a set by a unique new symbol.

Definition 2.8 (*One-point-extension*) Let S be a set over type T . Then we define a new set S_\perp in $\text{Set}(T)$ by adding to S a new element \perp (called¹³ “undefined”) that is assumed to be different from all elements in S .

$$S_\perp = \{x \in T \cup \{\perp\} \mid x \in S \vee x = \perp\}$$

2.2.2 Relations

Relations are special sets. We have already seen some relations in the previous section, the equality and subset relation are both binary relations on sets. A binary relation R is simply a set of pairs and we say that two elements s and t are in this relation R if the pair (s, t) is in the set R .

Definition 2.9 (*Relations*) Relations are sets. We define:

- a *unary relation* R over elements of type T is a subset $R \subseteq T$. An object $t \in T$ is said to be *in relation* R iff $t \in R$.
- a *binary relation* R over elements of type $S \times T$ is a subset $R \subseteq S \times T$. A pair (s, t) is said to be *in relation* R iff $(s, t) \in R$.

¹³The symbol itself is called a “perp”.

- a *ternary relation* R over elements of type $S \times T \times U$ is a subset $R \subseteq S \times T \times U$. A triple (s, t, u) is said to be *in relation* R iff $(s, t, u) \in R$.¹⁴

Example 2.5 The quality relation and subset relation over sets of elements of type T as defined in Definition 2.2 are actually binary relations in the following sense:

$$\begin{aligned} _ = _ &\subseteq \text{Set}(T) \times \text{Set}(T) \\ _ \subseteq _ &\subseteq \text{Set}(T) \times \text{Set}(T) \end{aligned}$$

We usually write $S_1 \subseteq S_2$ (so-called “infix notation”) instead of $(S_1, S_2) \in _ \subseteq _$.

2.2.3 Functions

We intuitively understand what the addition or multiplication functions are, and maybe also the factorial function. Functions describe *maps* from objects of a certain type into objects of another¹⁵ type. In functional programming languages, functions are first-class citizens. The programmer can define those functions syntactically. For us, however, functions are descriptive¹⁶ and *not* programs or part thereof. We can describe functions as special relations which we will do next.

2.2.4 Partial Functions

Definition 2.10 (*Partial Functions*) Let A and B be sets of possibly different types of elements, e.g. $A \in \text{Set}(S)$ and $B \in \text{Set}(T)$. A *partial function* f from A to B is a subset of $A \times B$ (i.e. $f \subseteq A \times B$) satisfying the following uniqueness condition:

For all $a \in A$ there is *at most one* $b \in B$ such that $(a, b) \in f$.

To abbreviate that f is a partial function from A to B we briefly write $f : A \rightarrow B_\perp$, where we call A the argument type of f and B the result type. The reason for actually writing B_\perp (defined in Definition 2.8) in this notation will become clear shortly when we define the following binary *application relation* $_@ _ \subseteq (A \rightarrow B_\perp) \times A$ for a partial function $f : A \rightarrow B_\perp$ and an element $a \in A$:

$$f@a = \begin{cases} b & \text{if } (a, b) \in f \\ \perp & \text{otherwise} \end{cases}$$

¹⁴At first glance, it is not obvious whether $S \times T \times U$ means $S \times (T \times U)$ or $(S \times T) \times U$. We assume cartesian products to be *associative*, identifying these two definitions, thus dropping the extra parentheses and writing (s, t, u) for triples, and similarly for n -tuples where $n > 3$.

¹⁵Which may possibly be the same type.

¹⁶Mathematical objects.

Instead of $f @ a$ we will be writing $f(a)$ which is the common notation for function application also widespread in programming languages. If $(a, b) \in f$ we therefore simply write $f(a) = b$ and say that “ f applied to a equals b .” We also call a the *argument* of the function (application) and b the *result* of the application. By the uniqueness condition we know that there can only be one result which always lies in B_{\perp} . It is possible that no such $b \in B$ exists, in which case $f(a) = \perp$ and we say that f is *undefined* for a and often use the short notation $f(a) \uparrow$ to express this. We also sometimes use $f(a) \downarrow$ to express that $f(a)$ is *defined* when we are not interested in the concrete result value.

Functions with more than one argument are simply described by using a cartesian product as argument type. In this case the function takes a tuple as input.

Example 2.6 Consider the integer division operator on natural numbers, *div*. This *partial* function takes two integers n and m and returns $\frac{n}{m}$ in case $m \neq 0$. Thus $\text{div} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}_{\perp}$.

$$((n, m), r) \in \text{div} \text{ iff } \exists k < m. m \times r + k = n$$

where $m, n, r, k \in \mathbb{N}$. Note that we have that $\text{div}(n, 0) \uparrow$ as there is no natural number k that is smaller than 0.

2.2.5 Total Functions

Total functions are total in the sense that function application always returns a defined value. Therefore, total functions are just a very special case of partial functions.

Definition 2.11 (*Total Functions*) Let A and B be sets of possibly different types of elements, e.g. $A \in \text{Set}(S)$ and $B \in \text{Set}(T)$. A *total function* f from A to B is a subset of $A \times B$ (i.e. $f \subseteq A \times B$) satisfying the following two conditions (where the first is the uniqueness condition for partial functions):

1. For all $a \in A$ there is *at most one* $b \in B$ such that $(a, b) \in f$.
2. For all $a \in A$ there is *at least one* $b \in B$ such that $(a, b) \in f$.

To abbreviate that f is a total function from A to B we briefly write $f : A \rightarrow B$ where we again call A the argument type of f and B the result type. We can take the binary *application relation* defined in Definition 2.10 for partial functions and restrict its type to $_@_ \subseteq (A \rightarrow B) \times A$ for a total function $f : A \rightarrow B$ and argument $a \in A$. Since for total functions we know from the second condition that there must always be a $b \in B$ for every $a \in A$ (which is unique by the first condition) so we can never have $f @ a = \perp$. As for partial function application, we write $f(a)$ for $f @ a$ and if $(a, b) \in f$ we simply write $f(a) = b$ and say that “ f applied to a equals b .”

Example 2.7 Consider the factorial function on natural numbers, fac . Often the notation $n!$ is used instead of application $fac(n)$. This *total* function $fac : \mathbb{N} \rightarrow \mathbb{N}$ takes an integer n and returns the factorial of n defined as follows:

$$(n, r) \in fac \text{ iff } (n = 0 \wedge r = 1) \vee (n > 0 \wedge (n - 1, s) \in fac \wedge r = n \times s)$$

where $n, r, s \in \mathbb{N}$. This is a recursive (actually inductive) definition of fac as we use the function (functional relation) fac on the left and right hand side of the definition. The definition is, however, well defined as the argument for the application of fac on the right hand side uses a “smaller” argument $n - 1$ than the one on the left hand side (which uses n). When defining functions in this book we will normally not define the relation that defines the function but write the (equivalent) definition of function application, i.e. we define the result of $f(n)$ rather than defining the relation $(n, r) \in f$. For the factorial function we would typically write:

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fac(n - 1) & \text{otherwise} \end{cases}$$

2.3 Problems

Our first quest is to find a problem that is not computable (or decidable by a computer program) to learn and understand that not everything is computable even with unlimited resources. In order to do this, we obviously need to define what we mean exactly by “*problem*” and what we mean exactly by “*computable*.” Whereas the former is easy to do the latter is a bit more tricky.

We will allow ourselves to restrict the definition of *problem*. Since we are interested in the *Limits of Computation*, we are interested in *negative* results, i.e. what *cannot* be achieved in computing. If there is a problem of a restricted kind that is not computable then we still have found a problem that is not computable, so this restriction does not take anything away from our ambition.

A *problem* of the kind we are interested in is characterised by two features:

1. It is a uniform class of questions. *Uniform* refers to the domain of the problem, i.e. what data the problem is about. The type of domain must be precisely definable.
2. It can be given a definite and finite answer. The type of the answer must be also precisely definable.

The type in question can be any set, like for instance \mathbb{N} , \mathbb{N}_+ , Σ^* and so on.

Definition 2.12 Let S and T some well defined (finite) types. A *function problem* is a uniform set of questions, the answers of which have a finite type. The solution of a function problem is given as a partial function $f : S \rightarrow_{\perp} T$ as described in Sect. 2.2.4. The uniform question of this problem is of the sort: “given an $x \in S$, what is a $y \in T$ such that a certain condition on x and y holds?”

A *decision problem* is a relation $R \subseteq S$. The uniform question of this problem is of the sort: “given an $x \in S$, does x belong to R , i.e. $x \in R$? The solution of a decision problem is given as a *total* function $\chi : S \rightarrow \mathbb{B}$, also called the *characteristic function* of R .

Example 2.8 Here are some examples of function and decision problems:

1. For a tree t , what is its height? Domain: trees (apparently with arbitrary number of children). Answer for any given tree t : a natural number describing the height of t (and we know what the meaning of “height of a tree” is). The answer type is the type of natural numbers.
2. For a list of integers l , what does l look like when sorted? In other words, what is the sorted permutation of l using the usual ordering on integers? Domain and answer type are here the type of integer lists.
3. For a natural number n , is it even? Domain: natural numbers. Answer for any given number n : a Boolean,¹⁷ stating whether n is even or not (we understand what even and odd mean). The answer type is the type of boolean values.
4. For a given formula in number theory (arithmetic) ϕ , is it valid? Domain: formulae in arithmetic. Answer for a given formula ϕ : a Boolean, stating whether the formula ϕ is true (and we understand what it means for a formula to be true).

The first two examples above are function problems, the last two examples are decision problems.

Example 2.9 Here are some examples of problems that *do not qualify* as problems for us.

1. “What is the meaning of life?”¹⁸ This is not a uniform family of questions. Moreover, we do not know what the answer type is. If we’d expect a string as answer then it would still not qualify as we don’t know whether there *is* a definite answer.
2. “Is the number 5 even?” This is not a *uniform* class of questions, as this question only refers to the number 5.

¹⁷Boolean values are named after George Boole (2 November 1815–8 December 1864), an English mathematician and logician famous for his work on differential equations and algebraic logic. He is most famous for what is called Boolean algebra. Throughout this book, we will use the term “boolean” to indicate a truth value for which the corresponding algebra operations are available.

¹⁸This question is easily confused with the one famously asked in Douglas Adams’s masterpiece: “The Hitchhiker’s Guide to the Galaxy” [2] which actually is called: “Ultimate Question of Life, The Universe, and Everything”. The computer in question, *Deep Thought*, after a considerable 7.5 million years answered famously: “42”. Alas, nobody understood the question. So *Deep Thought* suggested to build an even more powerful super-computer to produce the question to the answer. This computer was later revealed to be planet “Earth” which was unfortunately destroyed 5 min before completion of the calculations.

2.3.1 Computing Solutions to Problems

According to the two types of problems introduced, we will consider two concrete kinds of “solving a problem”: computing a function and deciding membership in a set.

The data type of Turing machines is the set of finite words over a finite alphabet. Recall that Σ^* denotes all finite words over the alphabet Σ , including the empty word. The comparison test for tape symbols is built into the construction set and from that it is possible to implement equality of words. In a general notion of effective procedure, the data type should be general enough to encode finite words and their equality test. The latter must be effective so it must be terminating. This means that equality of infinite objects is likely to be problematic and thus we do not cover computability over infinite objects in this book.

Definition 2.13 Provided a certain choice of effective procedures \mathcal{P} , a (function or decision) problem is called \mathcal{P} -*computable* if, and only if, its solution can be computed (calculated) by carrying out a specific such effective procedure in \mathcal{P} . A decision problem that is computable is also called \mathcal{P} -*decidable*.

If the kind of effective procedures is known by the context we also simply use the unqualified terms *computable* and *decidable*.

It is important to remember that programs are solutions to *computable problems*. The programs that solve computable decision problems are also called decision procedures.

Example 2.10 The solutions to the computable and decidable, resp., problems in Example 2.8 are given below as *programs*.

1. For a tree t what is its height? The solution is a function program that takes a tree as input and computes its height.
2. For a list of integers l what is l sorted? The solution is a program that takes a list of integers as input and returns a sorted copy of the list. The program can use various well known sorting algorithms, e.g. bubble-sort, merge-sort, or quicksort. They all perform the same task eventually, but use different methods to achieve this and also may take different time. This is an issue we will discuss in the complexity part.
3. For a natural number n , is it even? The solution in a program takes a natural number as the input and returns the boolean value true if the input is even and false if it is odd. We call such a program also a decision procedure for the property of “being even”.
4. For a given formula in number theory (arithmetic) ϕ , is it valid? As discussed in the introduction, this is undecidable, so there can’t be any program that takes as input as an arithmetic formula (suitably encoded) and returns true if the formula is satisfiable and false if it is not.

What Next?

Now that we know what we mean by “computable” and have seen that the historically first definition of computability via a machine involves tedious low level programming, we want to define a high-level language that can do the job as well. So in the next chapter we introduce the language `WHILE` and in the following chapter we show that `WHILE`-programs can be legitimately chosen for effective procedures.

Exercises

1. What is the “Entscheidungsproblem”? What is the type of its domain? Is it a decision problem?
2. Why did Alan Turing allow his (pencil and paper) computing device to use only finitely many symbols (on the tapes) and let the “state of mind” of the computer only glance at finitely many symbols at any given time?
3. Which of the following pairs of sets A and B are equal? Show either $A = B$ or $A \neq B$.
 - a. $A = \mathbb{N} \times \mathbb{N}$ and $B = \mathbb{N}^2$
 - b. $A = \{1, 3, 5\}$ and $B = \{1, 3, 5, 6\}$
 - c. $A = \{1, 3, 3, 3\}$ and $B = \{1, 3\}$
 - d. $A = \{x \in \mathbb{N} \mid x = x + 1\}$ and $B = \emptyset$
 - e. $A = \{x \in \mathbb{N} \mid \text{even}(x) \wedge x < 11\}$ and $B = \{0, 2, 4, 6, 8, 10\}$
4. Describe the relation that one natural number can be divided by the second natural number without remainder as $R_{\text{divisible}} \subseteq \mathbb{N} \times \mathbb{N}$.
5. Give an example of a partial function of type $\mathbb{N} \rightarrow \mathbb{N}_{\perp}$ and an example of a total function $\mathbb{N} \rightarrow \mathbb{N}$, respectively.
6. What is the difference between a decision problem and a function problem?
7. Give an example of a problem that is neither a decision nor a function problem. Why is it acceptable that we consider only those specific kinds of problems?
8. Give two other examples of decision and function problems, respectively, that have not been mentioned in this chapter.
9. Assume that we have fixed the notion of effective procedures \mathcal{P} . When do we call a function problem \mathcal{P} -computable?
10. Assume that we have fixed the notion of effective procedures \mathcal{P} . When do we call a decision problem \mathcal{P} -decidable?

References

1. ACM Home Page, available via DIALOG, <http://www.acm.org>. Cited on 30 August 2015
2. Adams, D.: The Hitchhiker’s Guide to the Galaxy. Pan Books (1979)
3. Church, A.: An unsolvable problem of elementary number theory. *Am. J. Math.* **58**(2), 345–363 (1936)
4. Copeland, J.: The Church-Turing Thesis. References on Alan Turing (2000). Available via DIALOG. http://www.alanturing.net/turing_archive/pages/Reference%20articles/The%20Turing-Church%20Thesis.html. Cited 2 June 2015

5. Goldin, D., Wegner, P.: The church-turing thesis: breaking the myth. In: Cooper, S.B., Löwe, B., Torenvliet, L. (eds.) *New Computational Paradigms. Lecture Notes in Computer Science*, vol. 3526, pp. 152–168. Springer, Heidelberg (2005)
6. Hilbert, D., Ackermann, W.: *Grundzüge der theoretischen Logik*. Springer, Berlin (1928). (Principles of Mathematical Logic.)
7. Jones, N.D.: *Computability and Complexity: From a Programming Perspective*. MIT Press, Cambridge (1997). (Also available online at <http://www.diku.dk/neil/Comp2book.html>.)
8. Makinson, D.: *Sets, Logic and Maths for Computing*, 2nd edn. Springer, UTiCS Series (2012)
9. René Descartes. Entry in *Encyclopædia Britannica*, <http://www.britannica.com/biography/Rene-Descartes/Final-years-and-heritage>. Available via DIALOG. Cited on 2 Sept 2015
10. Soare, R.I.: The history and concept of computability. In: Griffor, E.R. (ed.) *Handbook of Computability Theory*, pp. 3–36. North-Holland (1999)
11. The Joint Task Force for Computing Curricula 2005 (ACM, AIS, IEEE-CS): *Computing Curricula 2005*. Available via DIALOG, http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf (2005)
12. Turing, A.: Systems of logic based on ordinals. *Proc. London Math. Soc.* **45**(1), 161–228 (1939)

Limits of Computation

From a Programming Perspective

Reus, B.

2016, XVIII, 348 p. 80 illus., Softcover

ISBN: 978-3-319-27887-2