3. A Presentation of Logical Tools

Mathematical logic has spread out in a variety of ways — model theory, proof theory, set theory, computability — according to Barwise's classification [Bar77]. To this taxonomy we can add type theory, which has become more important since the time of Barwise's overview. From our point of view, the importance of logic can be summarized as follows:

- it provides a natural framework for precisely constructing and expressing various concepts in computing;
- it lends itself well to formalization.

The first of these points has been described in Chapter 2. The properties of a program are quite naturally expressed in logic. The language of sets also finds many applications in this domain. Variables manipulated by programs range over a state space that is nothing more than a set defined by composing particular basic sets (specifically, integers, characters, etc.) by means of set operations (for example, the `record` construct of the Pascal language or the `struct` construct of C are both a form of Cartesian product). In other respects, computability theory makes us aware of the existence of unrealizable specifications.[1] Finally, type correctness makes programming more accurate and more secure.

Returning to the second point, above, our interest in formalization is twofold. On the one hand, the rigor of our specification texts and our reasoning about them is increased, since this is based on the manipulation of symbols that may be easily verified; on the other hand, the effort may be automated, or at least aided, by computer. It must be noted that the complete formalization of proofs, whether in software development or in a mathematical context, has a tendency to submerge the principal ideas under a plethora of more or less trivial lemmas. For such an approach to be viable, at least a partial automation proves to be indispensable in practice.[2] Proof theory provides essential tools in this respect.

On a practical level, mathematical logic aids in developing specification languages. An intuitive understanding of concepts, such as we acquire in school and in college, is often sufficient. Certain specification languages such as Z or B transform the language of sets and logic to accommodate the organizational needs of computing by means of adequate structuring mechanisms.

[1]Not because they are contradictory, but more subtly because no program can be derived to compute the desired function.

[2]An alternative point of view is presented in § 9.6.

Knowledge of certain more advanced aspects of logic is often very useful. This will be illustrated in § 3.1. Section 3.2 will give an overview of the historical context of mathematical logical. We will describe the different branches in § 3.3. Basic mathematical terms will be recalled in § 3.4. We will end with more technical discussions on well-founded relations and ordinals from § 3.5 — these concepts play a key role in issues of termination and computability in § 3.7. The last two sections may be omitted on a first reading.

3.1 Some Applications of Logic

3.1.1 Programming

Let's take a piece of paper on which are drawn some ordinary figures, and try to determine if a given point is inside this figure, or if a given line cuts that figure. In three dimensions, this presents a very concrete problem of aerial control. The reader is invited to spend a few minutes considering a solution in the programming language of his or her choice.

Do we, for example, construct some form of structured variables for each basic form? Do we try to combine everything into a tree structure? We must consider every possible interaction.

It's much more simple: we use the *characteristic function* of the figure under consideration, that is a function that for every point returns the value *true* if the point belongs to the figure, and the value *false* otherwise. The reader should be able to easily express the characteristic function of basic figures (discs, rectangles, etc.) in the programming language of choice. But this representation doesn't really catch our interest unless we can construct new figures from known figures. For example, the intersection of two figures represented by f and g is a function which, when applied to the point p, returns *true* if and only if $f(p) = \mathit{true}$ and $g(p) = \mathit{true}$. The function that computes the intersection is very general, and makes a total abstraction from the particulars of the figures themselves. Other forms of composition (complement, union) are also easy to obtain, as are transformations such as translations, symmetries or rotations.

Everything rests on one essential ingredient: the ability to pass functions as parameters and return functions as a result. What programming language should we choose? At first sight we find the concept of a pointer to a function, widely used in the C programming language, to be convenient. In reality, this is only sufficient to cover the case when the functions used are finite in number and are known in advance. The problem with not perceiving these limitations is that we may hope to be able to resolve the problem by taking a sufficiently shrewd approach. In reality, only the functional languages, based on the λ-calculus (see later) such as Scheme, ML or Haskell, provide a sufficiently general mechanism.

The underlying problem is to know if functions are considered as objects that can be manipulated in the same way as data structures. This is not a trivial question. We will see that in set-based specification techniques, we regularly manipulate binary relations, functions being a particular case of relations. These relations are intended to be implemented with data structures (tables, pointers, etc.) or algorithms (procedures, functions). Choosing the right solution is delicate. If the development is undertaken unadvisedly, or rashly, it may well end up with an inefficient or overly complex implementation — or just fail.

3.1.2 Sums and Unions

Let us examine some other constructs used in formal languages. The reader probably knows already how to use symbols such as $\cup$ and can associate it with a simple intuitive interpretation — combining the elements of two sets. This notation is generally used to combine sets of the same "kind". For example we can state:

$$\begin{aligned}&\{x \in \mathbb{R} \mid 1 \le x \le \pi\} \cup \{x \in \mathbb{R} \mid 2 \le x \le 2\pi\}\\ &= \{x \in \mathbb{R} \mid 1 \le x \le 2\pi\} \ .\end{aligned}$$

We don't feel the need to combine dissimilar sets, for example a set of integers, a set of couples and a set of sets:

$$\{1,2,3\} \ , \quad \{\langle 1,2\rangle, \langle 3,4\rangle\} \quad \text{and} \quad \{\{1,3,4\},\{1,5\}\}$$

which would yield:

$$\{1,\ 2,\ 3,\ \langle 1,2\rangle,\ \langle 3,4\rangle,\ \{1,3,4\},\ \{1,5\}\} \ ;$$

but after all, nothing is impossible. We actually often need to mix heterogeneous data in computing. For example, in protocols, when we want to manipulate messages having different formats in a uniform way. Or in parsers, when we construct a syntax tree: a node corresponding to a statement can have two children if it represents the sequential composition of two statements, three children if it represents an if-then-else statement, etc.; moreover we see that nodes can represent statements or expressions. A data structure representing elementary geometric figures, say circles or triangles, would have, respectively, two fields (the center, which is a point, and the radius, which is a distance) or three fields (the vertices, which are points). A more elaborate example is the set of finite integer sequences, which can be seen as an infinite union:[3]

$$\{\varnothing\} \cup \mathbb{N} \cup (\mathbb{N} \times \mathbb{N}) \cup (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \cdots$$

However, mixing heterogeneous objects is not harmless. It is plainly meaningful to reject, at compile time, a test like $a = b$ if a and b have different types. The usual interpretation is that a and b take their values from two different

[3]We need a singleton for representing the empty sequence. The usual set-theoretical trick is to take $\{\varnothing\}$.

sets A and B, say floats and strings. But we could just as easily agree that a and b take their values from the same set: $A \cup B$! And let us stress that we cannot just disallow $A \cup B$, as this notion is needed in the previous examples.

How can we get the flexibility that we need while simultaneously controlling the coherency of data and operations? The concept of sum introduced in 2.3.4 is just the ticket. In a good type system, A, B and $A + B$ can be distinguished.

A sum is dealt with using an operator able to check whether a given element s comes from an element a of A or from an element b of B, and then to direct the computation appropriately; the computation depends on a in the first case and on b in the second case. Such constructs are available in modern languages like ML. In Pascal (or C) it is possible to emulate a sum using a `record` construct with variants and a switch field, but it is the responsibility of the programmer to ensure that a variant is always used in a way consistent with the switch field. Note that during the initial design of ASN1, a standardized language for describing the format of data exchanged in protocols, sums were not recognized as a primitive concept, leading to many complications.

In ASN1, the expression **CHOICE** $\{ \ a \ A, \ b \ B \ \}$ yields a value whose type is either A or B. Switch fields (like a or b) are mandatory only since 1994. Before this date, they were confused with labels, which are integers encoding the type of the fields of a compound value. They are clumsy and cannot solve the ambiguity which appears if A and B happen to represent the same type.

3.1.3 Chasing Paradoxes Away

Let us again consider the example of sequences. They can be characterized by the following property: "to be empty or an integer or a pair of integers or etc.".

We often need to form sets from elements satisfying a given property — such a set is defined by *comprehension*. In this way we enter into the realm of the first version of set theory, where every collection made of objects characterized by a given property is a set. This so-called "naïve set theory" turned out to be inconsistent! Technically, an inconsistent system is a system where one thing and its contrary can be proved (formally: $P \wedge \neg P$) or, equivalently, everything can be proved.

Let us consider one of the simplest paradoxes, called Russell's paradox. In general, a set is not a member of itself. For instance, we have $\neg(\mathbb{B} \in \mathbb{B})$ because $\mathbb{B}$ is *not* a Boolean. Could we imagine a set which is a member of itself? Yes, though we have to think a bit.[4] Anyway, what matters is not whether such sets exist or not, but that we consider the property $x \in x$ and its negation.

Let us define by comprehension $R \stackrel{\text{def}}{=} \{x \mid \neg(x \in x)\}$. If $R \in R$, R must satisfy the characteristic property of members of R, that is, $\neg(R \in R)$. If $\neg(R \in R)$, R possesses the characteristic property, hence $R \in R$. If we define $P \stackrel{\text{def}}{=} R \in R$, we have P and $\neg P$ at the same time, which is inconsistent.

[4]Consider, for instance, the set of sets which can be defined with less than a hundred English words.

Formally we have just shown that $P \Rightarrow \neg P$ and $\neg P \Rightarrow P$. By the equivalence (3.6) on page 47, the first implication yields $P \Rightarrow (P \Rightarrow \mathbf{f})$, which by (3.11) boils down to $(P \wedge P) \Rightarrow \mathbf{f}$, then to $P \Rightarrow \mathbf{f}$ which we use twice. First, it can be written $\neg P$, and we deduce P from the second implication. Second, combined with P we get $\mathbf{f}$.

The same paradox arises if one accepts too broad a concept of "property" (instead of set), more specifically if one accepts that the scope of a property may extend to all objects, including properties. Just replace every set by its characteristic property in the above reasoning. We then consider properties A which are false when applied to themselves and we define: $R(A) \overset{\text{def}}{=} \neg A(A)$, which has $\forall A\ R(A) \Leftrightarrow \neg A(A)$ as a consequence. Taking $A = R$ we deduce the absurd $R(R) \Leftrightarrow \neg R(R)$.

We will see in the following that several solutions have been proposed in order to avoid paradoxes. For the moment, let us just mention that the most celebrated in mathematics is the axiomatic set theory of Zermelo–Fraenkel. However, as it is an untyped theory, it is not well suited to computer science. This explains why specification languages based on set theory, such as Z and B, introduce an additional typing mechanism.

In summary, logic provides concepts and tools that allow us to understand the benefits, limitations and design issues of specification and programming languages. One has to pay attention to two pitfalls:

- a lack of expressiveness may lead to complications in using a language; for instance, it is sometimes just impossible to state the properties we wish to verify;
- conversely, some powerful constructs which seem correct at first sight may turn out to be much too powerful; that is, in the case of a property language, the underlying logic may become inconsistent; or, in the case of a programming language, they may lead to run-time errors which are difficult to analyze.

3.2 Antecedents

From an historical perspective, mathematical logic emerged a century ago for the purposes of precisely and rigorously constructing the foundations of mathematics. It was known, since the times of Dedekind and Cantor, that all mathematical objects (numbers, functions, vectors and so on) could be constructed from natural integers using only set-theoretic operations. However, those operations, when defined in an intuitive way, allowed one to derive paradoxes such as Russell's paradox. The whole mathematical edifice was threatened, leading to the "foundation crisis", and then to an intensive activity aiming at establishing common reasoning principles, such as deductive or inductive reasoning, on firm ground. This was one of the main motivations for David Hilbert to put forward his well-known programme, that would (in principle) reduce mathematics to finite manipulation of symbols.

A number of techniques invented in this framework happen to fit well with the needs of computer science, because, on the one hand, symbol manipulation plays a central role and, on the other hand, manipulated objects (both programs and data) are of finite or countable size (see § 3.4.6). Among theories born at that time, and which are of interest to us, we can cite predicate logic, type theory, axiomatic set theory, the λ-calculus, and intuitionistic logic. If we add the works of the 1930s on proof theory (Gentzen and Herbrand) we can see that the foundations of modern programming were largely available before the birth of the first computer!

On the mathematical side, things took an unexpected path in 1931, when Kurt Gödel proved his famous incompleteness theorem for arithmetic, sounding the death-knell of Hilbert's programme. To put it in a concrete way, it means that the most secure and restrictive reasoning forms are not strong enough to justify the principle of induction, not even to mention the stronger axioms contained in Zermelo–Fraenkel set theory. However, the latter turned out to be sufficiently powerful to serve as a basis for all known mathematics, and it is unlikely that an inconsistency will be discovered in it. The Zermelo–Fraenkel system remains the most commonly used nowadays.

3.3 The Different Branches of Logic

3.3.1 Model Theory

There are basically two complementary ways of writing a specification:

– describing the properties of a system;
– providing a model of the system by means of built-in constructs.

One sometimes uses the terminology **property oriented** and **model oriented** formal specification. Properties are expressed by logical axioms whereas models are derived with the help of set-theoretic operations. This duality is already present in mathematical logic, where we have a syntax for expressing logical properties and a semantics describing what we are talking about. This aspect of logic is called **model theory**. One distinguishes, on the one hand, the concept of a logical statement built upon a formal language, for example:

$$\forall x \exists y (y > x) \ , \tag{3.1}$$

and on the other hand the concept of a model satisfying this statement; for instance, (3.1) admits, among other models, $\mathbb{N}$ endowed with the relation "greater-than", $\mathbb{R}$ endowed with the relation "less-than" and $\mathbb{N}$ endowed with the relation "is-a-multiple-of".

A fundamental concept of model theory is the relation called **logical consequence** or **semantic consequence**. A sentence E is a semantic consequence of the sentences A, B, C... if *every* model having the properties A, B, C... has also the property E. This is a very concrete relation. Let us consider, for instance, the three properties "every terminal is a piece of equipment", "every piece of equipment possesses a registration number" and "this phone is a terminal". A practical consequence, of interest to the department in charge of inventories, is that in any situation where the above three properties hold true, we have, systematically, "this phone possesses a registration number". The concept of model is represented here by what we just called a situation.

3.3.2 Proof Theory

However, the concept of semantic consequence suffers from a big handicap: it is very difficult or even impossible to check it directly, because we must consider every possible model and there is, in general, an infinite number of them. This is why one may prefer to use another relation called **provability**. We say that a sentence E is provable from the sentences A, B, C... if we can construct a formal proof of E using only hypotheses A, B, C... in combination with axioms and the rules of logic. E is **refutable** if its negation is provable.

Of course, the logician must ensure that those formal manipulations respect the semantics, hence the concept of **soundness**. The converse property (every semantic consequence is provable) is a form of **completeness**. Another kind of **completeness** relates a collection of formulas Γ with *one* intended model $\mathcal{M}$, stating that the latter is completely characterized by Γ, i.e. every true (respectively false) formula in $\mathcal{M}$ is provable (respectively refutable) from Γ.

If we consider the formal specification of a piece of software, we can easily admit a specification to be incomplete at a high level stage. We only expect that the operations of our software respect a number of constraints, expressed by the means of logical formulas, but we may want to leave several options open. For instance, if we specify the calculation of $\sqrt{2}$ with a tolerance of 10^{-3}, the programmer is free to provide an implementation computing any result between $\sqrt{2} - 10^{-3}$ and $\sqrt{2} + 10^{-3}$. In many protocol specifications, some messages have to be answered in a very precise manner while others are considered less important. Sometimes we cannot afford incompleteness: in security software, all possible cases must be handled.

Apart from the links between semantic consequence and provability, there are interesting issues concerning provability alone. For example: if we know that E is provable, can we find a proof of E using only sub-formulas of E? If the answer is yes, the proof search space can be significantly restricted. This is especially important for automated proof tools. The study of axiom sets and logical rules, seen as formal calculations (by this we mean purely syntactic manipulations where we forget how formulas are interpreted) and their relationship with the concept of semantic consequence are the realm of **proof theory**.

In model theory, the semantics of logical sentences is provided by truth values. This is sometimes called the **Tarskian** tradition, in honor of the logician Tarski who deeply clarified its basis. Proof theory provides a different semantic perspective, which is in some sense more accurate, where logical sentences are associated with a set of proofs that conclude to these sentences, instead of to a simple value (true or false). This set of proofs can also be seen as a set of algorithms. This tradition is sometimes called **Heytingian** [GLT89].

The aim of Heyting was to interpret **intuitionistic logic** invented by Brouwer in a formal manner, during the foundation crisis. (At the same time normal logic was termed **classical logic**.) Intuitionistic logic contests the validity of a number of laws. The most well-known of these is the law of the excluded middle, which is formally stated as $p \vee \neg p$. Let us first point out that some consequences of this law are somewhat unexpected, for instance: "when you cast a dice, if you get an even result then it is smaller than three, or conversely". Formally, $p \Rightarrow t \vee t \Rightarrow p$ is accepted by classical logic but rejected by intuitionistic logic. We will see in § 3.7.3 another surprising example which is related to recursive functions. More deeply, the excluded middle is rejected because of a new interpretation of disjunction: in order to accept $p \vee q$, intuitionists want to know which proposition is provable amongst p and q. More precisely, it is enough for them to have the capability to compute the answer to that question. Then they can accept some instances of $p \vee \neg p$, but not any one.

In order to illustrate the difference in points of view, let us take a situation x in a game of chess and let $r(x)$ denote the fact that the black king is in check and in the situation x. An intuitionist can accept the sentence $r(x) \vee \neg r(x)$ because, by a mechanical application of the rules of the game (the explicit definition of $r(x)$) we can know whether the black king is in check in the situation x. Such reasoning remains valid in classical logic, of course. But in this framework we can also conclude this immediately using the law of the excluded middle. We can see that the explanation required by the intuitionist provides much more information.

The existential quantifier is interesting as well. In order to prove $\exists x\, P(x)$, the intuitionist wants to know, or to be able to compute a witness, x satisfying $P(x)$. A proof that the hypothesis $\neg\exists x\, P(x)$ leads to a contradiction, for instance, is not sufficient.

Simple common situations, where the law of the excluded middle is rejected by intuitionists, can be expressed in the form $(\exists n\, p(n)) \vee \neg(\exists n\, p(n))$, or, equivalently, $(\exists n\, p(n)) \vee (\forall n\, \neg p(n))$, where p is a property of natural numbers for which it is unknown whether, or not, there exists an n such that $p(n)$. Even if we have a mechanical procedure for deciding, for any given n, whether $p(n)$ holds or not — formally: even if we know $\forall n\, p(n) \vee \neg p(n)$ — the obvious algorithm for testing $(\exists n\, p(n)) \vee (\forall n\, \neg p(n))$, which successively checks whether, or not, $p(0)$, $p(1)$, ..., would involve an infinite number of tests if p happens to be false everywhere. As this algorithm may not terminate, it cannot be considered as reliable for providing an answer to our question. Suitable properties p can be constructed from unsolved mathematical conjectures. So-called Brouwerian arguments use, typically, the existence of 100 consecutive '9's arbitrarily far into the decimal expansion of the number π.

Intuitionistic logic has important uses in computer science because of its constructive features. In particular, there is a close relationship with type systems which we consider in Chapter 11.

3.3.3 Axiomatic Set Theory and Type Theory

A real model, like the one considered above in the example of telephone equipment, is not quite conventional in model theory. One merely considers mathematical structures, that is, sets endowed with particular operations. The samè is true in computer science: in a model-oriented technique, models are written using set-theoretical constructs, though they are much less sophisticated than in model theory. For instance, one would consider an abstract set of equipment, having the same relation with reality as data structures of the corresponding software.

In order to be able to reason in a safe manner, building blocks for such sets need to be well defined. However, we know that the problems raised are not trivial. Several solutions have been proposed for eliminating the paradoxes of "naïve" set theory.

3.3.3.1 Typing Formulas. The most ambitious solution was proposed by Bertrand Russell [vH67, p. 199]. His idea was to introduce ***types*** in order to prohibit expressions like $x \in x$, or any expression which would yield the latter after a calculation. Actually **type theory** was not an attempt to save set theory or to reconstruct it on safe ground, but rather a new approach to establishing the foundations of mathematics. The first versions of type theory turned out to be unsatisfactory because they imposed inconvenient restrictions and some axioms were ad hoc. The idea has been significantly reshaped since then, expecially following the work of Martin-Löf [ML84], and a fair amount of mathematics can now be developed in a typed framework.

Ideas progressed in a similar way in computer science, and even more successfully: the first typing systems, for languages such as Pascal, proved to be too restrictive. But, subsequent progress led to programming languages that are both convenient in practice and strongly typed (notably, languages of the ML family).

A number of important ideas came to light with typing, such as the idea of **stratification**. Typing, at least in its most elementary form, stratifies sets (and properties) in distinct layers: at layer 0, individuals; at layer 1, sets of individuals (and properties about individuals); at layer 2, sets of sets of individuals (and properties of sets of individuals); and so on. Distinguishing first-order logic, second-order logic, etc. (see below) comes directly from this idea. This kind of typing is called **predicative**, which means that in order to define a concept, only concepts defined in lower layers can be used.

We find something analogous in computer science, when a software system is structured into layers. A function or a procedure which is defined using only previously defined functions and procedures can also be qualified as predicative. Note that in computer science we generally use the terminology **recursive** instead of impredicative: saying that a recursive function is defined as "a function of itself", a paradoxical way of stating things, is precisely recognizing that this function has an impredicative definition. There is a clear motivation to use only predicative definitions in logic: paradoxes like Russell's are then avoided. Note that in our presentation in § 3.1.3, the set R is impredicatively defined.

3.3.3.2 Axiomatizing Set Theory. The other attempt to suppress paradoxes consisted of defining **set theory** using an axiomatic form, in the framework of predicate logic. The main inventors were Zermelo,[5] Fraenkel and Skolem. In Shoenfield's presentation in [Bar77, ch. B.1], the idea of stratification appears quite clearly. This can explain why the well-known paradoxes could not be reproduced. One of the most important points concerns the definition of a set by comprehension, that is, by the means of a characteristic property of its elements. An axiom, called the separation axiom, states that we can form a set by comprehension *only* if we first have a sufficiently large set where we take elements having the desired property. As a consequence, we cannot directly define $A \cup B$ as the set of elements x such that $(x \in A) \vee (x \in B)$.

Thanks to this axiomatization, it proved possible to retrieve the ingredients provided by the "naïve" theory of Cantor, that were needed for developing the desired mathematical concepts, and hence its quick operational success.

3.3.4 Computability Theory

A last part of logic is the study of **computable functions**, that is, functions which can actually be defined by computations. This is an intuitive concept which must be formalized in order to become workable. Several proposals were made in the 1940s, among others: Turing machines, λ-calculus (Church) and recursive functions[6] (Gödel, Herbrand). Each of these approaches is a way of formalizing the concept of an algorithm, and in essence, defines a primitive programming language.

A simple reasoning on set sizes shows that many functions are not computable.[7] Moreover, it turned out that all aforementioned formalisms represent exactly the same class of functions: for instance we can encode any partial recursive function with a Turing machine and vice versa. The concept of a computational process seems then to be faithfully represented by any of these formalisms. This postulation is known as the **Church thesis**. To date, it has never been shown to be wrong.

[5]Zermelo's first paper on this topic was published the same year as the one by Russell on type theory, cf. [vH67].

[6]Note that the meaning of "recursive" in logic is precise but, unfortunately, different from its meaning in computer science. We saw that the latter corresponds rather to "impredicative". The definition of "recursive" is given at the end of this chapter.

[7]If we restrict ourselves to functions over natural integers (without loss of generality, because all useful data structures can be encoded by integers), the set of functions from $\mathbb{N}$ to $\{0, 1\}$ — and *a fortiori* to $\mathbb{N}$ — is not countable, whereas the set of functions defined by the means of a language having a finite or countable vocabulary is countable.

As a consequence, a programming language is said "to have the power of Turing machines" if it has the maximal expressive power we can expect — but it does not tell us whether this language is easy or difficult to use. Informally, a **Turing machine** is composed of an internal state, a tape with an infinite number of squares, a read-write head and instructions used to move the head and/or to write on the current square according to the current state and the symbol present on the current square. All common programming languages, including assembly languages, have the power of Turing machines. Among formalisms which do *not* have the power of Turing machines, we can cite finite state automata (which can parse or generate regular languages) and push-down automata (which can parse or generate context-free languages). Roughly, in order to get the power of Turing machines, the key ingredients are:

- basic arithmetic operations (addition, compare to zero);
- a notion of loop where the exit condition is computed at each iteration (e.g. the `while` of Pascal in comparison with the `for`);
- unbounded memory space; note that only a finite amount of memory is available on real computers, but the difference is hardly perceptible in practice.

Once this class of functions came to light, a number of fundamental questions could be asked and sometimes solved. The most well known of them is **the halting problem of Turing machines**: can we *mechanically* and in a *finite* number of steps, decide whether an arbitrary Turing machine running on arbitrary input data will eventually reach the state "computation end"? To put it in other words, can we know in advance — say, at compile time — whether or not the execution of a program will end, or whether a partial recursive function is defined on a given input data? It can be shown that this problem is actually **undecidable**, which means that no Turing machine can compute the answer to this question. As a practical consequence, a computer that could tell in advance whether an arbitrary program "loops" or not is definitely magical.

Notes. When we try to prove the correctness of algorithms, proving their termination is a crucial issue. The aforementioned result does not prevent us from doing it, it just states limitations on the extent of the help that we can expect from automation.

Note that this undecidability result came after the incompleteness theorem of Gödel; it is, moreover, proven along similar lines. Decidability and completeness are actually strongly related questions.

Computability (or recursion) theory comprises many other technical results that are not covered in this book. Their impact on formal methods is, in any case, quite weak nowadays.

3.4 Mathematical Reminders

We recall here useful basic concepts of set theory, logic and algebra.

3.4.1 Set Notations

In the following, A, B, $C\ldots$ denote "sets" whereas a, b, $c\ldots$ denote "elements"; we use quotes because the concepts of "elements" and sets are in fact relative, the members of a set can quite acceptably be sets themselves.

A **singleton** is a set having exactly one element, such as $\{a\}$. An **unordered pair** is a set having exactly two elements, such as $\{a, b\}$. The set with no elements is denoted by $\varnothing$. Two sets are **disjoint** if their intersection is empty.

We say that A **is included** in B, and written $A \subset B$, if every element of A is also an element of B. In particular, we have $A \subset A$ and $\varnothing \subset A$ for any set A. If A is included in B, we also say that A is a **subset** of B and that B is a **superset** of A. Two sets A and B are **equal** if they contain exactly the same elements. Hence $A = B$ if and only if $A \subset B$ and $B \subset A$. A is **strictly included** in B if $A \subset B$ and $A \neq B$. Then A is also called a **proper subset** of B. The set of subsets of A is called the **powerset** of A, it is denoted by $\mathcal{P}(A)$ or 2^A.

The **union**, the **intersection** and the **Cartesian product** of two sets were previously introduced on page 20. The **Cartesian square** of A is $A \times A$. The **difference** $A - B$ is the set of elements which are members of A but not of B. The **symmetric difference** $A \setminus B$ is the set of elements which are members of either A or B (but not A and B). Thus $A \setminus B = (A \cup B) - (A \cap B)$.

The set A^n denotes the Cartesian product $A \times A \ldots \times A$ (with n occurrences of A), i.e. the set of n-tuples $\langle a_1, \ldots, a_n \rangle$ such that $a_i \in A$. A^1 is identified with A. We agree that A^0 is the singleton $\{\varnothing\}$ — another singleton would do the job just as well, this one is the most simple we can construct in a universe where no element is known *a priori*.

Besides definitions by extension introduced on page 20, it is possible to define a set by **comprehension**, i.e. by providing a characteristic property of its elements. We use $\{x \mid P(x)\}$ to denote the set of elements x such that $P(x)$, and $\{x \in E \mid P(x)\}$ to denote the set of elements x which are members of E and such that $P(x)$. The second form is better because the first can lead to paradoxes.

3.4.2 Logical Operators

Tables 3.1 and 3.2 summarize the intuitive meaning of logical operators as well as their relation to set-theoretic operations. These intuitions will be developed and explained in subsequent chapters.

The meaning of conjunction $\wedge$ and of negation $\neg$ is just the one you would expect. The same is true of disjunction $\vee$ as well, but be aware that we have a *non exclusive or*. Interpreting implication $P \Rightarrow Q$ must be done with greater caution: nothing tells us that there is an actual causality relation between P and Q. We can only say

Table 3.1

t	true
f	false
$\neg$	not
$\wedge$	and
$\vee$	or
$\Rightarrow$	implies
$\Leftrightarrow$	is equivalent to
$\forall$	for all
$\exists$	exists

Table 3.2

E		$x \in E$
A, B		P, Q
$A \cap B$	inter	$P \wedge Q$
$A \cup B$	union	$P \vee Q$
$A - B$	minus	$P \wedge \neg Q$
$A \setminus B$	symmetric difference	$\neg(P \Leftrightarrow Q)$
$\varnothing$	empty set	**f**

that Q happens to be true when P is true. Thus $\forall x\, R(x) \Rightarrow S(x)$ means that all x verifying R verify S as well. If no x verifies R, we agree that the formula $\forall x\, R(x) \Rightarrow S(x)$ is true. We have, therefore, in this case $\forall x\, \mathbf{f} \Rightarrow S(x)$ and, as $S(x)$ may be true or false, we see that both $\mathbf{f} \Rightarrow \mathbf{t}$ and $\mathbf{f} \Rightarrow \mathbf{f}$ are true.

The logical equivalence $P \Leftrightarrow Q$ is an abbreviation for the conjunction $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$. It behaves like an equality; hence we can replace P with Q when $P \Leftrightarrow Q$. Table 2 above can read: $x \in A \cap B \Leftrightarrow x \in A \wedge x \in B$, etc., $x \in \varnothing \Leftrightarrow \mathbf{f}$.

Numerous logical laws can be stated using equivalences. For instance, consecutive conjunctions can be reordered with $P \wedge Q \Leftrightarrow Q \wedge P$ and $(P \wedge Q) \wedge R \Leftrightarrow P \wedge (Q \wedge R)$. The same is true for disjunction. We have also $P \wedge \neg P \Leftrightarrow \mathbf{f}$ and $P \vee \neg P \Leftrightarrow \mathbf{t}$.

The constants **t** and **f** can be eliminated using $P \wedge \mathbf{t} \Leftrightarrow P$, $P \vee \mathbf{f} \Leftrightarrow P$, $P \wedge \mathbf{f} \Leftrightarrow \mathbf{f}$ and $P \vee \mathbf{t} \Leftrightarrow \mathbf{t}$. Hence we see that $x \in \varnothing \Leftrightarrow \mathbf{f}$ boils down to $x \in \varnothing \Rightarrow \mathbf{f}$.

Here are other very useful identities:

$$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R) \tag{3.2}$$
$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R) \tag{3.3}$$
$$\neg\neg P \Leftrightarrow P \tag{3.4}$$
$$P \Rightarrow Q \Leftrightarrow \neg P \vee Q \tag{3.5}$$
$$P \Rightarrow \mathbf{f} \Leftrightarrow \neg P \tag{3.6}$$
$$\neg(P \wedge Q) \Leftrightarrow \neg Q \vee \neg P \tag{3.7}$$
$$\neg(P \vee Q) \Leftrightarrow \neg Q \wedge \neg P \tag{3.8}$$
$$\neg \forall x\, P(x) \Leftrightarrow \exists x\, \neg P(x) \tag{3.9}$$
$$\neg \exists x\, P(x) \Leftrightarrow \forall x\, \neg P(x) \tag{3.10}$$
$$(P \wedge Q) \Rightarrow R \Leftrightarrow P \Rightarrow (Q \Rightarrow R) \tag{3.11}$$
$$(P \wedge U) \Rightarrow Q \Leftrightarrow P \Rightarrow (\neg U \vee Q) \tag{3.12}$$

For example, using (3.6), the last line of Table 2 is equivalent to $\neg x \in \varnothing$: as expected, no element can be a member of $\varnothing$. The laws (3.7) to (3.10), called De Morgan's laws, allow us to distribute negation across other connectives. The equivalence (3.11) provides two ways for expressing "if I have P, if I have Q then I have R". This can also be written $P \Rightarrow Q \Rightarrow R$. Using (3.5) we get the equivalence (3.12) that allows us to move a formula U to the opposite side of an implication at the price of a negation.

EXERCISE. Show the equivalence $(P \vee Q) \wedge \neg(P \wedge Q) \Leftrightarrow \neg(P \Leftrightarrow Q)$. Justify $A \setminus B \Leftrightarrow \neg(P \Leftrightarrow Q)$, where $P \stackrel{\text{def}}{=} x \in A$ and $Q \stackrel{\text{def}}{=} x \in B$, from $A \setminus B = (A \cup B) - (A \cap B)$.

EXERCISE. Find the logical laws used in the reasoning on page 29 for proving partial correctness of the first bounded linear search program.

3.4.3 Relations and Functions

A (binary) **relation** R from A to B is a subset of $A \times B$. Its elements, which are ordered pairs $\langle a, b\rangle$ with $a \in A$ and $b \in B$, are also denoted by $a \mapsto b$. Then we say that a **is related to** b or that a **maps to** b by R. We often use the infix notation aRb instead of $\langle a, b\rangle \in R$.
A simple example of a relation is the **identity relation** on a set A, which is the set of all ordered pairs $a \mapsto a$ such that $a \in A$.
A relation R on A is **reflexive** if for all x in A, xRx.
It is **symmetric** if $\forall x, y \in A,\ xRy \Rightarrow yRx$.
It is **anti-symmetric** if $\forall x, y \in A,\ (xRy \wedge yRx) \Rightarrow x = y$.
It is **transitive** if $\forall x, y, z \in A,\ (xRy \wedge yRz) \Rightarrow xRz$.
An **equivalence relation** is a reflexive, symmetric and transitive relation. An **order** is a reflexive, anti-symmetric and transitive relation. An order is **total** when two elements can always be compared: $\forall x, y \in A,\ xRy \vee yRx$. In the opposite case (or if we don't know) we have a **partial** order.
If R is an order on A and if B is a subset of A, an element m of A is a **lower bound** (respectively an **upper bound**) of B if $\forall b \in B\ mRb$ (respectively $\forall b \in B\ bRm$).

A relation R from A to B is **defined at** a with $a \in A$, if there exists an ordered pair $a \mapsto b$ in R, i.e. if a is mapped to an element of B by R. The **domain** of R is the set of elements a such that R is defined at a. R is a **total** relation if its domain is A. In the opposite case (or if we don't know) we say that R is **partial**. The set of total functions from A to B is denoted by $A \rightarrow B$.

A **function** f from A to B is a relation such that if $x \mapsto y_1$ and $x \mapsto y_2$ are members of f, then $y_1 = y_2$ (intuitively, applying a function to a given element always yields the same result). If $x \in A$ and if x is in the domain of f, we denote fx or $f(x)$ the unique element y of B such that $x \mapsto y$ is a member of f.

The **composition** of two functions f and g from B to C and from A to B, respectively is the function from A to C denoted by $g \circ f$ such that $(g \circ f)x = g(fx)$. This definition generalizes if f and g are relations. In that case $x \mapsto z$ is a member of $g \circ f$ if and only if there exists a y in B such that $x \mapsto y$ is a member of f and $y \mapsto z$ is a member of g.

The first **projection** p_1 is the function from $A \times B$ to A defined by $p_1\langle a, b\rangle = a$. Similarly the second projection p_2 is defined by $p_1\langle a, b\rangle = b$. More generally, the ith projection is the function from $A_1 \times \ldots A_i \times \ldots$ to A_i defined by $p_i\langle a_1, \ldots a_i, \ldots\rangle = a_i$.

A function is **injective** if distinct elements are mapped to distinct elements. A function f from A to B is **surjective** if all elements of B are mapped by f. A **bijection** is a total, injective and surjective function.

3.4.4 Operations

An **operation** $\star$ on the set A is a total function from $A \times A$ to A. It is **commutative** if for all x, y of A we have $x \star y = x \star y$. It is **associative** if for all x, y, z of A we have $(x \star y) \star z = x \star (y \star z)$.

The element e is called an **left identity** element of $\star$ (respectively a **right identity** element) if for all x in A we have $e \star x = x$ (respectively $x \star e = x$). The element a is called a **left absorbing** element of $\star$ (respectively a **right absorbing** element) if for all x in A we have $a \star x = a$ (respectively $x \star a = a$). The element x' is called a **left inverse** (respectively **right inverse**) of x if $x' \star x = e$ (respectively $x \star x' = e$). An **identity element** (respectively an **inverse**, an **absorbing element**) is a left and right identity (respectively inverse, absorbing) element.

Notation: when the underlying operation $\star$ is clear from the context, it is often omitted: one writes xy instead of $x \star y$. If $\star$ is associative, one also writes x^n for $x \star \ldots \star x$ (with n occurrences of x). The inverse of an element x (when it exists) is denoted by x^{-1}.

Example: given a set A, let $\mathcal{R}_A$ denote the set of relations on A. Then $\circ$ is an operation on $\mathcal{R}_A$, with the identity relation as an identity element. The **inverse** of a relation R (written R^{-1}) is then the set of ordered pairs $y \mapsto x$. such that $x \mapsto y$ is in R. A function is injective if and only if the inverse relation is a function. A function is surjective if and only if the inverse relation is total. A function is bijective if and only if the inverse relation is a total function.

An element x is said to be **idempotent** if $x \star x = x$. The operation $\star$ is **idempotent** if all elements of A are idempotent.

EXERCISE. The connectives $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ can be seen as operations on $\mathbb{B}$ (see § 5.1.3). Which of them are commutative? Associative? Idempotent? Which ones possess an identity element? An absorbing element? Invertible elements? Do not neglect $\Leftrightarrow$.

3.4.5 Morphisms

Let us consider the set of natural integers endowed with addition and the identity element 0 on the one hand, the set of natural integers endowed with multiplication and the identity element 1 on the other hand. The function φ which maps n in $\mathbb{N}$ to 3^n preserves the identity element and the operation in the following sense: $\varphi(0) = 1$ and $\varphi(m + n) = \varphi(m) \times \varphi(n)$. We say that φ is a morphism from $\langle \mathbb{N}, +, 0 \rangle$ to $\langle \mathbb{N}, \times, 1 \rangle$.

Let us consider a more general case. We take a set E endowed with a function f, an operation $\star$, and a relation R. This structure is denoted by a 4-tuple: $\langle E, f, \star, R \rangle$. Let us take a similar structure $\langle E', f', \star', R' \rangle$. A **morphism** of $\langle E, f, \star, R \rangle$ to $\langle E', f', \star', R' \rangle$ is a function φ from E to E' which preserves the structure in the following sense. Let x, y, z be arbitrary elements of E and let x', y', z' their respective targets by φ: thus we have $x' = \varphi(x)$, $y' = \varphi(y)$ and $z' = \varphi(z)$. The function φ is a morphism if:

- φ preserves the function: if $y = f(x)$, then $y' = f'(x')$;
- φ preserves the operation: if $z = x \star y$ then $z' = x' \star' y'$;
- φ preserves the relation: if xRy then $x'R'y'$.

An **isomorphism** is a bijective morphism. Two structures are **isomorphic** if they are related by an isomorphism. Intuitively, we can to a fair extent agree that they are identical because they have exactly the same properties.

3.4.6 Numbers

Common number sets ($\mathbb{N}$, $\mathbb{Z}, \mathbb{Q}$ and $\mathbb{R}$) are recalled on page 22. Natural numbers can be generated from the empty set using the following encoding: 0 is encoded by $\{\} = \varnothing$, 1 is encoded by $\{0\} = \{\varnothing\}$, 2 is encoded by $\{0, 1\} = \{\varnothing, \{\varnothing\}\}$, ... n is encoded by $\{0, \ldots n-1\}$.

It is not as obvious as it may seem to define what is a finite or an infinite set. A first idea could be to count its elements and to say that the set E is infinite if there is an injection (an injective function) from $\mathbb{N}$ to E. In fact, the "axiom of infinity" stated in Chapter 7 says that there is a set containing $\mathbb{N}$. We can avoid the reference to $\mathbb{N}$ in the following way: a set E is said to be **infinite** if and only if there is a bijection from E to a proper subset of E.

A set E is **countable** if there exists a sequence (u_n) of elements of E covering E, or, equivalently, if there exists a surjective function from $\mathbb{N}$ to E (intuitively: we can count the elements of E). For example, finite sets, $\mathbb{N}$ itself, $\mathbb{Z}$, $\mathbb{Q}$ $\mathcal{P}_F(\mathbb{N})$ (the set of **finite** subsets of $\mathbb{N}$) are countable. Among sets that are not countable we have $\mathbb{R}$ and $\mathcal{P}(\mathbb{N})$ (the set of all subsets of $\mathbb{N}$). Here is an important example for computer science: a set whose elements can always be denoted by a finite sequence of characters taken in a finite alphabet is countable. In particular, the set of programs defined in all programming languages is countable, whereas the set of functions on natural numbers is not countable.

A collection where the element can be repeated is called a **family** or a **multiset**. Formally, if E is a set, a family of elements of E is a total function from E to $\mathbb{N}$.

3.4.7 Sequences

A sequence $u_0, u_1, \dots u_n, \dots$ of elements u_i of E is a total function from $\mathbb{N}$ to E: u_n is just another notation for $u(n)$. A sequence can be defined directly (for example $v_n = n^2$) or by **induction**, by providing the value of u_0 and a function yielding the value of u_{n+1} from u_n (for example $u_0 = 0$ and $u_{n+1} = u_n + 2n + 1$). The typical way of proving properties of such sequences is through proof by induction. On the last example it is easy to prove: $\forall n \, u_n = v_n$.

We sometimes need to talk about sequences that are finite or infinite. We mean, total functions from A to E, where A is either a subset of $\mathbb{N}$ of the form $\{n \in \mathbb{N} \mid n < a\}$ for a given natural number a, or $\mathbb{N}$ itself. We will then use the explicit terminology "finite or infinite sequence", "finite sequence" when A has the first form and "infinite sequence" when A has the second form. In other contexts "sequence" will always denote an infinite sequence.

3.5 Well-founded Relations and Ordinals

3.5.1 Loop Variant and Well-founded Relation

We have seen in § 2.4.1.2 that the termination of a program can be studied by considering a quantity v that decreases at each step while staying in $\mathbb{N}$. Let us emphasize the last point. It is not enough to ensure that the variant v is a decreasing number:

- an integer can decrease *ad vitam æternam* by taking arbitrarily large negative values;
- a positive rational or real number can decrease while approaching a lower limit without reaching it.

The point is that v must take a *finite* number of values. Reasoning with a "decreasing number" is of course an incorrect wording, which has to be formalized with a finite or infinite sequence v_0, v_1, ...v_n, ... as we will see below.

In order to model the problem of termination, let us first consider the set S of the values that can be taken by the state of a program.[8] The change in this state is observed at certain points between which we admit that nothing important can happen.[9] Each execution step corresponds to a state transition which is modeled as an ordered pair $\langle s_i, s_f \rangle$ where s_i and s_f, the value of the state respectively at the beginning and at the end of the transition, are

[8]For the sake of completeness we should include in the state a component for the program counter and another for the execution stack. We proceed in this manner in order to define an operational semantics.

[9]We can choose fine grain observation, corresponding to elementary instructions or large grain observation, corresponding to blocks of such instructions: the point is that executing those "grains" always terminates.

members of $\mathcal{S}$. We then introduce the set of transitions $\mathcal{T}$, which is a relation on $\mathcal{S}$.

When we reason with a variant v, the latter is a function of the state s. Each transition $\langle s_i, s_f \rangle$ at the level of states corresponds to a transition $\langle v(s_i), v(s_f) \rangle$ at the level of the variant. The general situation is then captured by a set $\mathcal{S}$ endowed with a relation $\mathcal{T}$.

The changes of the state during an execution beginning at initial state s_0 are then modeled by a finite or infinite sequence s_0, s_1, ...s_n, ... such that two consecutive elements s_k and s_{k+1} are always related by $\mathcal{T}$. Ensuring the termination of the program boils down to prohibiting the sequence from being infinite. For example, in the case of natural integers, there is no infinite sequence v_0, v_1, ...v_n, ... such that $v_0 > v_1 > ... > v_n > ...$, which allowed us to justify the technique of the variant on page 24. When no such sequence exists the relation is said to be **Noetherian.** We can similarly consider the inverse relation (recall that, for instance, $<$ and $>$ are inverse relations). We then have a well-founded relation. Let us develop this concept.

Let E be a set and R a relation on E. Let x and y be two elements of E, we say that x is a **predecessor of y for R** if xRy. When there is no ambiguity we simply say that x is a predecessor of y. A **chain** is a finite or infinite sequence e_0, e_1, ...e_n, ... of elements of E such that e_{n+1} is always a predecessor of e_n: $\forall n \in \mathbb{N} \;\; e_{n+1} R e_n$. R is a **well-founded** relation if R contains no infinite chains.[10]

The concept of predecessor that we use here generalizes from the usual one on integers: just take for R the relation noted R_1 below. For an arbitrary relation R, the predecessor of an element, when it exists, need not be unique.

In summary, expressing that a program terminates boils down to saying that the underlying transition relation $\mathcal{T}$ is Noetherian, or that the inverse relation $\mathcal{T}^{-1}$ is well founded. In practice, instead of reasoning directly on the set of states $\mathcal{S}$ endowed with $\mathcal{T}^{-1}$, it is worth considering a simplified view E of $\mathcal{S}$ endowed with a corresponding relation R, which must be well founded as well. The loop variant presented in the above example amounts to taking $\mathbb{N}$ for E and $<$ for R.

3.5.2 Examples

The relation $<$ is well founded on $\mathbb{N}$, but is not well founded on $\mathbb{Z}$, nor on any interval of $\mathbb{R}$ or of $\mathbb{Q}$. Any relation included in a well-founded relation is also well founded. Hence all sets of ordered pairs of natural integers $\langle m, n \rangle$ verifying $m < n$ are well founded. Here are three examples:

$$R_1 \stackrel{\text{def}}{=} \{\langle n, n+1 \rangle \mid n \in \mathbb{N}\}$$

[10]Nothing prevents the repetition of an element in a sequence. If x is such that xRx, the sequence x, x, ... x, ... is then an infinite chain. If x and y satisfy xRy and yRx, the sequence x, y, ... x, y, ... is an infinite sequence as well.

$$R_2 \overset{\text{def}}{=} \{\langle n, 2n{+}\varepsilon\rangle \mid n \in \mathbb{N} \wedge n{>}0 \wedge (\varepsilon{=}0 \vee \varepsilon{=}1)\}$$
$$R_3 \overset{\text{def}}{=} \{\langle n, n{+}2\rangle \mid n \in \mathbb{N}\}$$

The relation $>$ is not well founded on $\mathbb{N}$, but it becomes so on a finite subset of $\mathbb{N}$. As a consequence, relations having the form $R_4(q)$ are well-founded:

$$R_4(q) \overset{\text{def}}{=} \{\langle n{+}1, n\rangle \mid n \in \mathbb{N} \wedge n{<}q\}$$

Here is a very important example. Let R be a well-founded relation on E and let S be a well-founded relation on F, the relation defined over $E \times F$ by

$$\begin{aligned} R_5 \overset{\text{def}}{=} & \{\langle\langle x,u\rangle, \langle x',v\rangle\rangle \mid x,x' \in E \wedge u,v \in F \wedge xRx'\} \;\cup \\ & \{\langle\langle x,u\rangle, \langle x,u'\rangle\rangle \mid x \in E \wedge u,u' \in F \wedge uSu'\} \end{aligned}$$

is well founded. This construction corresponds to the **lexicographic ordering** used by all of us when consulting a dictionary.[11] This example is more subtle than the previous ones. If we consider the relation $<$ on $\mathbb{N}$ (or its subsets R_1, R_2 and R_3), we have already observed that all decreasing sequences are finite. But additionally, we know an upper bound on the length of such sequences as soon as we know the first element (the latter is such an upper bound). In contrast, if we take the structure $\langle \mathbb{N}, <\rangle$ or even $\langle \mathbb{N}, R_1\rangle$ for $\langle F, S\rangle$ in R_5, it is no longer possible to give an upper bound for decreasing sequences starting from $\langle x_0, n_0\rangle$ if there is no x_1 in E such that $x_1 R x_0$. In that case there exist an infinite number of finite decreasing sequences starting from $\langle x_0, n_0\rangle$, and their length is arbitrarily large.

The lexicographic ordering on the Cartesian product of two or of any finite number of well-founded sets is well founded. Note however, that the lexicographic order on words, that is, arbitrarily large finite sequences of elements of a well-founded set E, is *not* well founded. For instance, with $E = \{0, 1\}$ and $0 < 1$, we have the infinite decreasing chain 1, 01, 001, 0001, etc.

Generalizing the technique of loop variants with well-founded relations can be useful in two ways:

1. We can acquire a knowledge of the number of iterations performed when executing a loop.
2. We can cope with more complex situations involving several loops, whether embedded or not.

3.5.2.1 Counting Iterations in a Loop. First recall that the number of iterations n_i depends on the initial value v_0 of the variant. In general, the latter depends in turn on a preliminary computation or on an external event — reading a number for example — and is then essentially unpredictable. In contrast we can ask how n_i depends on v_0.

[11] One should pay attention to the following technical point: a well-founded relation like R or S is not an order because it cannot be reflexive. We come back to the links between these concepts in § 3.5.4.1.

Let us take $\mathbb{N}$ as the domain of the variant. If the well-founded relation at hand is $<$, we only know that $n_i \leq v_0$. If the relation is R_1, we have $n_i = v_0$. If the relation is R_2, we know that n_i is close to the base 2 logarithm of v_0.

3.5.2.2 Using more Complex Well-Founded Relations. In order to study the termination of programs composed of several loops using only one well-founded relation, the domain E we have to consider for the latter has to be larger[12] than $\mathbb{N}$. Here we content ourselves with the simple case of a program made of a first loop, followed by the computation or the reading of an arbitrarily large positive integer L and finally a second loop.

Let us first consider each loop separately. Assume that the variant of the first is v in $\mathbb{N}$ endowed with R_1 whereas the variant of the second is w also in $\mathbb{N}$ endowed with R_1. For all initial values v_0 and w_0, it is intuitively clear that the program terminates since each loop terminates. If we knew in advance the value of w_0, we could take $u = v + w$ as the global variant, in the same domain $\mathbb{N}$ endowed with R_1. To be more precise, u would be defined as $v + w_0$ in the first loop, as w_0 between the two loops and as w in the second loop. But we cannot proceed in this way if the value L taken by w_0 is unknown in advance and arbitrarily large.

A satisfactory solution is to take for E the sum of two copies of $\mathbb{N}$ or, equivalently, the Cartesian product[13] $\{0,1\} \times \mathbb{N}$. The variant u is $\langle 1, v\rangle$ in the first loop, $\langle 1, 0\rangle$ between the two loops (let us call this element ω) and $\langle 0, w\rangle$ in the second loop. Our well-founded relation $R_{1,1}$ is defined by $\langle i, n\rangle R_{1,1} \langle i, n+1\rangle$ (intuitively it behaves like R_1 on each copy of $\mathbb{N}$) and $\langle 0, n\rangle R_{1,1} \omega$. $R_{1,1}$ is contained in the relation R_5 above, where we take $E = \{0,1\}$, $R = \{\langle 0, 1\rangle\}$, $F = \mathbb{N}$ and $S = R_1$.

Let us point out that, in contrast with most relations presented so far, the element (ω) admits an infinite number of predecessors in $R_{1,1}$. However, a decreasing sequence starting from any element of $\{0,1\} \times \mathbb{N}$ is necessarily finite.

A relation like $R_4(q)$ can be convenient in practice. For instance, $R_4(N)$ may be used for a direct termination proof of the bounded linear search program instead of reasoning on the difference $N - x$, as we did on page 25.

We also remark on R_2, R_3 and R_4 that it is not required that only one value (0) has no predecessor, even if we consider only natural (i.e. non-negative) numbers:[14] in R_2, we have 0 and 1; in R_3 we have 0 and all odd natural numbers; in $R_4(q)$ we have all numbers greater or equal to q. This is reflected in the loop invariant and in the exit test. For example, with R_4, we have to ensure that, at the beginning of the loop, the variant v is strictly less than q (condition ($V_<$) on page 24, reshaped with R_4, tells us that during an iteration v is necessarily incremented by 1); in this situation, we are led to put $v \leq q$ in the invariant,

[12]In a sense coming from the theory of ordinal numbers, see below.

[13]Technically we can also represent $\mathbb{N} + \mathbb{N}$ by $\mathbb{N}$ (consider even and odd numbers). But it would only make the definition of the well-founded relation more complicated with no compensation in the reasoning. The concept of ordinal presented below clarifies the situation.

[14]We choose to keep $0 \leq v$ in the invariant.

and then to take $v = q$ as the exit condition. With R_3 the exit test would correspond to $v = 0$ and the invariant would entail that v is even.

3.5.3 Well-founded Induction

Given a well-founded relation R on a set E, we can prove that a property P is true on all elements of E by showing the following proposition (H) which tells us, in familiar terminology, that P propagates:

> given any element x of E,
> if P is true on all predecessors of x, (H)
> then P is true on x.

In particular, we have to show that P is true on all x without a predecessor, which corresponds to the base cases.

This kind of reasoning is called **well-founded induction.** Usual induction on $\mathbb{N}$ is a (simple) special case of well-founded induction, where the relation considered is R_1. Assume that, despite the fact that (H) has been shown, we have an element e_0 where P is not true; e_0 has at least one predecessor, since P is true for all elements without a predecessor; by (H) we also know that P is false on at least one of the predecessors of e_0; let e_1 be one of them. Repeating the process would then yield an infinite decreasing chain e_0, e_1, ... e_n, ... , which is impossible because R is well-founded.

The previous reasoning implicitly uses a principle called the *axiom of choice*, which will be introduced in Chapter 7. Indeed, in order to construct the chain e_0, e_1, ... e_n, ... we simultaneously construct the infinite family P_0, P_1, ... P_n, ... where P_i is the non-empty set of predecessors of e_i. At each step, we have to choose e_{i+1} in P_i.

The rule of the loop is an application of well-founded induction. Let us illustrate what happens with the relation R_3. This corresponds to a loop $\mathcal{B}$ where the initial value of v is even:

while $v{\neq}0$ **do** ... $v{:=}v - 2$... **done**

We then have to show the property $P(n)$ defined by $\{v{=}n \wedge I\}\ \mathcal{B}\ \{I\}$, where I is the loop invariant. We distinguish the "true" base case $n = 0$ (corresponding to a successful exit test) from the "false" ones (odd values of v). In the latter cases $P(n)$ is trivially true by reduction to the absurd, provided we put "v is even" in the invariant.

3.5.4 Well Orders and Ordinals

We can present well-founded induction from special order relations. Here are some preliminary definitions. The main point to remember is that two isomorphic ordered sets are essentially the same up to the name of their elements. A set E endowed with an order R will be denoted by a 2-uple $\langle E, R\rangle$.

Let $\langle E, R\rangle$ and $\langle F, S\rangle$ be two ordered sets. A function f from E to F is **monotone** if the order is preserved by f:

$$\forall x, y \in E \quad xRy \Rightarrow f(x) S f(y) \ .$$

An **isomorphism** is a monotonic bijection. Two ordered sets $\langle E, R\rangle$ and $\langle F, S\rangle$ are **isomorphic** if there is an isomorphism from $\langle E, R\rangle$ to $\langle F, S\rangle$.

3.5.4.1 Well Orders. Let E be a set endowed with an order R. Given a subset A of E, a **minimum** of A, if it exists, is an element a of A such that there is no predecessor different from a in A: if $xRa \wedge x \in A$ then $x = a$ (R is reflexive!). If R is total, a minimum of A must be unique.

We say that R is a **well order** if R is total and if every subset of E possesses a minimum. Note that E possesses a unique minimum m in that case.

Let us note $R_{\neq}$, the relation defined by $x R_{\neq} y$ if and only if xRy and $x \neq y$. If R is a well order, $R_{\neq}$ is a well-founded relation. Conversely it is possible to construct a well order from a well-founded relation. But beware: a given well order can come from several well-founded relations.

The concept of a well-founded induction is defined as in § 3.5.3 if we replace R with $R_{\neq}$. The base case concerns only m. This principle can be justified as follows. Suppose that the set A of elements e which do not verify P is not empty, A possesses a minimum a which must be different from m; the predecessors of a are not members of A, hence they verify P, but with (H) we then have that P is also true of a, so a cannot be a member of A, a contradiction.

Some well orders are especially important: ordinals.

3.5.4.2 Ordinals. Let E be a set endowed with the well order R. The **section** X_a determined by an element a of E is defined as the set of elements x which are smaller than a:

$$X_a \stackrel{\text{def}}{=} \{x \in E \mid x R_{\neq} a\} \ .$$

E endowed with the well order R is an **ordinal** if for all a of E we have $X_a = a$. Thus, to verify that 3 is an ordinal, we just have to remember that in set theory $3 \stackrel{\text{def}}{=} \{0, 1, 2\}$, which actually yields $2 \stackrel{\text{def}}{=} \{0, 1\} = X_2$. The first ordinals are exactly $\varnothing$, $\{\varnothing\}$, $\{\varnothing, \{\varnothing\}\}$, etc., where the order is inclusion or, equivalently, membership (the two relations happen to coincide on ordinals).

Given an arbitrary ordinal x we can construct its successor $x \cup \{x\}$. We then start from $\varnothing$ and we construct all natural numbers step by step. The next step consists of taking $\mathbb{N}$ itself (it can be shown that $\mathbb{N}$ satisfies the required properties). $\mathbb{N}$ is traditionally noted ω in this context.

The process carries on in the same way: ω, $\omega \cup \{\omega\}$ (noted $\omega{+}1$), etc. Apart from 0 only two cases can occur for an ordinal: either it contains a greatest element, it has then the shape $x \cup \{x\}$ and it is called a **successor ordinal**; or, it does not contain a greatest element and it is called a **limit ordinal**.

The first limit ordinal is ω. The next one, noted 2ω, is the limit of $\{0, 1, \ldots \omega, \omega{+}1, \ldots\}$. Carrying on this process we define 3ω, ... $n\omega$, ... ω^2, ... ω^ω, ... ω^{ω^ω},

... until a new limit ordinal ϵ_0 which verifies $\omega^{\epsilon_0} = \epsilon_0$. There are still many other ordinals. Ordinals up to ϵ_0 are used in the automated proof assistant of Boyer–Moore and in PVS in order to formalize termination arguments [Rus93].

An important theorem about ordinals states that a well order is always isomorphic to an ordinal. Ordinals can then be used for measuring the complexity of termination proofs of algorithms. Let us also remember that the most general form of induction is well-founded induction, because the concept of a well-founded relation is finer than the concept of well order.

3.5.4.3 Ordinals and Cardinals. **Cardinals** are another concept of set theory that can be used for measuring the size of a set. We will not go into detail here. We say that two sets have the same cardinality if there exists a bijection between them. Finite sets have a cardinal 0, 1, 2, ... n with $n \in \mathbb{N}$.

Next we have $\mathbb{N}$ itself, whose cardinality is denoted $\aleph_0$ (pronounced *aleph zero*). We already know from § 3.4.6 that many infinite sets are countable: in other words, their cardinality is $\aleph_0$.

Another important point is the following. If the cardinal of a set E is α, then the cardinal of $\mathcal{P}(E)$ is strictly greater than α.

All ordinals presented so far are countable. A better wording is: the underlying sets of those ordinals are countable. We must remember that what matters in an ordinal is the corresponding order. Indeed, there are many (non-isomorphic) ways to order the elements of $\mathbb{N}$, and each of them corresponds to a different ordinal.[15] However, the order is completely irrelevant for cardinals.

In contrast to ordinals, cardinals don't seem to have applications in formal methods. Note, however, that they play an important role in set theory.

One of the first questions raised at the very beginning of development in set theory was the following: let c be the cardinal of $\mathbb{R}$; c is also the cardinal of $\mathcal{P}(\mathbb{N})$, thus we have $c > \aleph_0$; but is there an intermediate cardinal? Cantor thought that the answer should be no — this is called the *continuum hypothesis* — but the question turned out to be arduous. Gödel showed in the 1930s that this hypothesis is consistent with (i.e. cannot be disproved from) the axioms of set theory, while conversely Cohen showed in 1963 that it cannot be proven in set theory. This reveals the somewhat arbitrary character of set theory. We come back to this point at the end of Chapter 7.

3.6 Fixed Points

Let $\mathcal{E}$ be a set and f be a function from $\mathcal{E}$ to $\mathcal{E}$. A **fixed point** of f is an element x of $\mathcal{E}$ such that $x = f(x)$. For example 1 and 5 are fixed points of the function

[15]For example, if the order we consider is $<$, the corresponding ordinal is ω. However, let us consider the order R, defined by xRy if $x < y$ and $x \neq 0$, and by $x < 0$ for all x: the corresponding ordinal is $\omega + 1$. Intuitively, in the latter case, natural integers are put in the following order: 1, 2, ... 0. The two relations $<$ and R are not isomorphic since only the second one possesses a greatest element.

on $\mathbb{R}$ that maps x to $(x^2 + 5)/6$. The theorem of Knaster–Tarski states that under quite general conditions, f is guaranteed to have a least or a greatest fixed point. This allows us to *define* x by a fixed-point equation.

We suppose that (1) $\mathcal{E}$ is ordered by a relation $\leq$; (2) f is monotone, that is, $x \leq y \Rightarrow f(x) \leq f(y)$; (3) all non-empty subsets A of $\mathcal{E}$ have a greatest lower bound $glb(A)$ (it is not necessary that $glb(A)$ is a member of A); and (4) $post_f = \{x \in \mathcal{E} \mid f(x) \leq x\}$ is non-empty (elements of $post_f$ are called post-fixed points of f). In our example we have $4 \in post_f$. Then f possesses a least fixed point which is $\mu_f = glb(post_f)$.

Indeed — let us remove the index f — as μ is a lower bound of post-fixed points, we have $\mu \leq x$ for all x such that $f(x) \leq x$, then, as f is monotone: $f(\mu) \leq f(x) \leq x$; then $f(\mu)$ is also a lower bound of $post$. As μ is greater than all lower bounds, we get $f(\mu) \leq \mu$. By monotony $f(f(\mu)) \leq f(\mu)$, hence $f(\mu) \in post$, then $\mu \leq f(\mu)$ since μ is a lower bound of $post$. By anti-symmetry of $\leq$ we have that $\mu = f(\mu)$.

Symmetrically, if all non-empty subsets A of $\mathcal{E}$ have a least upper bound $lub(A)$ and if the set $pre_f = \{x \in \mathcal{E} \mid x \leq f(x)\}$ of pre-fixed points of f is non-empty, then f possesses a greatest fixed point $\nu_f = lub(pre_f)$.

The least fixed point can also be reached from below when $\mathcal{E}$ possesses a least element $\perp$ (take $\mathcal{E} = [0, +\infty[$ in the previous example): we construct the monotonic sequence $(u)_\alpha$ with $u_0 = \perp$, $u_{\alpha+1} = f(u_\alpha)$ and $u_{\lim(\alpha_n)} = lub\{u(\alpha_n)\}$. The process ends at the first limit ordinal ω if f is continuous, i.e. $f(lub\{x_i\}) = lub\{f(x_i)\}$ for all monotonic sequences $(x_i)_{i \in \mathbb{N}}$. For the greatest fixed point, one would proceed symmetrically from a greatest element $\top$ in E.

The relation $\leq$ is not required to be total here. We can then apply the previous results with the inclusion relation on a set of sets, for example $\mathcal{E} = \mathcal{P}(E)$: $\varnothing$ plays the role of $\perp$, $glb(A)$ is the intersection of elements of A, $lub(A)$ is the union of elements of A and E plays the role of $\top$.

3.7 More About Computability

Here we give more precise definitions for the concepts of computability mentioned above [Gir87b, Bar90]. Here, unless we explicitly write *partial recursive function*, a *recursive function* will mean a *total recursive function*, according to the original definition of Gödel and Herbrand. Note that, following the work of Kleene, many textbooks use the opposite convention.

Let us consider a problem P. If we have a search process for solutions of P at our disposal which (i) succeeds if a solution exists, and (ii) answers "no" in the converse case, this process is called a **decision procedure.** If condition (i) only is satisfied, i.e. if the process may go on looking indefinitely for a solution where no solution exists, it is called a **semi-decision procedure.** To summarize what follows, a decision algorithm is a recursive function, while a semi-decision procedure is a partial recursive function.

For the remainder of this chapter, the functions considered are arithmetic functions. By that we mean functions that take natural integers as input and that return a natural integer. For the sake of uniformity, constants are considered to be functions of arity 0. In order to lighten the notation, applying a function f to n arguments $x_1 \dots x_n$ is denoted by $f(\vec{x})$, where $\vec{x}$ is seen as the n-tuple $\langle x_1, \dots, x_n \rangle$ — the value of n is the arity of f.

In order to formalize the concept of an algorithm, we need a formal language capable of expressing algorithms, and we have to stipulate the computations associated with legal expressions. This can be done with very low level constructs, but it is more convenient to use functions directly. It is easy to understand, for example, how to compute the composition of two functions provided one knows how to compute each of them separately. We proceed by introducing primitive recursive functions, then recursive functions and finally partial recursive functions, which correspond to progressively larger classes of algorithms.

It is important to keep in mind the distinction between the function which is computed, that is, a set of ordered pairs (the **extension** of the function), and the algorithm which performs the computation: two different algorithms may independently and correctly compute the same function f; for example one of them could be primitive recursive while the other is not. According to the following definition, f is then considered as primitive recursive. Indeed, the word *function* below takes its extensional meaning — though the underlying computation remains crucial in the rules (R_i) given below.

It may transpire that the most efficient algorithm that computes a given primitive recursive function is not primitive recursive. For instance, the obvious primitive recursive way for computing the minimum of two integers m and n is not symmetrical: it takes e.g. m steps, while a better algorithm would take $\min(m, n)$ steps. Indeed, a result due to Loïc Colson shows that the latter algorithm cannot be encoded using primitive recursion. Recursion theory is then an important theoretical tool, but the light shed on the concept of expressivity is limited.

3.7.1 Primitive Recursion

The **initial functions** are:

- the constant 0;
- the successor function $S(n) = n + 1$;
- the projections $\mathrm{pr}_i^n(x_1, \dots, x_n) = x_i, \quad 1 \le i \le n$.

We then consider the formation rules:

(R_1) **composition** rule: take $k+1$ functions h_1, ... h_k and g already constructed and construct the function f defined by $f(\vec{x}) = g(h_1(\vec{x}), \dots, h_k(\vec{x}))$;

(R$_2$) **primitive recursion** rule: take two functions g and h already constructed and construct the function f defined by

$$\begin{cases} f(\vec{x},0) = g(\vec{x}) \\ f(\vec{x},n+1) = h(\vec{x},n,f(\vec{x},n)) \ . \end{cases}$$

A **primitive recursive presentation** or, a **primitive recursive algorithm**, is an expression constructed only from initial functions and by application of rules (R$_1$) and (R$_2$). A **function** f is **primitive recursive** if there exists a primitive recursive presentation which computes f.

The occurrence of f on the right of "=" in (R$_2$) is not that problematic. Indeed it is clear that $f(\vec{x},0)$ is defined for all $\vec{x}$, then $f(\vec{x},1)$, and so on. The function f can be regarded as a sequence defined by induction but parameterized by $\vec{x}$: $f(\vec{x})_0, \dots f(\vec{x})_n, f(\vec{x})_{n+1}, \dots$

Examples. Addition is primitive recursive, as it can be defined by $\mathsf{add}(m,0) = m$ and $\mathsf{add}(m,n+1) = S(\mathsf{add}(m,n))$. Multiplication is defined in a similar way. We can then define the factorial function ($\mathtt{fact}(0) = 1$ and $\mathtt{fact}(n+1) = \mathtt{mult}(n+1, \mathtt{fact}(n))$) subtraction (see below), the exponential function, and many other functions over integers. The linear search of an integer n such that $P(n) = 0$, *is not* primitive recursive even if P is:

$$\begin{aligned} &R = R'(0) \\ &R'(n) = \mathtt{if}\ P(n) = 0\ \mathtt{then}\ n\ \mathtt{else}\ R'(n+1) \ . \end{aligned}$$

There is no way to define this function using only the previous rules. By contrast, there is a primitive recursive presentation of bounded linear search between p and q similar to the program given in § 2.4.4.

$$\begin{aligned} &R = R'(q-p) \\ &\begin{cases} R'(0) = q \\ R'(n+1) = h(n, R'(n)) \end{cases} \\ &h(n,r_2) = \mathtt{tzer}(q-(n+1),\ r_2,\ P(n+1)) \\ &\begin{cases} \mathtt{tzer}(r_1,r_2,0) = r_1 \\ \mathtt{tzer}(r_1,r_2,n+1) = r_2 \ . \end{cases} \end{aligned}$$

Note that testing the equality to zero, realized by tzer, makes use of rule (R$_2$), with $g = \mathrm{pr}_1^2$ and $h = \mathrm{pr}_2^4$.

In a programming language like Pascal, we get primitive recursive functions if we restrict iterative control structures to **for** loops (general **while** loops have to be prohibited[16]): in **for** loops, the number of iterations is computed (at run-time, however) *before* the loop. One of the main properties of primitive recursive functions is that they are total, in other words the corresponding programs terminate in all cases.

[16]goto statements and "recursive" (!) procedures must also be prohibited, as it is clear that such mechanisms are at least as powerful as the while loop.

A number of functions over natural integers, like subtraction, are usually not defined everywhere. As a consequence of the last remark, their primitive recursive presentation extends them over the whole set $\mathbb{N}$. The default value is often 0. Thus the usual primitive recursive definition of the predecessor function is $P(0) = 0$ and $P(n+1) = n$ — using (R_2) with $g = 0$ and $h(n, a) = n$, that is, $h = \mathrm{pr}_1^2$. We get subtraction by iteration of P.

There are total functions that cannot be defined by a primitive recursive presentation, but they are not that easy to find. One of the simplest is the Ackermann function:

$$\begin{cases} A(0,n) = n+1 \\ A(m+1,0) = A(m,1) \\ A(m+1,n+1) = A(m,A(m+1,n)) \ . \end{cases}$$

It can be shown that this function grows faster than all primitive recursive functions. Its termination can be proven by well-founded induction using a lexicographic ordering based on relation R_5 of § 3.5.2, with $E = F = \mathbb{N}$ and $R = S = R_1$.

3.7.2 Recursion, Decidability

The previous examples clearly show that primitive recursive functions do not exhaust intuitively computable functions. In order to enrich our set of functions, let us introduce the following rule:

(R_3) **minimalization** rule: take a function g already constructed such that

$$\forall \vec{x}\, \exists m\; g(\vec{x}, m) = 0 \tag{3.13}$$

and construct the function f that maps $\vec{x}$ to the smaller m such that $g(\vec{x}, m) = 0$, denoted by $f(\vec{x}) = \mu m[g(\vec{x}, m) = 0]$.

Intuitively, a way to compute this function is by a linear search program: successively try $m = 0$, $m = 1$, etc. until an m satisfying $g(\vec{x}, m) = 0$ is found.

A **recursive presentation**, also called an **algorithm**, is an expression constructed only from initial functions and by application of rules (R_1), (R_2) and (R_3). A **function** f is **recursive** if there exists a recursive presentation which computes f.

For example, the linear search program R given on page 60 is encoded by a trivial application of (R_3): by hypothesis there exists an n such that $P(n) = 0$, where P is primitive recursive; then we take simply $R = \mu m[P(m) = 0]$.

Again, recursive functions are total functions: requiring condition (3.13) amounts to ensuring *a priori* that the previous linear search program terminates. In other words, intuitively, an algorithm is a program which provides an answer for all input data. We then get a precise formal definition for the intuitive concept of an algorithm. This formal definition may be considered as arbitrary. However, as in physics, experience decides the matter.

Here we encode a predicate P by a function f_P from tuples of integers to $\{0,1\}$. A **predicate is recursive** if the corresponding function f_P is recursive. We can also define a **recursive set** E as a set (of integers, or of tuples of integers) having a recursive characteristic function. It means that we have at our disposal an algorithm for deciding, given any tuple $\vec{x}$, whether or not $P(\vec{x})$, or equivalently, whether or not $\vec{x}$ is a member of E. We say that a **problem** is **decidable** if the corresponding predicate is recursive. In the opposite case we say that the problem is **undecidable**.

3.7.3 Partial Recursion, Semi-Decidability

In practice and in logic as well, we need to consider programs which do not always terminate. Thus we are led to weaken the rule (R_3) by relaxing condition (3.13).

(R_3') **partial minimalization** rule: take a function g already constructed and construct the function f such that, if there exists an m such that $g(\vec{x}, m) = 0$, returns $f(\vec{x}) = \mu m[g(\vec{x}, m) = 0]$, or else is not defined.

The new rule (R_3') allows one to construct partial functions. Therefore, we now agree that our rules construct partial functions from partial functions.

A **partial recursive presentation** is an expression constructed only from initial functions and by application of rules (R_1), (R_2) and (R_3'). A **function** f is **partial recursive** if there exists a partial recursive presentation which computes f. Here is another definition: a partial recursive function is a function which can be encoded using a Turing machine.[17] The *Church thesis for partially computable functions* states that the class of partial recursive functions formalizes the intuitive concept of program.

Roughly, we can say that a recursive function is a partial recursive function whose termination is proven in all cases. Let us consider the linear search program given on page 60, where we add an integer parameter x in the search criterion P:

$$R(x) = R'(x, 0)$$
$$R'(x, n) = \texttt{if } P(x, n) = 0 \texttt{ then } n \texttt{ else } R'(x, n+1) \ .$$

In general, the search succeeds only for special values of x. For example, if we want to search the smaller n such that $2n = x$, we can choose for $P(x, n)$ the expression $(x - 2n) + (2n - x)$ (pay attention to the definition of subtraction!); then it is clear that the computation terminates for even values of x and for no others.

Here is another example of a partial recursive function, sometimes called the Syracuse function. It can only return 1, and in all known experiments

[17] We don't present a formal definition of Turing machines here: it is a bit long but raises no difficulty.

it does return. But termination for all inputs remains an open problem so far, thus we don't know if this function is recursive.

$$\begin{cases} U(0) = U(1) = 1 & \\ U(n) = U(\frac{n}{2}) & \text{if } n \text{ is even and } n > 1 \\ U(n) = U(3n+1) & \text{if } n \text{ is odd and } n > 1 \ . \end{cases}$$

What is the status of the function t_U that returns 1 if U is total and otherwise returns 0? This presentation of t_U is not recursive. However, the function k_i, which returns a fixed i, is recursive; then t_U is recursive as well, since t_U is either k_0, or k_1, though we don't know which one. We conclude that a computable function, as formally defined in recursion theory — a classical theory admitting the excluded middle principle, is not quite the same as a function we know how to compute.

The last important basic concept we present here is the concept of a recursively enumerable predicate or set. As suggested by the name, it is a set which can be completely covered by application of a calculable function on 0, 1, 2, etc. Equivalently, we can say that membership of this set is a **semi-decidable** problem.

We say that a **set is recursively enumerable** if it is the domain of a partial recursive function. We say that a **predicate** P **is recursively enumerable**:

- if it is the characteristic predicate of a recursively enumerable set;
- or, equivalently, if the function g defined by $g(x) = 0$ for all x such that $P(x)$, and undefined elsewhere, is partial recursive;
- or, equivalently, if there exists a recursive function f such that, for all y verifying $P(y)$, there exists x such that $y = f(x)$.

Let x be an integer and let P and Q be partial recursive predicates. The functions computing $P(x) \wedge Q(x)$, $P(x) \vee Q(x)$ and $\neg P(x)$ are partial recursive. If P and Q are recursive, these functions are recursive as well, which allows us to determine if $P(x)$, $Q(x)$, $P(x) \wedge Q(x)$, $P(x) \vee Q(x)$ and $\neg P(x)$ are true. If P and Q are only recursively enumerable, we are only able to determine if $P(x)$, $Q(x)$, $P(x) \wedge Q(x)$ and $P(x) \vee Q(x)$ are true. We have also the following theorem:

Theorem 3.1
A predicate (respectively, a set) is recursive if and only if itself and its negation (respectively, its complement) are recursively enumerable.

3.7.4 A Few Words on Logical Complexity

If the predicate P is recursively enumerable, then so are the predicates $P(x) \wedge Q(x)$, $P(x) \vee Q(x)$ $\forall x < n\ P(x)$, $\exists x < n\ P(x)$ and $\exists x\ P(x)$. However, $\neg P(x)$ and $\forall x\ P(x)$ are not always recursively enumerable.

The intuitive idea is that it is possible to encode the search for an x satisfying $P(x)$ by pseudo-simultaneously checking $P(0)$, $P(1)$, etc., but checking $\forall x\ P(x)$ would in general require an infinite number of verifications. As a consequence, a formula including unbounded quantifiers and (partial) recursive predicates specifies a relation between its free variables, but we may not have any algorithm for computing it. A relation thus specified is called **arithmetical.**

Kleene established that arithmetical relations can be classified according to the **arithmetical hierarchy**, which measures their logical complexity. Formulas are put under the form $\forall x_n \exists x_{n-1} \ldots \Delta$ or $\exists x_n \forall x_{n-1} \ldots \Delta$, where the predicate Δ is primitive recursive. Each class is characterized by the first quantifier and the number of quantifier *alternations*. Formulas of the first kind are designated by Π^0_n, formulas of the second kind by Σ^0_n. For example $\exists n\ n^2 = 25$ is Σ^0_1, while $\forall c \exists r\ r^2 \leq c \wedge c < (r+1)^2$ is Π^0_2. The reader can consult [vL90a, Cou91, Sho93] for a rigorous definition.

There is a tight link between complexity of program termination proofs, ordinals and logical complexity [Gir87b, CW97, Wai91, Wai93].

3.8 Notes and Suggestions for Further Reading

The reader interested in the sources of mathematical logic can find the texts of founding fathers edited and commented on by J. van Heijenoort in [vH67]. The *Handbook of Mathematical Logic* [Bar77] is a reference book for specialists. However, a number of chapters are very accessible, notably: the first, which is a good introduction to model theory; the chapter written by Shoenfield is a good introduction to set theory; and the chapter written by Rabin includes many decidability results.

The example of geometric figures comes from a contest organized by the US Air Force. Two teams, championing a functional language (Haskell, in fact), submitted similar solutions based on the principles[18] indicated in § 3.1.1. There are many introductory books on functional programming, for instance [Pau91], [BW88], [CMP02] and [CM98].

[18]They beat all other approaches hands down, which came as a surprise because traditionally favorite domains for functional languages were compilation or theorem provers.