Recursion Theory

Pieter J. W. Hofstra
Dept. of Mathematics and Statistics, University of Ottawa,
Ottawa, K1N 6N5, Ontario, Canada
email : phofstra@uottawa.ca

December 2007

Contents

Informal introduction

We start by asking a grand question: "What problems are solvable?". Of course, that is a vague question and we have to make precise what we take it to mean before we can start studying it. First, what kind of problems are we interested in here? Just certain mathematical problems, or other kinds as well? And what exactly do we mean by solvable?

A few examples

In order to get an idea of the kind of problems which could be of interest and what kind of solutions we may have in mind, we consider a few examples. (If you don't have the background for all of them, don't worry; the idea is to get the flavor across):

1. Given a natural number n, determine whether n is prime.
2. Given a computer program in some programming language (say Pascal, for sake of concreteness), determine whether the program will eventually halt on a given input or will loop forever.
3. Given a computer program, determine whether the program will eventually halt on *every* possible input.
4. Given a formula ϕ in the language propositional logic, find out whether ϕ is a tautology.
5. Given a formula ψ in predicate logic, find out whether ψ is provable using natural deduction. More generally, given a collection of axioms Γ (in a fixed first-order language) and a formula ϕ in that same language, determine whether $\Gamma \vdash \phi$. (For example, Γ could be Peano Arithmetic and ϕ could be the statement that there are infinitely many prime numbers.)
6. Given a polynomial $P(x_1, \ldots, x_n)$ in n variables with integer coefficients, determine whether the equation $P(x_1, \ldots, x_n) = 0$ has positive integer solutions.
7. Let G be a group, and suppose G is presented by a set of generators $g_1, \ldots, g_n$ and relations $r_1, \ldots, r_m$. Now let us suppose we are given a

```
while input>=0 do
begin
  write('Regardless of the input');
  write('I will be running forever');
end
```

Figure 1: A program which never halts

word w in these generators. Can we decide if the word reduces to the empty word?

8. Determine, given a natural number $n > 0$, whether there is a sequence of n consecutive 7's in the decimal expansion of π.

While we will not attempt to settle all these questions right now (by the end of the course we will have answered almost all of them), they do raise a number of interesting points, which will guide us later.

Particular versus general solutions First, for each of these problems, it is clear that there are *instances* which we can solve easily. For some polynomials it is easy to find roots. Also, there are predicate logical theorems for which it is not hard to construct a natural deduction proof We know that $\pi = 3.1415729\ldots$, so we have our answer to problem 8 for the case $n = 1$. And everyone who has a little bit of experience with programming knows what happens if we try to run the toy program shown if Figure 1.

But right here we're not just asking whether some (particularly easy) instances of our problems can be solved; we are asking whether there is a uniform way of solving all of them. So, while having a strategy to solve some particular cases is nice, we are after something bigger here, namely a *general solution* to the problem, which will work for all instances alike.

Decidable problems It should be clear that in problem 1: there is an algorithm which takes as input a number n and outputs "Yes" if the number is prime and "No" otherwise. This is a completely mechanical procedure, in the sense that one can program it on a computer.

The situation with problem 4 is similar: using the technique of truth tables, one can always settle the question whether a given proposition is tautologous; one says that propositional logic is *decidable*. Again, not only can we solve the problem for each given proposition ϕ, but we have a general recipe for solving the problem which works for each instance, and we could write a computer program to do this for us.

Thus, we may say that a problem is *decidable*, or *solvable*, if there exists an effective procedure which gives the correct outcome for each possible instance of the problem.

Undecidable problems By contrast, such a recipe can not exist for predicate logic. You could try to construct a natural deduction proof of a formula ψ, but if you don't succeed after several hours, you will not be sure whether that lack of success is because ϕ is really not provable or because natural deduction just isn't your strong point. Note that it is not just the case that we don't happen to know of a good algorithm: there is a theorem (Church's Theorem) which tells us that such an algorithm cannot exist!

Problem 7 about group representations is a classical problem in group theory: it is called the *word problem*. Again it is not hard to see that for some particular examples this may be doable, but it is a famous result that this problem is undecidable in general.

Problem 6 is another famous one which is usually called "Hilbert's tenth problem". It was shown in 1970 by Matijasevic that this is an unsolvable problem.

Unknown problems The example about the decimal expansion of π brings another aspect to the stage. Currently, it seems that the best we can do is the following: start generating the decimal expansion of π (there are known algorithms for doing so) and then see if, at some point in the process, the desired sequence of 7's appears. Up to this point, the situation seems analogous to that for predicate logic, but there is an important difference: in the case of the latter the existence of a decision procedure is impossible *in principle*, while in the case of π we simply don't have a procedure. It is entirely possible that more research on our favorite transcendental number will eventually lead to a simple procedure for settling this problem.

Semi-decidable problems There is another interesting observation to make: even though some of these problems are unsolvable, it may still be easy to decide whether a given answer to the problem is correct. For example, finding solutions to a polynomial equation may be too difficult, but checking whether a given sequence $n_1, \ldots, n_k$ is a solution is straightforward calculation. Also, you may not have been able to come up with a natural deduction proof for a certain formula ψ, but if you are given a proposed solution, then it is not hard to verify whether that solution is indeed correct!

This suggests the following strategy to approach example 5: start generating possible proofs in a systematic way, making sure that you eventually generate all of them, and for each of them, check if it is a valid proof of the formula ψ in question. Clearly, if a proof of ψ exists, then we will find it this way. But if there is no solution, we will never know and try more and more candidates in vain.... Problems for which this strategy works are sometimes called *semi-decidable*.

Really unsolvable Problems 2 and 3 seem closely related. The question whether a given program $\mathcal{P}$ halts on input x is semi-decidable; one simply runs the program with input x and waits for something to happen. But for

problem 3, where we are asking the question not for one particular value of x but for *all* values, this doesn't do: this question involves *infinitely many things to test*, and we simply don't have time for that.

The rough picture Thus, we see that we can classify these problems into three kinds: those which are solvable (1 and 4), those for which the "try and pray" strategy will at least eventually find a solution if it exists (6, 2, 5 and 7), and those for which even that is not possible (this is the case for problem 3 as we shall see later in the course). Note that problem 8 must belong to one of these three classes, but it is currently unknown to which it belongs!

A decision

While the above examples are instructive (and, indeed, the solutions to some of them are celebrated results), it is not clear yet how the general question "What problems are solvable?" can be attacked in a systematic manner. It seems as if we're forced to make a decision on which problems we are going to consider, otherwise we're doing philosophy and not science. At this point, we make a seemingly drastic decision: we are only going to consider problems of the form

Given a function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$, compute $f(x)$ for input x

There are several ways to justify this choice; for one thing, functions on the natural numbers (or, equivalently, real numbers) are well-defined and natural objects of mathematical study. There are also historical reasons for looking at these functions: one of the most important formal systems considered in the first half of the 20th century was *Peano Arithmetic*.

This brings us to the one-line summary of Recursion theory one may find at the beginning of textbooks on the subject:

Recursion theory is the study of functions over the natural numbers.

Does this mean that we will have nothing to say on problems which are not of this type? Not at all: it turns out that once we understand what it means for a function on the natural numbers to be computable, we can readily use this understanding to address other problems. This is done in roughly two ways: first, the concepts and techniques which we develop for the study of computable functions may sometimes be applied to other settings, working by analogy. Second, and more importantly, many other problems can often be "encoded" into the natural numbers in such a way that we can use our knowledge of the latter to reason about the problem in question. (The process of encoding a certain formalism into the natural numbers in an effective way is called *arithmetization*. This is an extremely useful and powerful technique, which will be exploited on several occasions.)

Thus, perhaps quite unexpectedly, our drastic decision to limit ourselves to problems about natural numbers is not so drastic after all and does not imply a severe loss of generality.

Effective procedures

We have set ourselves the new task of studying functions over the natural numbers from the point of view of their level of difficulty, but one important ingredient is missing: we still haven't said what we mean precisely by a "solution" to a problem. From the examples it has become clear that what we are after is some kind of "effective procedure", which takes as input a possible instance of the problem (suitably represented) and produces as output the correct answer. This procedure should be completely precise and deterministic, while at the same time should consist of simple steps. The procedure may not resort to "magical" tricks or things which "cannot done by hand"; this can perhaps be made more clear by considering a few examples of steps which should not be allowed:

- If there is no sequence of n consecutive 7's in the decimal expansion of π, add one to the input n.
- Pick an arbitrary prime number.
- Given input n, output 0 if $f(x) > n$ for all $x \in \mathbb{N}$ and 1 otherwise. (Here, f is a given function which we may assume to be effectively computable.)

The first step is not effective since it calls for a solution to a problem which we cannot effectively solve as of yet. The second step seems less harmful, but the problem lies in the word "arbitrary": the procedure should be *deterministic*, meaning that each step should have at most one unambiguous outcome. In the third example the problem is not that we invoke a call to the function f (which was assumed to be computable), but the fact that we have to test infinitely many things (we would have to compute $f(x)$ for each value of x and compare the outcome with n.

Rather, the kind of steps we have in mind are things like: adding 1 to a given number, given two numbers, discard the first and output the second, testing whether a given number is equal to zero, etc. In other words, we are looking for an explication of the notion of effective procedure in terms of elementary, mechanical computation steps, each of which involves the manipulation of finite data. The functions which can then be computed using such a procedure will be called *effectively computable*.

Two approaches...

There are two ways of approaching this problem: The first is mathematical in nature and follows a plan of attack which is quite common in mathematics: it starts with a few simple and well-understood objects (in our case, some very

simple functions which are intuitively computable) and gradually expands the horizon by building more complicated objects from these basic building blocks. One important way of constructing new functions from old is by means of *recursion*; it is because of this, that the functions obtained are called *recursive functions*. The fact that many interesting functions are defined using a form of recursion is responsible for the name of our subject. Thus, the mathematician's answer to the question "Which functions are effectively computable?" is: the recursive functions.

The second approach is more practical and may be regarded as belonging to computer science: find a theoretical framework for computation, and study what can be computed in that framework. The idea is that the framework models the actions of an idealized human computing agent, and that we systematically analyse the possibilities and limitations. There are several such frameworks, and Turing machines are perhaps the best-known among them. The purpose of such a framework is to study computation in an idealized setting (where there are no practical limitations of memory, time or space) which is sufficiently general but yet technically simply, so that its study will not become overly complicated.

In this course we will consider *register machines*; a register machine is an idealized computing device, tailor-made for operating on natural numbers, on which you can run programs. We then identify the execution of a program with a computation, and define a function to be computable precisely when there exists a register machine program which can compute it.

...which coincide

It turns out that for once, the mathematician and the computer scientist agree on the outcome: the recursive functions are exactly those which are computable by a register machine. The proof of this fact is not difficult, but it requires technique of *arithmetization*, the effective encoding of register machine programs into arithmetic.

One could object that our choice of looking at register machines was simply a fortunate one, and that one piece of corroborating evidence, while encouraging, does not mean we have proved that we have yet found the correct notion of effective procedure.

But it turns out that the coincidence of the recursive functions with the register machine-computable functions is not a fluke (nor a phenomenon manufactured by a biased choice of formalism): one can repeat the exercise with different formalisms for computation (we mention here Turing machine computations, flowchart computations, definability in the lambda-calculus, Post canonical systems) and the outcome remains the same. The fact that all these technically different (and intuitively quite nonsimilar) formalisms lead to the same notion of computable function is strong evidence that we have found the correct explication of the notion of effective procedure.

An even stronger statement is *Church's Thesis*, which identifies the intuitive (and hence non-mathematical) notion of "effectively computable" by "recursive". Philosophically, one may read this as implying that the fact that all the formalisms known so far give rise to the same notion of computability is a consequence of some higher principle, a law of some kind. Of course, such a statement is not strictly mathematical in nature, so can never be proved (or refuted) by mathematical means.

Elementary recursion theory...

The above results (which were developed by Church, Gödel, Kleene, Post and Turing in the 1930s and 1940s) form the foundation of the subject. Recursion theory takes off with the study of the class of recursive functions of recursive sets (which, using characteristic functions, may be viewed as functions). While one can ask a few basic questions about these classes, the theory derives most of its depth from the fact that the recursive functions can be *effectively enumerated*. By this we mean, that we can encode each of them as a natural number, in such a way that from that code we can effectively retrieve the function. (Again, this is an instance of arithmetization!) This means that in Recursion theory, the natural numbers appear in two guises: once as (codes for) functions, and once as arguments or values of functions. There are a few key results which are fundamental, namely the *Universality Theorem* and the *Parameter Theorem*. Without going into the details, we can say that these results describe the ways in which natural numbers *qua codes* interact with natural numbers *qua arguments*. This interaction lies at the basis of virtually all of the theory.

While the realm of solvable problems is a pleasant place to be, it is only natural that we soon begin to wonder about functions and sets which are *not* recursive. Are these outside the scope of Recursion theory? On the contrary: it is probably fair to say that most of the subject is concerned with problems which are unsolvable. Indeed, the solvable problems are from a certain point of view, not the most fascinating ones: they are all alike, so what more is there to say[1]?

One step more complicated than the recursive sets (those whose membership can be effectively decided) are the *recursively enumerable* sets (r.e. sets). A set is called recursively enumerable if there is an effective procedure for generating (enumerating) its elements, as in our problems 6, 2, 5 and 7 above. Thus if a given element belongs to an r.e. set we can run this procedure and it will eventually confirm that the element was in the set. But if the element was not in the set, the procedure will search forever and we will never know the answer. For this reason, r.e. sets are sometimes called semi-decidable, or semi-recursive sets.

[1]This is not completely fair, since even if we know that a given problem is solvable we may still wonder about what different kinds of solutions there exist, or how efficient these are. This is the subject matter of Complexity theory, an active branch of Recursion theory.

One can show that there are indeed sets which are not recursive but which are r.e.; the most famous example of such a set is called the *Halting set* this is the set of numbers n such that, when one applies the function coded by n to input n, the outcome is defined (remember, the recursive functions may not halt on every input). The proof that this set is not recursive is a classic example of a *diagonal argument*, which occurs in many places (the reader may already have seen examples of such arguments, for example in Cantor's proof that the continuum is not denumerable). Together with arithmetization, diagonalization is one of the important proof techniques in the subject.

...and beyond

But things do not end there. The r.e. sets turn out to be only the tip of the iceberg of unsolvable problems. In order to study those problems in a systematic way, various tools are available. First, there are various *hierarchies*, which aim at stratification of a certain collection of unsolvable problems. Not only do these provide insight into how various problems are related to each other, but they also point out an important connection to proof theory, by emphasizing the relation between computability and *definability* in certain logical systems. The arithmetical hierarchy for example, of which the lowest levels are occupied by the recursive sets and the r.e. sets, is based on definability in the system of Peano Arithmetic.

Reduction relations allow us to express that one given set is at least as complicated as another given set; thus, such a relation organizes sets with respect to their level of difficulty. One of the most important such relations is *Turing reducibility*; roughly, one says that a set A is Turing-reducible to a set B if one could compute membership of A *assuming that one had a procedure for determining membership of* B. (Sometimes this is called *oracle computability*: one can compute A with help of an oracle for B.) Under this relation, all recursive sets are of the same difficulty, but there are r.e. sets which are strictly more difficult (such as the Halting set). Identifying sets which have the same level of difficulty, we arrive at the branch of Recursion theory called *degree theory*, a rich and complicated subject, whose difficult problems and ingenious solutions dominated recursion theory in the 1950s. A crucial problem, known as *Post's problem*, asks whether there exist r.e. degrees between that of the recursive sets and that of the Halting set. The solution, achieved independently by Friedberg and Muchnik (1958), is still regarded as one of the milestones in the subject.

We briefly mention a number of other branches of recursion theory, some of which will be studied later in this syllabus. Many of these fall under the heading of so-called "generalized Recursion theory". By this, one typically understands an extension from the classical theory (about functions over and subsets of the natural numbers) to wider settings. In one direction, this leads quickly to deep connections with set theory, when one studies recursion theory on certain kinds of ordinals. In another direction, one aims at defining notions

of higher-type computation. For example, one can wonder what a computable functional from $\mathbb{N}^{\mathbb{N}}$ to $\mathbb{N}$ is. Thus, higher recursion theory, as this branch is usually called, stands to the classical theory in much the same way as functional analysis stands to ordinary analysis. Apart from its intrinsic appeal, higher recursion theory is attractive because it brings about an intimate connection between computability and *continuity*, since most of the objects of concern are naturally topologized.

Connections with other areas

The examples at the beginning of this introduction already indicated the relevance of recursion theory to other areas of mathematics and computer science. Indeed, the solutions to two of them (Hilbert tenth's problem and the word problem for groups) are famous applications of recursion theory to number theory and algebra, respectively. While classical recursion theory can be considered as being somewhere between number theory, and analysis, nowadays part of the subject arguably belongs to theoretical computer science, especially complexity theory.

Recursion theory also plays an essential role in understanding and making precise what, in a given area of mathematics such as algebra, is the *constructive content* of certain results. A famous example of this is Steinitz' Theorem, which says that any field possesses an algebraic closure (which is unique up to isomorphism). Unfortunately, the proof only tells us that such an algebraic closure exists, and it doesn't tell us how to find one! One would like to have a method (effective procedure!) for *constructing* this object of interest. It can be shown that, provided the original field is given to us in a suitable way (to be precise, when we have a *recursive presentation* of it) then such a construction exists.

Similar questions can be asked in many other situations, and such questions have been an important driving force behind the development of the subject. (Students of logic will certainly have recognized how this is closely related to questions about the intuitionistic validity of proofs.) *Recursive mathematics* aims at studying mathematics from the viewpoint of effective constructions. For example, in recursive analysis one looks at the recursive real numbers (those numbers whose decimal expansion can be generated by an effective procedure), and one investigates which results of classical analysis are true when interpreted in a constructive manner.

Want to know more?

We conclude this introduction by mentioning a few texts for further reading. Throughout the rest of the text, we will provide references to texts on specialized subjects which we cannot further explore. Here, we mention a few non-technical works which tell us more about the history and background of the subject, or which are informal treatments of the material which we will treat in

a formal fashion. A good example of the latter is Douglas Hofstädter's "Gödel, Escher, Bach": among many other things, it explains various approaches to effective computability and unsolvability, culminating in Gödel's incompleteness theorem.

For a thorough treatment of the history of the concepts of computable and recursive function, as well as Church's Thesis, there is Robert Soare's paper "Computability and recursion" (parts of it require familiarity with some of the theory).

Nowadays, Wikipedia is a great source of information. While the main article on recursion theory is perhaps not the most gentle introduction to the subject, there are various interesting articles on more specialized topics. For example, there is a list of interesting undecidable problems expanding our list of problems.

Finally, the book "Classical recursion theory" (Volume I) by Piergiorgio Odifreddi is not only highly recommended as a general reference for this course, being both very clear and encyclopaedic, but it also contains a lot of background and history, as well as more details on many of the issues considered here.

Chapter 1

Recursive functions

As explained in the introduction, the mathematical approach to defining what is meant by a computable function consists of starting with a few elementary functions, and then build new functions by allowing certain constructions on functions. The end result will be a class of functions on the natural numbers (of several variables), and the claim is that this class captures exactly those functions which are effectively computable.

In this chapter we carry out this program in a couple of steps. First, we introduce a smaller class of functions, called the *primitive recursive functions*. This class already contains quite a few functions which are used in everyday mathematics. We also look at primitive recursive *relations* and properties of those. Our main aim is to get familiar with this class by looking at a variety of examples and by studying some of its closure properties.

Next, we introduce two important ideas, namely arithmetization and diagonalization. These will allow us to construct examples of functions which are not primitive recursive. Those examples indicate that the class of primitive recursive functions is still "not large enough": the examples are functions which are effectively computable, but not PR.

Thus we allow one further construction, called *minimalization*. This further enlarges our class of functions, and the result will be called the class of *(partial) recursive functions*.

1.1 Primitive recursive functions

In this section we explain what primitive recursive functions are, and how they are constructed. We consider two ways of building new functions from old, namely generalized composition (substitution) and primitive recursion. The idea of inductive definitions and primitive recursion go back to Dedekind (1888), who was the first to succeed in giving a characterization of the well-order structure on the natural numbers. He also introduced the class of primitive recursive functions, later also defined by Skolem (1923) and Gödel (1931).

1.1.1 Basic functions

We begin by listing the basic functions.

Definition 1.1.1 (Basic functions)**.** The following functions are called *basic functions*:

- **Zero function:** $\mathbf{0}(x) = 0$ (The constant function $\mathbb{N} \longrightarrow \mathbb{N}$ with value 0)
- **Successor function:** $\mathbf{S}(x) = x + 1$
- **Projection functions:** $\mathbf{U}_k^n(x_1, \ldots, x_n) = x_k$ (where $n \geq k > 0$)

As a special case of a projection function, we may take $n = k = 1$: then we get the function $\mathbf{U}_1^1(x) = x$, the identity function. We will usually denote this identity function by $\mathbf{I}(x) = x$.

1.1.2 Generalized composition

We next look for ways of combining these functions into new functions. The first such method is *composition* (also called: *substitution*). Composition of two functions $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ of one variable is well-known: the composite function $f \circ g : \mathbb{N} \longrightarrow \mathbb{N}$ is defined by:

$$(f \circ g)(x) = f(g(x)).$$

Usually we omit the symbol $\circ$ and simply write fg. Since composition of functions is associative, we also omit parentheses in expressions with iterated composition, e.g. we write $fgh(x)$ for $f(g(h(x)))$.

However, we wish to deal with functions of possibly more than one variable; how should we compose those? This is called *generalized composition* and is defined in the following:

Definition 1.1.2 (Generalized composition)**.** Let $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ be a function of n variables ($n > 0$) and let $g_1, \ldots, g_n : \mathbb{N}^m \longrightarrow \mathbb{N}$ be an n-tuple of functions of m variables. Then the function $h : \mathbb{N}^m \longrightarrow \mathbb{N}$, defined by

$$\boxed{h(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))} \tag{1.1}$$

is the *generalized composite* of f and $g_1, \ldots, g_n$.

Usually, we write $h = f \circ \langle g_1, \ldots, g_n \rangle$ for the composite just defined, but sometimes we'll also use the notation $h = \mathbf{Comp}(f, g_1, \ldots, g_n)$.

Observe that each of the functions $g_1, \ldots, g_n$ must have the same number of arguments for this to work.

Example 1.1.3. Any constant function may be obtained from basic functions using composition. This can be proved by induction: the constant function $(x_1, \ldots, x_k) \mapsto 0$ is equal to the composite $\mathbf{0} \circ \mathbf{U}_1^k$. And if we assume that $(x_1, \ldots, x_k) \mapsto n$ is given, then composing with the successor function gives the function $(x_1, \ldots, x_k) \mapsto n + 1$.

As you may have noticed, there are often several ways in which you can show that a function is built from basic functions. For example, the function $f(x, y) = y + 3$ could be decomposed as $f = \mathbf{SSSU}_2^2$. This is probably the most straightforward way, but equally valid would be

$$f = \mathbf{S} \circ \mathbf{U}_2^2 \circ \langle \mathbf{S} \circ \mathbf{U}_2^2,\ \mathbf{SS} \circ \mathbf{U}_2^2 \rangle.$$

In order to check that this is also correct, we calculate:

$$\begin{aligned} \mathbf{S} \circ (\mathbf{U}_2^2 \circ \langle \mathbf{S} \circ \mathbf{U}_2^2,\ \mathbf{SS} \circ \mathbf{U}_2^2 \rangle)(x, y) &= \mathbf{SU}_2^2(\mathbf{SU}_2^2(x, y), \mathbf{SSU}_2^2(x, y)) \\ &= \mathbf{SU}_2^2(\mathbf{S}(y), \mathbf{SS}(y)) \\ &= \mathbf{SU}_2^2(y + 1, y + 2) \\ &= \mathbf{S}(y + 2) \\ &= y + 3. \end{aligned}$$

Note that both expressions $\mathbf{SSSU}_2^2$ and $\mathbf{S} \circ \mathbf{U}_2^2 \circ \langle \mathbf{S} \circ \mathbf{U}_2^2,\ \mathbf{SS} \circ \mathbf{U}_2^2 \rangle$ embody an *algorithm for computing* f!

1.1.3 Primitive recursion

Just composing basic functions doesn't give rise to the most spectacular of functions. The following construction however, does greatly expand our supply of computable functions. We first give a simple version which will get the main idea across, and then state the general version.

Definition 1.1.4 (Primitive recursion, simple version)**.** Let $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$ be a function of two variables, and let $c \in \mathbb{N}$ be a number. Define a new function $h : \mathbb{N} \longrightarrow \mathbb{N}$ as follows:

$$\boxed{h(0) = c \qquad h(n + 1) = g(n, h(n))} \tag{1.2}$$

Then h is said to be defined from g, c by *primitive recursion*.

Our notation is: $h = \mathbf{Pr}[g, c]$.

At first sight this is a suspicious definition of the function h; after all, we seem to be defining it in terms of itself. However, the idea is that in order to calculate $h(n)$ for a given value of n, we only need the previous values $h(0), \ldots, h(n - 1)$. Thus, we should read the definition as an algorithm for computing $h(n)$.

Example 1.1.5. Consider the function defined by $h(x) = 2x$. This function satisfies the equations

$$h(0) = 0 \qquad h(x+1) = h(x) + 2.$$

In order to get this into the form of 1.2 we let $g(m, n) = n + 2$, so that $g(x, h(x)) = h(x) + 2$. This function g can in turn be obtained from the basic functions using composition, say as $g = \mathbf{SSU}_2^2$, and therefore h can be obtained, via composition and primitive recursion, from the basic functions.

Again we stress that this expression of h actually embodies an algorithm for computing its values: for example, to compute $h(3)$, we first use $h(3) = h(2+1) = g(2, h(2)) = h(2) + 2$, according to the second clause. This means we have to compute $h(2) = h(1+1) = h(1)+2$; then we are forced to compute $h(1) = h(1+0) = h(0) + 2$, and finally $h(0) = 0$. We have computed all necessary values and can now work back to give the answer:

$$h(3) = h(2) + 2 = h(1) + 2 + 2 = h(0) + 2 + 2 + 2 = 0 + 2 + 2 + 2 = 6$$

as expected.

Here's another simple example: the predecessor function is defined by

$$\mathbf{p}(x) = \begin{cases} 0 & \text{if } x = 0 \\ x - 1 & \text{otherwise.} \end{cases}$$

We may reformulate this as

$$\mathbf{p}(0) = 0 \qquad \mathbf{p}(x+1) = x$$

which means that taking $g(x, y) = x$ gives $p(x + 1) = x = g(x, p(x))$, which matches equation 1.2. Thus the predecessor function is defined by primitive recursion from g, which is the projection function $\mathbf{U}_1^2$.

We now turn to the full-fledged version of primitive recursion. The difference with the simple version is that we allow parameters in the definition. In what follows, we abbreviate a sequence of variables $x_1, \ldots, x_k$ to $\mathbf{x}$.

Definition 1.1.6 (Primitive recursion, parametrized version)**.** Let $k \geq 0$, and let $g : \mathbb{N}^{k+2} \longrightarrow \mathbb{N}$ and $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ be given functions. Define a new function $h : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ as follows:

$$\boxed{h(\mathbf{x}, 0) = f(\mathbf{x}) \qquad h(\mathbf{x}, n+1) = g(\mathbf{x}, n, h(\mathbf{x}, n))} \tag{1.3}$$

Then h is said to be defined from g, f by *primitive recursion*.

Our notation is: $h = \mathbf{Pr}[g, f]$.

First observe that the simple version is indeed an instance of this version, when we let $k = 0$. Second, in the above definition we call the variables

$x_1, \ldots, x_k$ *parameters*; and the last variable is the *recursion variable*. Also, a pair of equations of the above form is usually referred to as *recursion equations* (for h).

We will soon see many functions obtainable in this fashion. But first we make an important definition which collects all those functions into one class.

Definition 1.1.7 (Class of primitive recursive functions)**.** The class $\mathcal{PR}$ is the least class of functions with the following properties:

(i) The basic functions $\mathbf{0}, \mathbf{S}, \mathbf{U}_i^n$ are in $\mathcal{PR}$

(ii) If $f, g_1, \ldots, g_n$ are all in $\mathcal{PR}$, then so is $f\langle g_1, \ldots, g_n\rangle$, provided this is well-defined. (The class $\mathcal{PR}$ is closed under generalized composition.)

(iii) If $f : \mathbb{N}^k \longrightarrow \mathbb{N}, g : \mathbb{N}^{k+2} \longrightarrow \mathbb{N}$ are in $\mathcal{PR}$ then so is $\mathbf{Pr}[g, f]$. (The class $\mathcal{PR}$ is closed under primitive recursion.)

The functions in $\mathcal{PR}$ are called *primitive recursive* functions. The above definition may be abbreviated to the following:

> $\mathcal{PR}$ is the least set which contains the basic functions and is closed under composition and primitive recursion.

The inductive character of $\mathcal{PR}$ means that in order to prove that a given function is primitive recursive, we have to show that it is either a basic function, or that it can be obtained via composition or primitive recursion from functions *which we have already shown to be primitive recursive*. Here are a number of examples of primitive recursive (PR) functions, together with a demonstration of why they are PR. Usually, this involves two things: rewriting the given function in such a way that it matches the definition of primitive recursion (as in equation 1.3), and then showing that the constituents are themselves primitive recursive. In the first two examples we provide details, and leave the other examples as an exercise.

Examples 1.1.8.

(i) Addition $h(x, y) = x + y$. We show that this function can be obtained by primitive recursion from simpler functions. First considering the equations

$$x + 0 = x \qquad x + (y + 1) = (x + y) + 1.$$

This suggests we take $f(x) = x$, and $g(x, y, z) = z + 1$. Then

$$x + 0 = f(x) \qquad x + (y + 1) = g(x, y, x + y)$$

as required, and $h = \mathbf{Pr}[g, f]$ Since f is a basic function and g can be obtained as a composite of the successor and a projection, both are PR and thus h is PR.

(ii) Multiplication $h(x,y) = x \cdot y$ can be obtained by considering first

$$x \cdot 0 = 0 \qquad x \cdot (y+1) = (x \cdot y) + x.$$

Thus we take $f(x) = 0$ and $g(x,y,z) = z + x$, so that

$$x \cdot 0 = f(x) \qquad x \cdot (y+1) = g(x,y,x \cdot y).$$

Now f is a basic function, and g may be written as the composite $g(x,y,z) = \mathbf{U}_3^3(x,y,z) + \mathbf{U}_1^3(x,y,z)$. Since addition was already shown to be PR, this shows that multiplication is also PR.

(iii) Exponentiation x^y

(iv) Factorial $x!$ (with the understanding that $0! = 1$)

(v) Generalized sum $x_1 + \cdots + x_k$

(vi) Generalized product $x_1 \cdots x_k$

(vii) Every polynomial function $P(x_1, \ldots, x_n)$ (with coefficients in $\mathbb{N}$) is primitive recursive

(viii) $zero : \mathbb{N} \longrightarrow \mathbb{N}$, defined by

$$zero(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

(ix) $nonzero : \mathbb{N} \longrightarrow \mathbb{N}$,

$$nonzero(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{otherwise} \end{cases}$$

(x) Minimum $min(x,y)$ (or more generally $min\{x_0, \ldots, x_k\}$)

(xi) Maximum $max(x,y)$ (or more generally $max\{x_0, \ldots, x_k\}$)

(xii) Cutoff subtraction

$$x \ominus y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise.} \end{cases}$$

We end this section with two remarks: remember that one of our aims is to give a mathematical definition of what we mean by "effectively computable function". We have defined the PR functions, so we should check now whether these indeed match our intuitions about being effective. First, the basic functions: the zero function corresponds to the procedure of always returning the value 0 regardless of the input; the successor function to the procedure of adding one to the input, and the projections to the procedure of discarding

all but one of the inputs. All of these procedures are in accordance with our pre-theoretical conceptions about effectivity: indeed if these are not effective procedures, then what would we have left?

Now if we have two functions which are computable via certain effective procedures, then so is their composite: one applies the first procedure to the input, and then applies the second procedure to the outcome. Thus, effective procedures are closed under composition. The same holds for primitive recursion, which not only defines a new function from two given ones, but explicitly gives us a recipe for effectively computing this new function in terms of the procedures of the old functions. Thus, we may conclude that every PR function is effectively computable.

Our second remark is a continuation of the previous one. Not only is every PR function effectively computable, but if we are given a proof that a given function f is PR, then that proof contains a derivation of f from the basic functions. But such a derivation may be viewed as a description of a procedure for computing the values of the function! Hence:

> A proof that a function is PR contains an algorithm for computing the function.

The philosophically minded reader may wish to reflect upon how this relates to what in the introduction was called the "constructive content" of mathematics.

1.2 Closure properties

We have introduced the primitive recursive functions, but as witnessed by some of the examples it may be a bit of work to show that a given function belongs to this class. In order to facilitate work we now discuss a few closure properties, i.e. constructions which, when applied to PR functions, always give PR functions. We start by considering PR relations and subsets; not only are these of independent interest, but they will also allow for better insight in the properties of $\mathcal{PR}$.

1.2.1 Primitive recursive relations

So far we have been concerned with functions on the natural numbers. Equally interesting are subsets of the set of natural numbers. Just as we can think of a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ as presenting us with the problem "compute $f(n)$ for given n", a subset $A \subseteq \mathbb{N}$ corresponds to the problem "given x, determine if $x \in A$". For example, the set A could be the set of all prime numbers. Then the problem is, to determine whether a given number x is prime or not.

Now a subset $A \subseteq \mathbb{N}$ corresponds to a function $\chi_A : \mathbb{N} \longrightarrow \mathbb{N}$, via

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise.} \end{cases}$$

The function χ_A is called the *characteristic function* of A. It has the property that it takes values in $\{0, 1\}$; a function with this property is called a *predicate on* $\mathbb{N}$.

Conversely, if we are given a predicate P on $\mathbb{N}$ then we may define the set $\{x \in \mathbb{N} | P(x) = 1\}$, the set of those elements for which the predicate is true. It is easily seen that these two constructions are inverse. All of the above is equally valid when we replace subsets of $\mathbb{N}$ by subsets of $\mathbb{N}^k$; these are usually called k-ary relations, and may be identified with $\{0, 1\}$-valued functions on $\mathbb{N}^k$, called k-ary predicates.

Because of the fact that every subset corresponds to a function, we may now apply the concepts we defined earlier for functions to subsets:

Definition 1.2.1 (Primitive recursive set)**.** A relation $A \subseteq \mathbb{N}^k$ is *primitive recursive* if its characteristic function $\chi_A : \mathbb{N}^k \longrightarrow \mathbb{N}$ is.

We will also speak of primitive recursive predicates, which are simply PR functions which happen to be predicates. Often, when $P(x)$ is a predicate, we will simply write $P(x)$ for $P(x) = 1$.

Example 1.2.2.

- The predicate *zero*, defined by $zero(x) = 1$ if and only if $x = 0$, (see example 1.1.8) is PR.
- The set $\{(n, m) | n \leq m\}$ is PR. One way of proving this is by noting that $n \leq m$ if and only if $n \ominus m = 0$, which, by the previous example, is equivalent to $zero(n \ominus m)$. From example 1.1.8 we know that cutoff subtraction is PR, so it follows that this predicate is PR.

1.2.2 Boolean combinations

As known from elementary logic we may combine predicates using Boolean connectives $\wedge$ (conjunction), $\neg$ (negation) and $\vee$ (disjunction) in order to form new predicates. On the level of subsets/relations, these correspond to intersection, complementation and union, respectively.

Lemma 1.2.3. *Let $A, B \subseteq \mathbb{N}^k$ be primitive recursive relations. Then the sets $\mathbb{N} - A, A \cap B$ and $A \cup B$ are also primitive recursive. Moreover, the sets $\emptyset$ and $\mathbb{N}^k$ are both PR.*

Proof. If A is PR, then by definition its characteristic function χ_A is PR. The characteristic function of $\mathbb{N} - A$ is the function

$$\chi_{\mathbb{N}-A}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \notin A \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $\chi_{\mathbb{N}-A}(\mathbf{x}) = zero(\chi_A(\mathbf{x}))$, where $zero$ is the function from example 1.2.2.
The other cases are left as exercise. □

We summarize the above results by saying that the primitive recursive sets and relations are *closed under Boolean operations*. We give a simple application, which uses the earlier proved fact that $n \leq m$ is a PR predicate.

Example 1.2.4. The relations

(i) $\{(n, m)|n = m\}$

(ii) $\{(n, m)|n \neq m\}$

(iii) $\{(n, m)|n < m\}$

are PR. Indeed, $n = m$ iff $n \leq m \wedge m \leq n$, and thus it is a Boolean combination of PR predicates. The other two are similar.

When we apply the Boolean operations to sets we use the notation $A \cap B, A \cup B$ and $\mathbb{N} - A$; when we apply them to predicates we use $P \wedge Q, P \vee Q$ and $\neg P$.

Definition by cases One useful feature of PR predicates is that we can use them to define new PR functions by cases. Let $f, g : \mathbb{N}^k \longrightarrow \mathbb{N}$ be functions, and let P be a predicate on $\mathbb{N}^k$. Define the function h as follows:

$$\boxed{h(\mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{if } P(\mathbf{x}) \\ g(\mathbf{x}) & \text{otherwise.} \end{cases}} \tag{1.4}$$

We say that h is obtained from f, g and P via *definition by cases*.

Lemma 1.2.5. *If f, g and P are primitive recursive, and h is obtained via definition by cases, then h is also primitive recursive.*

Proof. The function h may be expressed as

$$h(\mathbf{x}) = (P(\mathbf{x}) \cdot f(\mathbf{x})) + (\neg P(\mathbf{x}) \cdot g(\mathbf{x})).$$

Thus it is a composite of PR functions. □

One can generalize this to an arbitrary finite number of cases: if we have functions $f_1, \ldots, f_k$ (of the same arity) and predicates $P_1, \ldots, P_k$ with the property that for each $\mathbf{x}$ there is exactly one i such that $P_i(x)$ holds, then the function defined by $h(\mathbf{x}) = f_i(\mathbf{x})$ if $P_i(\mathbf{x})$ is PR if all of the f_j, P_j are. The proof is left as an exercise.

1.2.3 Bounded quantification

Now let $P(\mathbf{x}, y)$ be a predicate on $\mathbb{N}^{k+1}$ and let $n \in \mathbb{N}$ be a fixed natural number. We define two new relations:

$$\exists y \leq n.P(\mathbf{x}, y) \qquad \forall y \leq n.P(\mathbf{x}, y)$$

The first, $\exists y \leq n.P(\mathbf{x}, y)$ is true precisely when there exists a number $y \leq n$ for which $P(\mathbf{x}, y)$ is true, and $\forall y \leq n.P(\mathbf{x}, y)$ is true precisely when every $y \leq n$ satisfies $P(\mathbf{x}, y)$. The constructors $\exists y \leq n$ and $\forall y \leq n$ are called *bounded quantifiers* (existential and universal, respectively).

Lemma 1.2.6. *If $P(\mathbf{x}, y)$ is primitive recursive, then so are $\exists y \leq n.P(\mathbf{x}, y)$ and $\forall y \leq n.P(\mathbf{x}, y)$.*

Proof. We may rewrite $\exists y \leq n.P(\mathbf{x}, y)$ as a disjunction:

$$\exists y \leq n.P(\mathbf{x}, y) \Leftrightarrow P(\mathbf{x}, 0) \vee \ldots \vee P(\mathbf{x}, n).$$

Each of the $P(\mathbf{x}, i)$ is a PR predicate on $\mathbb{N}^k$ (check!). The proof for the bounded universal quantifier is obtained by replacing disjunction by conjunction. □

More general (and useful) is the following construction, where we replace the constant n by a parameter: from $P(\mathbf{x}, y)$ define the relations

$$\boxed{A = \{(\mathbf{x}, z) | \exists y < z.P(\mathbf{x}, y)\} \qquad B = \{(\mathbf{x}, z) | \forall y < z.P(\mathbf{x}, y)\}.} \tag{1.5}$$

In the above, we say that A is defined from P using *bounded existential quantification* and that B is defined from P using *bounded universal quantification.*

Lemma 1.2.7. *The PR predicates are closed under bounded quantification, i.e. if P is PR then so are A, B as defined above.*

Proof. We observe that A satisfies

$$(\mathbf{x}, 0) \notin A, \qquad (\mathbf{x}, z+1) \in A \Leftrightarrow (\exists y < z.P(\mathbf{x}, z)) \vee P(\mathbf{x}, z+1).$$

We leave it to the reader to verify that this defines A by primitive recursion in terms of P. The case of the universal quantifier is also left as an exercise. □

Bounded quantification allows for a lot of quick definitions of PR functions.

Example 1.2.8. The following functions and relations are primitive recursive.

(i) $\{(n, m) | m \text{ is a multiple of } n\}$

(ii) $n|m$, the characteristic function of $\{(n, m) | n \text{ divides } m\}$ (where by convention $0|0$).

(iii) $\{n|n \text{ is } prime\}$

Proof. For the first, observe that m is a multiple of n iff $\exists z \leq m.nz = m$. For the second, we have $n|m$ iff m is a multiple of n. For the third, observe that n is prime iff $n > 1$ and $\forall x < n.\neg x|n$. □

We end with two constructions for functions. Let $f(\mathbf{x}, y)$ be a PR function, and define

$$g(\mathbf{x}, z) = \sum_{y<z} f(\mathbf{x}, y) \qquad \text{and } h(\mathbf{x}, z) = \prod_{y<z} f(\mathbf{x}, y).$$

These operations are called *bounded sum* and *bounded product*, respectively. In the exercises you will be asked to show that if f is PR, then so are g, h.

Our last construction is called *bounded minimalization*. Given $f(\mathbf{x}, y)$ as before, define

$$\boxed{k(\mathbf{x}, z) = \mu y \leq z.f(\mathbf{x}, y) = 0} \tag{1.6}$$

This is the function which gives the least $y \leq z$ satisfying $f(\mathbf{x}, y) = 0$, and 0 if such y doesn't exist; we say that k is defined from f by bounded minimalization. Again it can be shown that if f is PR, so is k.

As an application, we show that the function $f(n) = p_n$, where $p_0 = 0$, and p_n is the n-th prime number (i.e. $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, etc.) is PR. Indeed, we may define by induction $p_0 = 0$, $p_{n+1} = \mu z \leq (n!) + 1.p_n < z\&\text{Prime}(z)$. (This uses the number-theoretic fact that there always exists a prime number between p_n and $n! + 1$. We only use that here in order to get an upper bound for our search.)

We summarize the results in this section by saying that the class $\mathcal{PR}$ is closed under bounded sums, bounded products, bounded minimalization and definitions by cases. The PR predicates are closed under Boolean operations and bounded quantifiers.

1.3 Diagonalization and more

So far, we have obtained a nice supply of primitive recursive functions and predicates. Indeed, many everyday mathematical functions are PR. In this section we give two examples of functions which are not PR, but still are effectively computable. Both examples will give us occasion to gently introduce some important theoretical ideas which will be used on many occasions later in the course.

The first example is a function called the Ackermann function, after its inventor. It is defined by a recursion scheme which does not fit the scheme of primitive recursion; it is defined using so-called *double recursion*. Functions defined in that way can grow very fast, and we will show that the Ackermann function does so, too: it grows faster than any primitive recursive function, and hence cannot be primitive recursive! However, there is a perfectly good algorithm for computing its values.

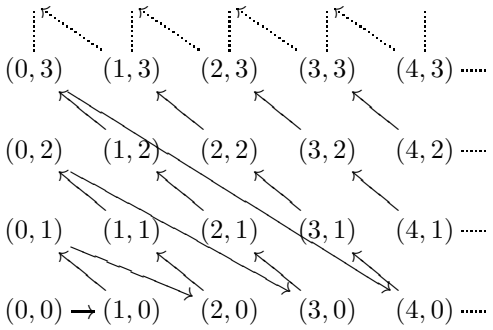


Figure 1.1: A coding of the plane

The second example of a function which is not PR has a distinctly different flavor, and is based on two important techniques, namely *arithmetization* and *diagonalization*. The idea is to enumerate the primitive recursive functions in a suitable way (this is the arithmetization part) and then to build a new function which must be different from all the functions in the enumeration using a diagonal trick.

We will first develop one technical ingredient, namely that of *effective coding* of the plane and of finite sequences and sets.

1.3.1 Codings

From basic set theory we know that the set $\mathbb{N}$ is in bijective correspondence with $\mathbb{N} \times \mathbb{N}$. Since we are interested in primitive recursive functions, it makes sense to ask if there exists a primitive recursive bijection $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$. This turns out to be the case; the idea is to enumerate the plane $\mathbb{N} \times \mathbb{N}$ in an effective manner. One such way is the following: start at the origin, and follow directions from there, as given by figure 1.1.

Thus, we are systematically enumerating the antidiagonals $y + x = c$. It is clear that this sets up a bijection from $p : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ (p for pairing). It is also plausible that this bijection is effectively computable, because the picture gives us clear and simple instructions to compute it: to compute the value at (n, m), start at the origin, and count the number of steps until you hit the point (n, m). For example, we find that $p(1, 3) = 13$.

There are also two decoding functions $p_0, p_1 : \mathbb{N} \longrightarrow \mathbb{N}$, which have the following properties:

$$p_0(p(n, m)) = n, \qquad p_1(p(n, m)) = m, \qquad p(p_0(x), p_1(x)) = x. \tag{1.7}$$

Thus, the tuple $\langle p_0, p_1 \rangle$ is the inverse to p. The equations in 1.7 are often called the *pairing equations*.

The algorithm for computing p_0 is as follows: given input x, start walking on the diagram for x steps. See at which point (n, m) you are, and output n. The algorithm for p_1 is of course similar.

We now have to show that the functions p, p_0, p_1 are actually primitive recursive. Without further ado we claim that the function p actually satisfies the equation

$$p(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y.$$

The interested reader may try to derive this. At any rate, p is a second degree polynomial, and therefore clearly PR.

In addition, it is easily shown (exercise) that the coding $\mathbb{N}^2 \longrightarrow \mathbb{N}$ satisfies $n \leq [n, m], m \leq [n, m]$ for all n, m. This allows us to make the following definitions for the unpairings:

$$p_0(x) = \mu y \leq x.(\exists z \leq x.p(y, z) = x), \qquad p_1(x) = \mu z \leq x.(\exists y \leq x.p(y, z) = x).$$

Notation We will often write $[n, m]$ instead of $p(n, m)$ in order to improve readability of large expressions. We will also write $(n)_0, (n)_1$ for $p_0(n), p_1(n)$.

Sequences Now that we have found a coding for pairs, we may extend this to sequences of arbitrary length, by defining inductively

$$\sharp\langle x\rangle = x, \qquad \sharp\langle x_0, x_1, \ldots, x_k\rangle = [x_0, \sharp\langle x_1, \ldots, x_k\rangle].$$

For each fixed k, this defines a primitive recursive bijection $\sharp : \mathbb{N}^k \longrightarrow \mathbb{N}$. This may be shown by induction on k: for $k = 1$, we get the identity on $\mathbb{N}$, and for the inductive step we use the fact that pairing is primitive recursive

Now we may define decodings in terms of those for the pairing. Our notation is as follows: if $y = \sharp\langle x_0, \ldots, x_n\rangle$ then $(y)_i = x_i$. The functions $(y)_i$ are defined by induction on i (where $n > 0$ is fixed)

$$(y)_0 = p_0(y), \qquad (y)_{i+1} = p_1((p_1(y))_i).$$

This coding of sequences has one drawback: given a number n, we cannot tell what sequence it codes, *unless we are told how long the sequence was*! In many applications, a more refined coding is needed, and in the exercises you will look at an example of a better behaved coding.

Finite sets We now consider a suitable coding of the finite powerset $\mathcal{P}_f(\mathbb{N})$ of the natural numbers. To each finite subset of $\mathbb{N}$ we will assign a suitable code, based on binary representation.

Let $A \subseteq \mathbb{N}$ be a finite set; write $A = \{a_0, \ldots, a_k\}$. We encode A as follows:

$$\sharp(A) = 2^{a_0} + \cdots + 2^{a_k}.$$

In case $A = \emptyset$, we set $\sharp(A) = 0$. For example, the set $\{0, 3, 4\}$ has code $2^0 + 2^3 + 2^4 = 1 + 8 + 16 = 25$.

The following notation is standard: if y is a number, then the finite set coded by y is denoted D_y.

The procedure for finding the elements of D_y from y is easy: write y as a sum of powers of 2; if in this representation there is a term 2^n, then $n \in D_y$.

Example 1.3.1. What is D_{75}? We write

$$75 = 1 + 2 + 8 + 64 = 2^0 + 2^1 + 2^3 + 2^6.$$

Thus, $D_{75} = \{0, 1, 3, 6\}$.

The above idea can be turned into a primitive recursive predicate:

Lemma 1.3.2. *The predicate*

$$n \in D_y \Leftrightarrow n \text{ is an element of the set coded by } y$$

is PR.

Proof. Indeed, we know that $n \in D_y$ iff $2^n | y$, and the latter is PR. □

We may also generalize the result from section 1.2.3 that the PR predicates are closed under bounded quantification to the following:

Lemma 1.3.3. *Let $P(\mathbf{x}, y)$ be a PR predicate. Then the predicates*

$$\exists y \in D_z.P(\mathbf{x}, y) \qquad \forall y \in D_z.P(\mathbf{x}, y)$$

are both PR.

Proof. From the definition of the coding of finite sets it is clear that $y \in D_z$ implies $y < z$. Thus we have $\exists y \in D_z.P(\mathbf{x}, y)$ iff $\exists y < z.(y \in D_z \wedge P(\mathbf{x}, y))$, and similarly for universal quantification. □

1.3.2 Arithmetization

We have seen that for every PR function there exists an expression in terms of basic functions, composition and primitive recursion. This has an important consequence: the collection $\mathcal{PR}$ is countable! We are going to take advantage of that by constructing a suitable *enumeration* of the primitive recursive functions, i.e. a sequence $f_0, f_1, \ldots$ of PR functions such that every PR function occurs somewhere in this sequence (possibly more than once).

Let us first think a bit more about why $\mathcal{PR}$ is countable. There are countable many basic functions (zero, successor, and for each $i \leq n$ a projection

function). Now look at the following sequence of sets:

$$\begin{array}{rcl}
\mathcal{PR}_0 & = & \text{the class of basic functions} \\
\mathcal{PR}_1 & = & \mathcal{PR}_0 \cup \{h | h = \mathbf{Comp}[f, g_1, \ldots, g_n]\} \text{ for some } f, g_i \in \mathcal{PR}_0 \\
\mathcal{PR}_2 & = & \mathcal{PR}_1 \cup \{h | h = \mathbf{Pr}[f, g]\} \text{ for some } f, g \in \mathcal{PR}_1 \\
\vdots & \vdots & \vdots \\
\mathcal{PR}_{2n+1} & = & \mathcal{PR}_{2n} \cup \{h | h = \mathbf{Comp}[f, g_1, \ldots, g_n]\} \text{ for some } f, g_i \in \mathcal{PR}_{2n} \\
\mathcal{PR}_{2n+2} & = & \mathcal{PR}_{2n+1} \cup \{h | h = \mathbf{Pr}[f, g]\} \text{ for some } f, g \in \mathcal{PR}_{2n+1} \\
\vdots & \vdots & \vdots
\end{array}$$

Now each of the classes $\mathcal{PR}_n$ is countable; since

$$\mathcal{PR} = \bigcup_{n \in \mathbb{N}} \mathcal{PR}_n.$$

the class $\mathcal{PR}$ is a countable union of countable sets, hence countable.

The idea is now to assign a natural number to each definition of a PR function in the following way:

- the code of the zero function is $[0]$
- the code of the successor function is $[1]$
- the code of the projection $\mathbf{U}_i^n$ is $[2, n, i]$

This gives a code to all the basic functions. The idea is that the first number gives the information that we are dealing with a basic function (level 0). Note that if you're given a natural number x, then you can effectively determine whether x is a code of a basic function and if so, of which.

- Let codes $p, q_1, \ldots, q_k$ of PR functions $f, g_1, \ldots, g_k$, respectively be given. Assume that $g_1, \ldots, g_k$ all have arity n, and that f has arity k. Then the composite $\mathbf{Comp}[f, g_1, \ldots, g_k]$ gets code $[3, p, q_1, \ldots, q_k]$.
- Given codes p, q of PR functions f, g, where $f : \mathbb{N}^k \longrightarrow \mathbb{N}, g : \mathbb{N}^{k+2} \longrightarrow \mathbb{N}$, $\mathbf{Pr}[g, f]$ gets code $[4, p, q]$.

At this point, the reader should convince him/herself of three things: first, that this is well-defined, and that every derivation of a PR function gives a code; second, that given a number c, one can effectively test whether that number is a code of something; and third, that if c codes some derivation of some function, then one can effectively unravel which derivation that is (and hence also which function).

We may now define the following sequence of unary functions $f_0, f_1, \ldots$:

$$f_n = \begin{cases} \text{the function with code } n & \text{if } n \text{ codes a unary function} \\ \text{the function } f(x) = 0 & \text{otherwise.} \end{cases}$$

Every unary PR function is of the form f_i for some i; conversely, if we are given a number n then we can effectively unravel it as to obtain a derivation of the function f_n. We have obtained an *effective enumeration of the unary PR functions*. We have restricted our attention to the unary functions here because we will use those in the next section, but one could equally well define an enumeration of the PR functions of arbitrary arity.

Suppose we are now given a number n, which we view as the code of a unary PR function f_n. If we are also given an input x, we may compute $f_n(x)$. Note that we can only do this because our coding is effective, and we can reconstruct f_n from n. Thus we have an effectively computable function of two variables

$$Eval(n, x) = f_n(x). \tag{1.8}$$

Theorem 1.3.4. *There exists an effectively computable function (namely the function Eval defined above) which is not primitive recursive.*

Proof. Assume that $Eval$ is PR. Then the function $g(x) = Eval(x, x) + 1$ is PR as well. Since g is a unary PR function, it must occur somewhere in the enumeration $f_0, f_1, \ldots$ of all unary PR functions, say $g = f_m$. Thus for all $x \in \mathbb{N}$, we have $g(x) = f_m(x)$. In particular for $x = m$ we get:

$$\begin{aligned} f_m(m) &= g(m) \\ &= Eval(m, m) + 1 \qquad \text{by definition of } g \\ &= f_m(m) + 1 \qquad \text{by definition of } Eval \end{aligned}$$

Clearly this is a contradiction, and hence our assumption that $Eval$ is PR must be false. □

1.3.3 Diagonalization*

The proof of theorem 1.3.4 is a typical *diagonal argument*; in this section we look at this kind of argument a bit more abstractly.

Let A be a set. Recall that a *fixed point* of a function $f : A \longrightarrow A$ is an element $a \in A$ such that $f(a) = a$. We suppose now that we have a function $f : A \longrightarrow A$ which is *fixed point free*, i.e. which has no fixed points. For example, the successor function on $\mathbb{N}$ is fixed point free.

We consider an infinite matrix $(a_{i,j})_{i,j\in\mathbb{N}}$ of elements from A:

$$\begin{matrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix}$$

We focus on the *diagonal* $(a_{n,n})$ of this matrix. To each of the elements we may apply our fixed point free f, and we obtain a sequence of elements

$$b_n = f(a_{n,n}).$$

We now ask whether this sequence $(b_n)_{n\in\mathbb{N}}$ can coincide with one of the rows of the matrix $(a_{i,j})$. Could it be the first row, i.e. $(b_n) = (a_{0,n})$? No, because $b_0 = f(a_{0,0}) \neq a_{0,0}$, so they differ on the first element. Could it be the second? No, because they differ on the second element. In general, because of

$$b_n = f(a_{n,n}) \neq a_{n,n}$$

we cannot have $(b_n) = (a_{m,n})$ for any m. Thus we have created, by diagonalization, a new sequence of elements not occurring as a row of the matrix.

Example 1.3.5 (Cantor's theorem)**.** For this example, take $A = \{0, \ldots, 9\}$, and let $f(x) = (x + 1) \mod 10$. We wish to show that the unit interval $(0, 1)$ is not countable. Every real number $a \in (0, 1)$ has a decimal expansion, which we view as a sequence of elements of A.

Suppose there would exist an enumeration of $(0, 1)$, say $a_0, a_1, \ldots$. Since every a_i is a sequence of elements of A, we write a_i as the sequence $a_{i,0}, a_{i,1}, \ldots$. Thus we get a matrix $(a_{i,j})$ as above, where the element $a_{i,j}$ is the j-the digit in the decimal expansion of the i-th real number in the enumeration.

Now apply the diagonal trick: it gives a new sequence (b_n). Now (b_n) is the decimal expansion of a number in $(0, 1)$, and hence should occur somewhere in the enumeration. But (b_n) is different from all the rows of the matrix and hence is not in the enumeration. Contradiction.

We leave it as a useful exercise to show that the diagonal argument we applied in the previous section to the function *Eval* fits into this framework.

1.3.4 Ackermann's function*

Our second example of a function which is not PR but which is still computable is the so-called Ackermann function. While the *Eval* function hinged on a coding trick, the Ackermann function is based on a recursion scheme which is wider than the scheme for primitive recursion.

The primitive recursion scheme 1.3 was based on the idea of defining a function in terms of its previous values. The actual recursion took place on one variable. In the definition of the following function we use a *double recursion*:

Definition 1.3.6 (Ackermann)**.** Define a function $A : \mathbb{N}^2 \longrightarrow \mathbb{N}$ by double recursion:

$$\begin{aligned} A(0, x) &= x + 1 \\ A(n + 1, 0) &= A(n, 1) \\ A(n + 1, x + 1) &= A(n, A(n + 1, x)) \end{aligned}$$

To get an idea for how this works let's compute $A(1,1)$. By the third clause, $A(1,1) = A(0, A(1,0))$, so we have to compute $A(1,0)$ first. By clause 2, $A(1,0) = A(0,1)$, which by clause 1 equals 2. Thus $A(1,1) = A(0,2) = 3$, again by clause 1.

The function A grows very rapidly: just try a few more values. However, it can easily be shown (exercise) that if we fix the first argument n, then the function $A_n = A(n,-)$ is still PR.

We will now prove that the function A grows faster than any primitive recursive function. First a bit of terminology:

Definition 1.3.7. Let $m : \mathbb{N} \longrightarrow \mathbb{N}$ be a function.

(i) We say that a function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ is *bounded by* m if $f(x_1, \ldots, x_k) < m(max\{x_1, \ldots x_k\})$ for all $x_1, \ldots, x_k$. Notation: $f \in \mathcal{B}(m)$

(ii) For $g : \mathbb{N} \longrightarrow \mathbb{N}$, we also say that m *dominates* g if there is an x_0 such that $g(x) < m(x)$ for all $x > x_0$.

We will outline a proof that the function $A'(x) = A(x,x)$ dominates every primitive recursive function. First, we claim that the following hold:

1. $n + x < A_n(x)$
2. $A_n(x) < A_n(x+1)$
3. $A_{n+1}(x) < A_n(x)$.

These properties are shown by double induction on n and x (exercise). We conclude that for each x, $A(x,x) < A(x+1, x+1)$.

From the above it follows that if we can show that if a PR function $g : \mathbb{N} \longrightarrow \mathbb{N}$ is bounded by some A_n, then $A'(x)$ dominates f. Indeed, suppose A_n is a bound for f. Then we have $f(x) < A_n(x)$ for all x. But then for $x > n$, we also have $A'(x) = A(x,x) > A_n(x)$, so that A' dominates f.

Thus, we have to show that for each primitive recursive f there is an A_n which is a bound for f. We claim, again leaving the proof as an exercise:

1. $\mathbf{0}, \mathbf{S}, \mathbf{U}_i^n \in \mathcal{B}(A_1)$
2. $A_n(A_n(x)) \leq A_{n+1}(x)$
3. If $h = \mathbf{Comp}[f, g_1, \ldots, g_k]$ and $f, g_1, \ldots, g_k \in \mathcal{B}(A_n)$ for some n, then $h \in \mathcal{B}(A_{n+1})$.
4. If $h = \mathbf{Pr}[f, g]$ and $f, g \in \mathcal{B}(A_n)$ for some n, then $h \in \mathcal{B}(A_{n+1})$.

Thus the basic functions are all bounded by A_1, and whenever we construct a new function using composition or primitive recursion then this preserves the property of being bounded.

We conclude that the function A' dominates every PR function and hence cannot be PR itself, and we have shown:

Proposition 1.3.8. *The Ackermann function is not primitive recursive.*

The recursion scheme which we used to define the Ackermann function is an instance of *double recursion*. We now give a more general format (without parameters):

Definition 1.3.9 (Double recursion scheme)**.** Let functions f, g, k, l, m (of appropriate arities) be given. Define

$$\begin{aligned} h(0,x) &= f(x) \\ h(n+1,0) &= g(x, h(n, l(n))) \\ h(n+1,x+1) &= k(n, x, h(n, x+1), h(n, m(n, x, h(n+1, x)))) \end{aligned}$$

Thus, in order to compute $h(n+1, x+1)$, one needs to know not only the value of $h(n+1, x)$ but also of $h(n, y)$ for various y. The above definition still gives us an algorithm for computing h in terms of the auxiliary functions f, g, k, l, m, but as the example of the Ackermann function showed, this cannot necessarily be formulated in terms of primitive recursion. A few more aspects of recursion schemes will be explored in the exercises.

The two examples of non-PR functions given in this section seem quite different, but upon further investigation, this is not quite so: it turns out that the function *Eval* can also be defined using double recursion!

We conclude this section with a few remarks. First, we have now seen two recursion schemes, namely primitive recursion and double recursion. One can easily imagine other schemes which define functions using recursion, and this leads to the general question of what kind of valid recursion schemes exist and what kind of functions they give rise to. In the exercises some other schemes are presented, and once we have proved the recursion theorem in Chapter 3 we can address this question more accurately.

1.4 General recursive functions

The examples given in the previous section show that there is something missing in the class $\mathcal{PR}$, and we now try to overcome the gap. By the way, there is, apart from these examples, a more banal reason why $\mathcal{PR}$ cannot comprise all functions which are effectively computable: some effective procedures give rise to *partial* functions. For example, the procedure defined by "Take input x; add one to x; if $x = 0$ then output x, otherwise repeat" is a perfectly good example of an effective procedure. However, once we start it, we never come to a halt (cf. the program in Figure 1 in the introduction, page vi).

But $\mathcal{PR}$ contains only total functions, and hence we need to add more functions, in particular partial functions.

In this section, we solve all those problems at once, by adding (on top of the composition and primitive recursion schemes already present) the scheme of *unbounded minimalization*, which captures the idea of a search operation.

This will lead us to the class of *partial recursive functions*, as defined by Kleene in 1936.

1.4.1 The μ-operator

In section 1.2.3, we saw that the class $\mathcal{PR}$ was closed under bounded minimalization: we could search the natural numbers from 0 up to a specified upper bound, and ask for the least number with a given property. The operation we define here is similar, only we leave out the upper bound, so that we can search arbitrarily far into the natural numbers.

Definition 1.4.1 (Unbounded minimalization)**.** Let $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ be a function (possibly partial). Define the function $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ by

$$\boxed{h(\mathbf{x}) = \begin{cases} \text{the least } y \text{ s.t.} & f(\mathbf{x}, y) = 0 \text{ and } f(\mathbf{x}, y')\downarrow \text{ for all } y' < y \\ \text{undefined} & \text{otherwise.} \end{cases}} \tag{1.9}$$

Notation: $h(\mathbf{x}) = \mu y.(f(\mathbf{x}, y) = 0)$.

Then the function $h(\mathbf{x}) = \mu y.(f(\mathbf{x}, y) = 0)$ is said to be defined from f by *unbounded minimalization*. To see how this works in practice, let us look at a few examples.

Examples 1.4.2.

1. Let $f(x, y)$ be a polynomial in x, y. Then $\mu y.(f(x, y) = 0)$ is the function which, on input x, gives the least y such that (x, y) is a root of f provided such root exists; otherwise, the function is undefined.

2. Let $g(x, y) = 1$, the constant function. Then $\mu y.(g(x, y) = 0)$ is the completely undefined function.

We may also consider minimalization over a predicate. For a predicate P, we write $\mu y.P(\mathbf{x}, y)$; this is the function which searches for the least y such that $P(\mathbf{x}, y)$ is true, and undefined if no such y exists. (Note that there is a bit of tension between the convention that $P(x) = 1$ means that $P(x)$ is true, and the definition of the minimalization operator which tests for 0.) There are a few remarks to be made. First, the μ-operator can be thought of as an ordered search strategy; if we have a way of computing f, then the strategy for computing $\mu y.f(x, y) = 0$ is to start by setting y to 0. Then compute $f(x, y)$ and test if $f(x, y) = 0$; if so, output y, if not, increase y by one and repeat. This procedure is effective, but it doesn't necessarily terminate.

On a more technical level, we note that the side condition that f is defined on all previous values $f(x, y'), y' < y$ cannot be omitted. For suppose f would be a function for which $f(0, 0)$ is undefined, and $f(0, 1) = 0$. Then the above strategy for computing $\mu y.f(0, y) = 0$ would not work, because we would get lost in the computation of $f(0, 0)$, and never find the answer $y = 1$. See the exercises for some more information on this phenomenon.

1.4.2 Recursive functions

The key definition of this chapter is the following:

Definition 1.4.3 ((Partial) recursive functions)**.** A function is called *(partial) recursive* (also: *general recursive*, or simply *recursive*) if it can be obtained from the basic functions using finitely many applications of composition, primitive recursion and unbounded minimalization.

This definition may need some clarification. When we were looking at PR functions, we only applied the schemes of generalized composition and of primitive recursion to total functions. Now we are considering non-total functions as well, and we have to think what application of such a scheme to partial functions results in. The solution is easy: we simply read the defining equations for a function *as equations of partial functions*. For example, if $f, g : \mathbb{N} \longrightarrow \mathbb{N}$ are both partial functions, then the composition $g \circ f$ is the partial function defined by $(g \circ f)(x) = g(f(x))$. Thus the domain of definition of $(g \circ f)$ is the set $\{x | f(x)\downarrow, g(f(x))\downarrow\}$. Also, if h is defined by primitive recursion from partial functions f, g via $h(x, 0) = f(x), h(x, n + 1) = g(x, n, h(x, n))$ then the domain of h may be described inductively by

$$\{(x, 0) | f(x)\downarrow\} \cup \{(x, n + 1) | h(x, n)\downarrow, g(x, n, h(x, n))\downarrow\}.$$

With this understanding we may now say that the operations of generalized composition and primitive recursion are really operations on partial functions, which, when applied to total functions, result in total functions.

The class of partial recursive functions will be denoted by $\mathcal{REC}$. Thus we have:

> $\mathcal{REC}$ is the least set which contains the basic functions and is closed under composition, primitive recursion and unbounded minimalization

Note that according to our terminology, a recursive function may or may not be total. If f happens to be total and recursive then we say that f is *total recursive*; the class of total recursive functions will be denoted $\mathcal{REC}_{tot}$. If we wish to emphasize that a recursive function is not total, then we will speak of a *strictly partial* recursive function.

Of course, we may also define recursive sets and relations:

Definition 1.4.4 (Recursive set, recursive relation)**.** Let $A \subseteq \mathbb{N}^k$, $k > 0$ be a relation. Then A is defined to be *recursive* if its characteristic function χ_A is (necessarily total) recursive. If $k = 1$ we speak of a *recursive subset* of $\mathbb{N}$.

The class of all recursive subsets of $\mathbb{N}$ is denoted by $\mathfrak{Rec}$. One also encounters the terminology *decidable*.

Before we look at recursive functions and relations more closely, we again observe that every recursive function must be effectively computable: the only new operation of minimalization, when applied to an effectively computable function, gives a new effectively computable function. Moreover, to show that a given function is recursive means that we derive it from the basic functions using the given operations. Thus:

> A proof that a given function is recursive contains an algorithm for computing the function.

1.4.3 First properties

The most interesting properties of the class $\mathcal{REC}$ will be studied in Chapter 3. Here, we gather a number of more pedestrian results, which are nevertheless quite useful.

Most of these results generalize their counterparts for primitive recursive functions, and are formulated in terms of closure properties. The proofs are identical to those for PR functions, so we just state the results here without proof.

Proposition 1.4.5 (Closure properties of $\mathcal{REC}$)**.** *The class of partial recursive functions is closed under*

(i) *bounded sums and products*

(ii) *bounded minimalization*

(iii) *definition by cases.*

Proposition 1.4.6 (Closure properties of recursive relations)**.** *The class of recursive relations is closed under*

(i) *Boolean combinations*

(ii) *bounded quantification.*

Again, this should be understood in terms of partial functions, e.g. if we define $h(x,z) = \sum_{y \leq z} f(x,y)$ and f is partial, then $h(x,z)$ is defined precisely when all of $f(x,0), \ldots, f(x,z)$ are defined.

The recursive sets are *not* closed under unbounded quantification! Indeed, if we have a recursive relation $R(x,y)$, then consider the set $\{x | \exists y.R(x,y)\}$ (usually simply written $\exists y.R(x,y)$). To see whether $\exists y.R(x,y)$ holds for given x, we have to search for a y such that $R(x,y)$ holds. Now the function

$$f(x) = \mu y.R(x,y) \tag{1.10}$$

is recursive, since it is defined by minimalization over a recursive predicate. However, if there is no y with $R(x, y)$ then $f(x)$ will be undefined. Later we will prove that for suitable R, the set $\exists y.R(x, y)$ is indeed non-recursive.

It is still worth observing that if $R(x, y)$ is a recursive predicate, then the function

$$g(x) = \begin{cases} h(x) & \text{if } \exists y.R(x, y) \\ \text{undefined} & \text{otherwise} \end{cases}$$

is partial recursive when h is. Indeed, it is the composite $U_2^1 \circ (h, f)$ where f is as in equation 1.10. In particular, taking $h(z) = 1$, we obtain the partial recursive function

$$g(x) = \begin{cases} 1 & \text{if } \exists y.R(x, y) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

This function is sometimes called the *semi-characteristic function* of the set $\exists y.R(x, y)$, which in turn is called *semi-decidable*. (Compare with the "try and pray"-strategy discussed in the introduction.)

Finally, we state a simple fact relating the recursiveness of a function and its graph.

Lemma 1.4.7. *Let $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ be a total function, and denote its graph by*

$$Gr(f) = \{(\mathbf{x}, y) | f(\mathbf{x}) = y\} \subseteq \mathbb{N}^k \times \mathbb{N}.$$

Then f is recursive if and only if $Gr(f)$ is a recursive set.

Proof. Exercise for the reader. □

This result breaks down when f is non-total. We will revisit this topic in Chapter 3.

1.4.4 The Ackermann function revisited*

We conclude our discussion of recursive functions in this Chapter by showing that the Ackermann function is total recursive. By lemma 1.4.7 we may show that the graph of the function is recursive. The idea for the proof is based on the algorithm for computing the values of the function in terms of previous values. Since a computation uses only *finitely many* such previous values, we may collect them in a finite sequence, and code that sequence as a single number.

Let us say that a *computation set* for the Ackermann function A is a finite set $S \subseteq Gr(A)$ satisfying

- if $(0, x, v) \in S$ then $v = x + 1$
- if $(n + 1, 0, v) \in S$ then $(n, 1, v) \in S$
- if $(n + 1, x + 1, v) \in S$ then $(n + 1, x, w) \in S$ for some w.

We now have:

Lemma 1.4.8. *If S is a computation set for A and $(n, x, v) \in S$, then $A(n, x) = v$. Moreover, if $A(x, n) = v$ then there is a computation set S containing (n, x, v).*

These facts are left as exercises to the reader.

Now suppose $S = \{s_1, \ldots, s_k\}$ is a computation set. Each s_i is a triple (n_i, x_i, v_i), which we may code as a single number as $[n_i, x_i, v_i]$. Then we code S as the finite set with elements $[n_i, x_i, v_i]$.

Now the predicate

$$CS(y) \Leftrightarrow y \text{ codes a computation set}$$

is primitive recursive, because it can be expressed as follows:

$$\begin{aligned} CS(y) \quad \Leftrightarrow \quad & \forall z \in D_y.((z)_0 = 0 \longrightarrow (z)_2 = (z)_1 + 1) \wedge \\ & \forall z \in D_y.((z)_0 > 0 \wedge (z)_1 = 0 \longrightarrow [(z)_0 - 1, 1, (z)_2] \in D_y) \wedge \\ & \forall z \in D_y.((z)_0 > 0 \wedge (z)_1 > 1 \longrightarrow \exists w < z.([(z)_0, (z)_1 - 1, w] \in D_y)) \end{aligned}$$

Since we have

$$A(n, x) = z \Leftrightarrow \exists y.(CS(y) \wedge (n, x, z) \in D_y)$$

and since the function $\mu y.(CS(y) \wedge (n, x, z) \in F_y)$ is total, the graph of A is recursive and we have proved:

Proposition 1.4.9. *The Ackermann function is total recursive.*

1.5 Exercises

Recursive functions

Exercise 1.1 (*)**.** Let $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ be a primitive recursive function.

1. Show that $g(x) = f(x, x)$ is also PR.
2. Same for $h(x, y, z) = f(x, y)$
3. Same for $k(x, y) = f(y, x)$.

Exercise 1.2 (**)**.** Generalize the previous exercise to the following: let $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ be PR, and suppose that $\sigma : \{0, \ldots, k\} \longrightarrow \{0, \ldots, k\}$ is a bijection.

1. Show that the function defined by

$$(x_0, \ldots, x_k) \mapsto f(x_{\sigma(0)}, \ldots, x_{\sigma(k)})$$

   is also PR. Note that this generalizes the third part of the previous exercise.

2. How would you generalize the other parts of the previous exercise?

Exercise 1.3 (**)**.** Complete the proofs for example 1.1.8.

Exercise 1.4 (**)**.** Show that the function

$$x \mapsto x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \qquad (x \text{ times})$$

is PR.

Exercise 1.5 (**)**.** Show that any finite set is primitive recursive.

Exercise 1.6 (*)**.** A predicate $R(x, y)$ is called *regular* if for each x there exists y such that $R(x, y)$ holds. Show that if $R(x, y)$ is regular, then the function $f(x) = \mu y.R(x, y)$ is total recursive. Is the converse true?

Exercise 1.7 (*)**.** Let f be a total function. Show that f is recursive if and only if its graph is recursive.

Exercise 1.8 (*)**.** Complete the proof for lemma 1.2.3.

Codings of pairs, sequences and finite sets

Exercise 1.9 (** Another way of coding pairs)**.** Consider the function $j(x, y) = 2^x(2y + 1) - 1$.

1. Show that this function is a primitive recursive bijection $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$.

2. Give an explicit formula for the unpairing operations and explain why they are PR.

Exercise 1.10 (** Another way of coding sequences)**.** In this exercise we study a better coding of finite sequences. In what follows, we write p_n for the n-th prime number. Recall that the assignment $n \mapsto p_n$ is PR. Given a sequence $\langle x_1, \ldots, x_n \rangle$, let the code of this sequence be

$$\sharp\langle x_1, \ldots, x_n \rangle = p_1^{x_1+1} \cdot \cdots \cdot p_n^{x_n+1}.$$

In the case of the empty sequence, this means $\sharp\langle\rangle = 0 = p_0$

1. Show that there are primitive recursive decoding functions $(x)_i$, where $1 \leq i \leq n$ such that $(\sharp\langle x_1, \ldots, x_n \rangle)_i = x_i$.

2. Show that there is a primitive recursive function $lh : \mathbb{N} \longrightarrow \mathbb{N}$, such that $lh(x)$ =length of the sequence coded by x.

Exercise 1.11 (** Operations on sequences)**.** Prove that the following operations on codes of sequences are PR (here, k, l are fixed):

1. $(\sharp\langle x_1, \ldots, x_k \rangle, y) \mapsto \sharp\langle x_1, \ldots, x_k, y \rangle$

2. $(\sharp\langle x_1, \ldots, x_k \rangle, \sharp\langle y_1, \ldots, y_l \rangle) \mapsto \sharp\langle x_1, \ldots, x_k, y_1, \ldots, y_l \rangle$

3. $\sharp\langle x_1, \ldots, x_k \rangle \mapsto \sharp\langle x_{\tau(1)}, \ldots, x_{\tau(k)} \rangle$ where τ is any function $\{1, \ldots, k\} \longrightarrow \{1, \ldots, k\}$.

Exercise 1.12 (** Operations on finite sets)**.** Using the coding of finite sets discussed in section 1.3.1, show that the Boolean operations $\cup, \cap$ are PR. Explicitly this means that the functions

$$(x, y) \mapsto \sharp(D_x \cup D_y) \qquad \text{and} \qquad (x, y) \mapsto \sharp(D_x \cap D_y)$$

are PR.

Exercise 1.13 (**)**.** We have seen two ways of representing a finite set A, namely via its characteristic function χ_A (which is always PR, see exercise 1.5), and as a number n such that $D_n = A$. Show that, given n, one can effectively find a code for χ_A. Can one also effectively pass back?

Exercise 1.14 (** Coding of PR functions)**.** This exercise refers to the coding of definitions of PR functions described in section 1.3.2.

Show that the function which takes as input two codes n, m and outputs a code for the composite function $f_m \circ f_n$ is PR.

Recursion schemes

Exercise 1.15 (*)**.** Why does the following scheme not define a total function?

$$\begin{aligned} h(0,0) &= 0 \\ h(n+1,x) &= h(n,x+1) \\ h(n,x+1) &= h(n+1,x)+1 \end{aligned}$$

Are there partial function satisfying the above scheme? If so, which?

Exercise 1.16 (*** Course-of-values recursion)**.** Consider the following variation on the primitive recursion scheme:

$$h(\mathbf{x},0) = f(\mathbf{x}) \qquad h(\mathbf{x},n+1) = g(\mathbf{x},n,\sharp\langle h(\mathbf{x},0),\ldots,h(\mathbf{x},n)\rangle) \tag{1.11}$$

This scheme is called *course-of-values recursion*. Prove that if f, g are PR, then so is h.

Exercise 1.17 (***)**.** Prove that the function $Eval$ (defined in (1.8) on page 16) can be defined from PR functions using double recursion. Hint: define $Eval(n,x)$ using a case distinction on what type of function is coded by n.

Longer problems

Exercise 1.18 (**** Decidability of propositional logic)**.** The syntax of propositional logic is inductively given by:

- Propositional variables: $\phi_0, \phi_1, \phi_2, \ldots$ are propositions
- Connectives: if ψ, χ are propositions, then so are $\neg\psi, \psi\wedge\chi, \psi\vee\chi, \psi \longrightarrow \chi$.

The set of propositions is denoted $PROP$.

1. Devise a suitable encoding of $PROP$ into the natural numbers; that is, construct an injective function $\flat : PROP \longrightarrow \mathbb{N}$.
2. Define a recursive predicate P which checks whether a number is the code of a proposition or not. (Note: if you have defined your coding very smoothly, every number codes a proposition and you can skip this step.)
3. Writing ψ_n for the proposition with code n, show that the function

   $$(n,m) \mapsto \flat(\psi_n \wedge \psi_m)$$

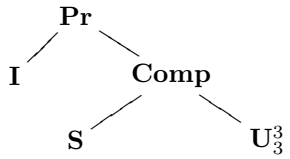
   is recursive. Same for the other connectives.
4. Define a recursive function $H : \mathbb{N} \longrightarrow \mathbb{N}$ such that $H(x) = 0$ if x doesn't code a proposition and $H(x) = \sharp\langle i_1,\ldots,i_n\rangle + 1$ if x codes a proposition which contains exactly the atomic propositions $\phi_{i_1},\ldots,\phi_{i_n}$.

5. Give a definition of a recursive predicate $M(x, v)$ which encodes "x is a code of a proposition with atomic formulas $\phi_{i_1}, \ldots, \phi_{i_n}$, and $v = \sharp\langle v_1, \ldots, v_n\rangle$ is a sequence of $0, 1$s". This means that v codes a valuation for x.

6. Define a recursive function $V : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ such that $V(x, v)$ is the truth value of the proposition coded by x under the valuation coded by v.

7. Show that the predicate $T(x)$, defined by "the proposition coded by x is a tautology" is decidable.

Exercise 1.19 (**** Derivations of PR functions)**.** In this exercise we represent PR functions as certain trees. The idea is that the tree encodes the derivation of the function from the basic functions. For example, we can write the addition function as $\mathbf{Pr}[\mathbf{Comp}[\mathbf{S}, \mathbf{U}^3_3], \mathbf{I}]$, and we represent this by the tree



1. Give a formal definition of derivation trees for PR functions. (Be careful about arities.)

2. Explain why every derivation tree uniquely determines a PR function.

3. Explain that every PR function has at least one derivation tree.

4. Explore relations amongst derivations trees such that related trees represent the same function.

Chapter 2

Machine models

We have introduced the notion of partial recursive function on the natural numbers and have argued that all partial recursive functions are effectively computable, in the intuitive sense that for each recursive functions there is an algorithm computing its values. We now turn to the question whether our class is large enough, or whether we have perhaps left out functions which should be considered computable.

To address this question we look at a machine model for computation, which purports to capture the actions of an idealized computing agent. This model, called a register machine, can perform computations on natural numbers, and thus gives rise to the class of all functions computable by register machine programs.

Our main goal in this chapter is to prove that the recursive functions are exactly those computable by register machines. It is a bit technical, but not difficult to show one direction of this result, namely that every recursive function can be computed by a register machine; however, the converse requires the technique of arithmetization, which we already briefly met in the previous chapter.

The main technical construction is the effective coding of register machine programs into the natural numbers. We set up this coding and then show how the execution of a program may be simulated on the level of codes. This development culminates in the definition of Kleene's T-predicate and the Normal Form Theorem, which, among other things, tells that one can compute values of a recursive function by performing a suitable search over the T-predicate.

2.1 Register machines

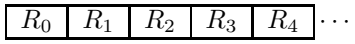
A register machine is both a simplified and an idealized version of an ordinary computer. We start by giving a formal definition of such a machine and of a register machine program. Next, we define what a computation of a register machine program is and what it means for a function to be computable. Finally, we look at some simple examples of programs.

Instruction	Meaning	Alternative notation
$\mathsf{Inc}(n)$	Increase value of R_n by 1	$R_n \leftarrow R_n + 1$
$\mathsf{Dec}(n)$	Decrease value of R_n by 1 if $R_n \neq 0$, do nothing otherwise	$R_n \leftarrow R_n - 1$
$\mathsf{Jump}(n, m)$	If $R_n = 0$, jump to instruction m; if $R_n \neq 0$, do nothing	`if` $R_n = 0$ `goto` m

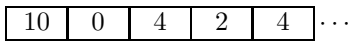
Figure 2.1: Register machine instructions

2.1.1 Programs

Let us imagine an infinite row of *registers* which serve as storage units for natural numbers. We picture the registers as



Each of the registers R_i contains a natural number, called its *value*; an assignment of values to each of the registers specifies a *configuration* of the machine. configuration We picture states by simply writing the values of the registers in the corresponding boxes, as in



(2.1)

thus, in the above example, the register R_0 has value 10, register R_3 has value 2, etc. We will simply write $R_0 = 10, R_3 = 2$, and so on. Typically, we will denote configurations by C, D, where these are functions $\mathbb{N} \longrightarrow \mathbb{N}$. Thus, $C(6) = 12$ means that in configuration C, we have $R_6 = 12$.

Viewing the registers as the (infinite) working memory of an idealized computer, we now discuss programs for this machine. A register machine program will be a finite list of instructions, which we introduce first.

There are three possible instructions: one for increasing the value of a register, one for decreasing it, and a conditional jump operation. Figure 2.1.1 summarizes these instructions and their effects.

The effect of $\mathsf{Inc}(n)$ is simply that it adds 1 to the value of register R_n. The instruction $\mathsf{Dec}(n)$ does the opposite: it subtracts 1 from the value of R_n. If the value of R_n was 0, the instruction has no effect. The instruction $\mathsf{Jump}(n, m)$ works as follows: first, it tests whether R_m has value zero. If this is not the case, the instruction does nothing. If $R_m = 0$, then the program jumps to the m-th instruction. It is required that $m > 0$, so that $\mathsf{Jump}(n, 0)$ is not a legal instruction.

Definition 2.1.1 (Register machine program)**.** A *program* is a finite sequence of instructions $I_1, \ldots, I_k$, where each I_i is one of $\mathsf{Inc}(m), \mathsf{Dec}(m), \mathsf{Jump}(p, q)$.

When $\mathcal{P} = I_1, \ldots, I_k$ is a program, we say that $\mathcal{P}$ has *length* k; notation: $lh(\mathcal{P}) = k$.

Before we look at an example, there are a few remarks: first, note that we do not exclude the empty program with no instructions (the unique program with length 0). Next, while each of the instructions I_i has to be a syntactically correct instruction, it is not required that the Jump-instruction always refers to an existing instruction number. So, the one-liner

$$1 \quad \mathsf{Jump}(44, 22)$$

is a perfectly acceptable program. (Although it is a bit sloppy from a programming point of view.)

The following is an example of a simple program (on the right I have given the code in alternative notation)

1	Inc(2)	1	$R_2 \leftarrow R_2 + 1$
2	Dec(3)	2	$R_3 \leftarrow R_3 - 1$
3	Jump(3, 5)	3	`if` $R_3 = 0$ `goto` 5
4	Jump(1, 1)	4	`if` $R_1 = 0$ `goto` 1
5	Inc(0)	5	$R_0 \leftarrow R_0 + 1$

We have written line numbers on the left of the instructions, but technically these are not part of the syntax; the program determines the ordering of the instructions, and we only write line numbers for readability.

We can "run" this program on a register machine. Suppose that initially, the configuration of the machine was as in 2.1. Then we get the following sequence of configurations:

Configuration						Next instruction
10	0	4	2	4	⋯	1 Inc(2)
10	0	5	2	4	⋯	2 Dec(3)
10	0	5	1	4	⋯	3 Jump(3, 5)
10	0	5	1	4	⋯	4 Jump(1, 1)
10	0	5	1	4	⋯	1 Inc(2)
10	0	6	1	4	⋯	2 Dec(3)
10	0	6	0	4	⋯	3 Jump(3, 5)
10	0	6	0	4	⋯	5 Inc(0)
11	0	6	0	4	⋯	6 Halts

Thus, the execution of the program begins by applying the first instruction of the program to the initial configuration of the machine. It then executes the instructions as long as there are instructions to follow. In this particular case, the program halts because there is no 6-th instruction. This is not the only possible reason for a program to halt; this will be discussed shortly.

2.1.2 Computations

We now make a number of definitions, leading to a precise notion of a computation of a program. As suggested by the above example, the execution of a program $\mathcal{P}$ on a register machine should be a suitable sequence of configurations and instruction numbers. To this end, we first define the notion of a state of a machine.

Definition 2.1.2 (State of a register machine)**.** Let $\mathcal{P} = \langle I_1, \ldots, I_k \rangle$ be a program.

(i) A *$\mathcal{P}$-state* of a register machine (or simply a *state* when it is clear which program is meant) is a pair (C, p), where C is a configuration and where $0 \leq p \leq k+1$.

(ii) A state (C, p) is called *initial* when $p = 1$. In that case, we call C an *initial configuration*.

(iii) A state (C, p) is called *terminal*, or a *halting state*, when either $p = 0$ or $p = lh(\mathcal{P}) + 1$. In that case, C is called a *halting configuration*.

The idea is that the number p tells us which instruction is about to be executed. The instructions of $\mathcal{P}$ are numbered $1, \ldots, k$; if $p = k+1$, then that indicates that the program has executed the last instruction and has come to a halt. The value $p = 0$ is meant to indicate that the program attempted to execute a Jump-operation to a non-existing line number (i.e. it encountered an instruction $\mathsf{Jump}(i, j)$, found that $R_i = 0$ and tried to jump to instruction number j, but $j > k$, the length of the program). In both cases ($p = 0, p = k+1$), we say that the machine is in a halting state. Note that for the empty program there are only halting states.

The following definition captures the idea that the state of a machine completely determines the next state:

Definition 2.1.3. Let $S = (C, p)$ and $T = (D, q)$ be two $\mathcal{P}$-states. We define T to be a *successor state* of S when $1 \leq p \leq lh(\mathcal{P})$ and when the following conditions are met (where we write $C = \langle C(0), C(1), \ldots \rangle$):

Case 1. If the p-th instruction I_p of $\mathcal{P}$ is $\mathsf{Inc}(m)$, then

$$D = \langle C(0), \ldots, C(m-1), C(m)+1, C(m+1), \ldots \rangle, \text{ and } q = p+1$$

Case 2. If the p-th instruction I_p of $\mathcal{P}$ is $\mathsf{Dec}(m)$, then

$$D = \langle C(0), \ldots, C(m-1), C(m) \ominus 1, C(m+1), \ldots \rangle, \text{ and } q = p+1$$

Case 3. If the p-th instruction I_p of $\mathcal{P}$ is $\mathsf{Jump}(i, j)$, then we distinguish the following cases:

Subcase a. If $c(i) = 0$ and $j \leq k$, then $D = C$, and $q = j$.

Subcase b. If $c(i) = 0$ but $j > k$ then $D = C$ and $q = 0$.

Subcase c. If $c(i) \neq 0$ then $D = C$ and $q = p + 1$.

Note that any state has *at most one* successor state. Thus, we are modelling *deterministic computations*, in which each step is completely determined by the previous state of the machine.

We are now in a position to make the expected definition of what a computation of a register machine is:

Definition 2.1.4 (Computation)**.** Let $\mathcal{P} = \langle I_1, \ldots, I_k \rangle$ be a register machine program. A *computation sequence* of $\mathcal{P}$, or a $\mathcal{P}$*-computation sequence*, is a sequence (finite or infinite) of states $S = \langle S_0, S_1, \ldots \rangle$ such that

(i) S_0 is initial

(ii) if S_{i+1} is in S, then it is the $\mathcal{P}$-successor state of S_i.

A computation sequence S is a $\mathcal{P}$*-computation* if it satisfies the additional condition that whenever a non-terminal state S_i is in S, then the successor state is also in S.

Thus, if a computation is finite, the last state in the computation is necessarily a halting state. We simply view the computation sequences as approximations to computations; the reason for introducing these as a separate concept will become apparent later.

2.1.3 Computable functions

We have made precise how register machines operate by defining the concept of a register machine computation, but we have not indicated yet what exactly is being computed. Suppose we have a program $\mathcal{P}$; recall that an initial $\mathcal{P}$-state is simply a state $(C, 1)$, where C is an arbitrary configuration. Thus, we can start a $\mathcal{P}$-computation with any initial configuration we like.

We take advantage of this freedom by using the initial configuration to specify the input on which we want the program to operate. Just as the input of a numerical function $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ is an n-tuple of numbers, we wish to feed the program an n-tuple $x_1, \ldots, x_n$. We agree to specify this in the following manner: let the registers $R_1, \ldots, R_n$ contain the desired input values $x_1, \ldots, x_n$, and set all other registers to 0.

We may then run the program $\mathcal{P}$ on this specific initial configuration. If the program halts, then we regard the value of R_0 to be the output of the computation.

Definition 2.1.5 (Input and output of a program)**.** Let $\mathcal{P}$ be a program and let $x_1, \ldots, x_n$ be an n-tuple of natural numbers, with $n > 0$.

(i) The configuration $\langle 0, x_1, x_2, \ldots, x_n, 0, 0, \ldots\rangle$ is denoted by

$$C[x_1, \ldots, x_n]$$

and is called the *initial configuration for input* $x_1, \ldots, x_n$.

(ii) If there is a finite computation of $\mathcal{P}$ with initial configuration $C[x_1, \ldots, x_n]$, then we say that $\mathcal{P}$ *accepts input* $x_1, \ldots, x_n$, or that $\mathcal{P}$ *halts on input* $x_1, \ldots, x_n$. Otherwise, $\mathcal{P}$ is said to *diverge* on input $x_1, \ldots, x_n$.

(iii) If $\mathcal{P}$ accepts $x_1, \ldots, x_n$ and the halting configuration of the computation is D, then we call $D(0)$ the *output* of $\mathcal{P}$ on input $x_1, \ldots, x_n$.

Here's a simple example: Let $\mathcal{P}$ be the following program

$$\begin{array}{ll} 1 & \mathsf{Inc}(0) \\ 2 & \mathsf{Jump}(1,1) \end{array} \tag{2.2}$$

We wish to regard this program as operating on one input variable x. If we run it with input $x = 3$ for example, that amounts to specifying the initial configuration $C[3] = \langle 0, 3, 0, 0, 0, \ldots\rangle$. This leads to the following computation:

Configuration						Next instruction
0	3	0	0	0	$\cdots$	1 $\mathsf{Inc}(0)$
1	3	0	0	0	$\cdots$	2 $\mathsf{Jump}(1,1)$
1	3	0	0	0	$\cdots$	3 Halts

The program halts because $R_1 \neq 0$, so it moves to the next instruction, which doesn't exist. Thus $\mathcal{P}$ accepts input $x = 3$, and outputs $y = 1$.

By contrast, if we run $\mathcal{P}$ on input $x = 0$, the following happens:

Configuration						Next instruction
0	0	0	0	0	$\cdots$	1 $\mathsf{Inc}(0)$
1	0	0	0	0	$\cdots$	2 $\mathsf{Jump}(1,1)$
1	0	0	0	0	$\cdots$	$\mathsf{Inc}(0)$
2	0	0	0	0	$\cdots$	2 $\mathsf{Jump}(1,1)$
$\vdots$						$\vdots$

The machine goes into a loop from which it never recovers. Thus $\mathcal{P}$ diverges on input $x = 0$.

Note that we may also regard this program as operating on more than one input variable. For example, we could regard the above computations as computations on inputs $(3, 0)$ and $(0, 0)$, respectively. Since the program does

not mention the register R_2, the second input variable does not influence the output.

Now we make the key definition:

Definition 2.1.6 (Computable function)**.**

(i) Let $\mathcal{P}$ be a program, and let $n > 0$ be a number. Then the *(partial) function computed by* $\mathcal{P}$, denoted $\Psi_{\mathcal{P}}^{(n)}(x_1, \ldots, x_n)$ is defined by:

$$\Psi_{\mathcal{P}}^{(n)}(x_1, \ldots, x_n) = \begin{cases} z & \text{if there is a finite } \mathcal{P}\text{-computation on input } x_1, \ldots, x_n \text{ with output z} \\ \uparrow & \text{if no such computation exists, i.e. when } \mathcal{P} \text{ diverges on input } x_1, \ldots, x_n. \end{cases}$$

(ii) A (partial) function $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ is *computable* if there exists a program $\mathcal{P}$ such that $f = \Psi_{\mathcal{P}}^{(n)}$.

As a brief illustration, the program in example 2.2 was seen to accept all inputs $x > 0$, in which case it halts with value 1 in register R_0. Thus we find:

$$\Psi_{\mathcal{P}}^{(1)}(x) = \begin{cases} 1 & \text{if } x = 0 \\ \uparrow & \text{otherwise.} \end{cases}$$

A few remarks are in order. The first part of definition 2.1.6 associates to each program $\mathcal{P}$ not just one but a *sequence* of functions

$$\Psi_{\mathcal{P}}^{(1)}, \Psi_{\mathcal{P}}^{(2)}, \ldots,$$

one function for each arity n. Most of the time however, we have a specific arity in mind, and we only wish to regard the program as computing a function of that arity. If we are regarding the program as operating on one input variable we will often simply write $\Psi_{\mathcal{P}}$ for the function computed by $\mathcal{P}$. For example, the program from example 2.2 ignores all input registers but the first, so it is most natural to view this program as operating on one input variable.

It must also be stressed that when one writes a program, it is important to keep in mind which registers are going to be used as input registers and which registers are "free" for use by the program (as auxiliary registers). See the examples below.

The reader should be warned that in other texts one finds slightly different versions of the definition of computable function; these differences arise from the fact that along the way we have made various choices (namely, which set of instructions to use, how to represent input and output, etc.). In most cases it is straightforward to show that the notion of computability is not essentially dependent on these choices.

Finally, one encounters different notations: for example, some authors use $\mathcal{P}(x_1, \ldots, x_n) = y$ for $\Psi_{\mathcal{P}}^{(n)}(x_1, \ldots, x_n) = y$.

2.1.4 Examples

Because of the parsimonious syntax, it may be quite a bit of work to write a program computing even a relatively simple function. In this section we therefore limit ourselves to a few elementary examples of computable functions; later we will develop some tools for showing the computability of more involved examples.

1. Let us start by considering the empty program $\mathcal{P}$ with no instructions. What function $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ does it compute? Well, if we run the program on input $x_1, \ldots, x_k$, then we find that the initial state is at the same time the halting state, since the machine is supposed to execute instruction 1, which doesn't exist. This is a computation of length 0. Since none of the registers is being altered, the register R_0 will have value 0 in the halting configuration. Therefore, we have

   $$\Psi_{\mathcal{P}}^{(n)}(x_1, \ldots, x_n) = 0,$$

   the constant function with value 0.

2. Another extreme example: can we find a program which computes the empty (completely undefined) function? That is, can we construct a program $\mathcal{Q}$ such that

   $$\Psi_{\mathcal{Q}}^{(n)}(x_1, \ldots, x_n) \uparrow$$

   for all inputs $x_1, \ldots, x_n$? Such a program is supposed to never halt, regardless of the input. Consider

   1 $\quad$ $\mathsf{Jump}(0, 1)$

   Since we always start the computation with $R_0 = 0$, this program will do. Of course, there are other possibilities.

3. Consider the following program $\mathcal{R}$:

   1 $\quad$ $\mathsf{Jump}(1, 5)$
   2 $\quad$ $\mathsf{Inc}(0)$
   3 $\quad$ $\mathsf{Dec}(1)$
   4 $\quad$ $\mathsf{Jump}(2, 1)$

   What is $\Psi_{\mathcal{R}}^{(1)}$? First the program checks whether the input register is 0. If so, it jumps to instruction 5, which is nonexistent. Thus, on input 0, the program halts and outputs 0 (the output register hasn't been altered). If the input is nonzero, it will move on to instructions 2 and 3, which increase the value of the output register by 1 and decrease the value of the input register by 1. Then it reaches instruction 4. This instruction tests if $R_2 = 0$, which always succeeds, since we are not touching that register. Thus we jump back to instruction 1 and repeat until $R_1 = 0$.

   Thus, we find that

   $$\Psi_{\mathcal{R}}(x) = x,$$

i.e. that $\mathcal{R}$ computes the identity function.

The trick in instruction 4 is a useful one to structure programs; in programming terms, we have defined an *unconditional jump*. It is important that the register involved in this definition (here R_2) is not being used as input register or output register.

4. Example 3 can be generalized to show that all projection functions $\mathbf{U}_k^n$ are computable. However, a bit of care is needed. If we would use the above program to compute a *binary* function, we would put the input variables x_1, x_2 in registers R_1, R_2, respectively. Then the jump in line 4 would not work as desired, since R_2 is not necessarily zero any more. Thus, we have to pick another auxiliary register.

5. How would we compute the successor function? The following program $\mathcal{S}$ is a good solution:

   1 $\mathsf{Jump}(1,5)$
   2 $\mathsf{Inc}(0)$
   3 $\mathsf{Dec}(1)$
   4 $\mathsf{Jump}(2,1)$
   5 $\mathsf{Inc}(0)$

   Note that this is just the program of example 3, but with one instruction added at the end. Now if we run this on input x, it will first transport x into register R_0, and then add 1 to that.

6. Consider the following program $\mathcal{T}$.

   1 $\mathsf{Jump}(1,6)$
   2 $\mathsf{Dec}(1)$
   3 $\mathsf{Jump}(1,7)$
   4 $\mathsf{Dec}(1)$
   5 $\mathsf{Jump}(2,1)$
   6 $\mathsf{Inc}(0)$

   We leave it as an exercise to the reader to show that this program computes the characteristic function of the set of even numbers.

7. The reader may adapt the program from example 6 to show that the semi-characteristic function of the set of even numbers is computable.

2.2 Recursive functions are computable

In this section we study the class of computable functions in more detail. We shall first develop a few tools which make it easier to construct programs, and introduce two conventions about programs which make reasoning about them slighly easier. We will then set out to demonstrate that all recursive functions are computable.

2.2.1 Macros and more

In the examples of section 2.1.4 we met the unconditional jump, disguised as a jump which always succeeds. Now that we know that the unconditional jump can be defined, we can introduce the shorthand notation $\mathsf{Jump}(j)$ for $\mathsf{Jump}(i, j)$ in situations where it is guaranteed that register R_i will have value 0.

We say that $\mathsf{Jump}(j)$ is a *macro*. A macro is nothing more than a shorthand notation for a fragment of program code. We use macros as if they were genuine instructions knowing that, when desired, we can always expand them by the program fragment which they abbreviate.

Another example of a useful macro is $\mathsf{Zero}(m)$, which has the effect of setting the value of R_m to 0. It may be taken to be shorthand for the following code:

```
1   Jump(m, 4)
2   Dec(m)
3   Jump(i, 1)
```

where i is the number of a "fresh" register, different from m. Again, we use this macro in programs as if it were a genuine instruction, for example as in:

```
1   Inc(2)
2   Jump(2, 4)
3   Zero(1)
4   Inc(1)
```

Now in order to expand this macro (and hence replace the above "pseudocode" by a real program, we have to replace the macro in line 3 by its code. We obtain:

```
1   Inc(2)
2   Jump(2, 6)
3   Jump(1, 6)
4   Dec(1)
5   Jump(4, 3)
6   Inc(1)
```

Note the following things: first we have adjusted the second instruction so that it still refers to the correct instruction number; second, we have done this for the first instruction of the macro as well. Finally, we have chosen a suitable register number for the fifth instruction, namely one which is not in use by the rest of the program.

In the exercises we will outline how one would formally prove that all macros of a certain kind can always be correctly expanded. For now, we simply regard them as a useful shorthand and hope that the reader is convinced that one can always replace the pseudocode by genuine programs if desired.

We mention a few other macros. The first is $\mathsf{Trans}(i, j)$. It transfers the values of register R_i to register R_j. One may use the following code, which in turn uses previously defined macros:

Macro	Effect
$\mathsf{Zero}(m)$	Sets the value of R_m to 0
$\mathsf{Jump}(j)$	Jump to instruction j
$\mathsf{Jump}_{\neq}(i,j)$	If $R_i \neq 0$, jump to instruction j; do nothing otherwise
$\mathsf{Jump}(i,j,k)$	If $R_i = R_j$, jump to instruction k; do nothing otherwise
$\mathsf{Trans}(i,j)$	Transfers the value of R_i to R_j; as a side-effect, sets R_i to 0
$\mathsf{Copy}(i,j)$	Copies the value of R_i into R_j
$\mathsf{Swap}(i,j)$	Swaps the values of R_i and R_j

Table 2.1: A few useful macros

1 $\mathsf{Zero}(j)$
2 $\mathsf{Jump}(i,6)$
3 $\mathsf{Inc}(j)$
4 $\mathsf{Dec}(i)$
5 $\mathsf{Jump}(2)$

The above macro $\mathsf{Trans}(i,j)$ destroys the value of R_i in the process of copying it to R_j. In many cases, one may wish to make a genuine copy and retain the value of R_i. To this end, we define the macro $\mathsf{Copy}(i,j)$ which copies the value of R_i to R_j, leaving R_i unharmed.

1 $\mathsf{Zero}(j)$
2 $\mathsf{Zero}(k)$
3 $\mathsf{Jump}(i,8)$
4 $\mathsf{Inc}(j)$
5 $\mathsf{Inc}(k)$
6 $\mathsf{Dec}(i)$
7 $\mathsf{Jump}(3)$
8 $\mathsf{Trans}(k,i)$

Here, we have picked an auxiliary register R_k (which should be different from all other registers being used) to temporarily store information.

Many more macros can be defined. We mention two other conditional Jump-operations. The first, $\mathsf{Jump}_{\neq}(i,j)$ does the following: if $R_i \neq 0$ then jump to instruction j, otherwise do nothing. The second, $\mathsf{Jump}(i,j,k)$ jumps to instruction k if $R_i = R_j$ and does nothing otherwise.

Also, one may define a macro $\mathsf{Swap}(i,j)$ which swaps the values of registers R_i and R_j. We leave it as an exercise to the reader to define those macros.

In Table 2.2.1, we summarize the macros introduced so far for future reference.

2.2.2 Composition

As illustrated by the problem of expanding macros, one often wishes to combine program fragments, and this combining involves both renumbering of instructions and the use of "fresh" registers. We will now define two operations on programs which help us carry out such manipulations more formally.

Construction 2.2.1. Let $\mathcal{Q} = \langle I_1, \ldots, I_k \rangle$ be a program.

(i) Define $Ren(\mathcal{Q}, n)$ to be the program obtained by replacing each instruction $\mathsf{Jump}(i,j)$ by the instruction $\mathsf{Jump}(i, j+n)$.

(ii) Define $Shift(\mathcal{Q}, m)$ to be the program obtained by replacing each instruction $\mathsf{Inc}(j)$ by $\mathsf{Inc}(j+m)$, each $\mathsf{Dec}(j)$ by $\mathsf{Dec}(j+m)$ and each $\mathsf{Jump}(i, j)$ by $\mathsf{Jump}(i+m, j)$.

These constructions allow us to form various combinations of programs in such a way that we have control over their interaction. Perhaps the simplest way of combining programs is to form their composition: given two programs $\mathcal{P}$ and $\mathcal{Q}$, where $lh(\mathcal{P}) = k$, define

$$\mathcal{P}\mathcal{Q} = \mathcal{P} * Ren(\mathcal{Q}, k),$$

where the * denotes concatenation of sequences. The program $\mathcal{P}\mathcal{Q}$ is called the *concatenation* of $\mathcal{P}$ and $\mathcal{Q}$.

There is one problem: it could happen that one of the instructions in $\mathcal{P}$ is a Jump-operation which refers to an instruction l with $l > k$. This instruction was intended to bring the program to a halt, but now it refers to an instruction in the $\mathcal{Q}$-part of the composite program. This is undesirable, so we will from now on assume that this doesn't happen:

Convention 2.2.2. *No program $\mathcal{P}$ shall contain instructions $\mathsf{Jump}(i, j)$ with $j > lh(\mathcal{P})$.*

Thus, we end the (ab)use of the Jump-instruction to bring computations to a halt. Clearly every program can be replaced by a program which satisfies this condition and which computes the same function. Example 5 is an example of a composition of programs: it was built from the program computing the identity function in example 3 and the one-liner $\mathsf{Inc}(0)$.

We are now interested in the following question: given two programs $\mathcal{P}$ and $\mathcal{Q}$, which compute functions f and g (of one variable, say), can we combine them in such a way as to obtain a program computing the composite function gf?

It should be clear that simply concatenating the programs will not do; for example, we have to ensure that the output of $\mathcal{Q}$ becomes the input of $\mathcal{P}$, and also that the computation $\mathcal{P}$ does not get affected by the fact that $\mathcal{Q}$ may have changed various registers.

We will now overcome those problems. The following notation will be helpful:

Definition 2.2.3. Let $\mathcal{P}$ be a program, and let $i > 0$. We say that $\mathcal{P}$ *uses* a register R_i if $\mathcal{P}$ contains an instruction of the form $\mathsf{Inc}(i)$, $\mathsf{Dec}(i)$ or $\mathsf{Jump}(i, j)$. We also put

$$Use(\mathcal{P}) = \{0\} \cup \{i | \mathcal{P} \text{ uses } R_i\},$$

and

$$max(\mathcal{P}) = max(Use(\mathcal{P})).$$

Proposition 2.2.4. *The computable functions are closed under generalized composition.*

Proof. Suppose $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ is computed by $\mathcal{P}$ and $g_1, \ldots, g_k : \mathbb{N}^n \longrightarrow \mathbb{N}$ are computed by $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$, respectively. We must exhibit a program computing $f\langle g_1, \ldots, g_k\rangle$.

In order to do so, we first define a sequence of numbers $N_1, \ldots, N_k$ as

$$\begin{aligned} N_1 &= max\{max(\mathcal{P}), k, n\} + 1 \\ N_2 &= N_1 + max(\mathcal{Q}_1) + n + 1 \\ &\vdots \\ N_k &= N_{k-1} + max(\mathcal{Q}_{k-1}) + n + 1 \end{aligned}$$

These numbers are chosen in such a way that each sub-program has enough space on the register machine to do its computation without interfering with other sub-programs. Thus, their exact value isn't too important; all that matters is that they are big enough.

The program which computes the composite function $f\langle g_1, \ldots, g_k\rangle$ is now given schematically as the composition of the following sequence of programs:

$$\begin{array}{|l|} \hline \mathsf{Copy}(1, N_1 + 1) \cdots \mathsf{Copy}(n, N_1 + n) \\ \qquad\qquad\qquad \vdots \\ \mathsf{Copy}(1, N_k + 1) \cdots \mathsf{Copy}(n, N_k + n) \\ \mathsf{Zero}(1) \cdots \mathsf{Zero}(N_1 - 1) \\ Shift(\mathcal{Q}_1, N_1) \cdots Shift(\mathcal{Q}_k, N_k) \\ \mathsf{Copy}(N_1, 1) \cdots \mathsf{Copy}(N_k, k) \\ \mathcal{P} \\ \hline \end{array}$$

□

We illustrate the above schematic proof by considering again the example of the projection $\mathbf{U}_3^3$ and the identity function. Programs $\mathcal{P}$ and $\mathcal{Q}$ may be taken to be

1	$\mathsf{Jump}(1, 5)$	1	$\mathsf{Jump}(3, 5)$
2	$\mathsf{Dec}(1)$	2	$\mathsf{Dec}(3)$
3	$\mathsf{Inc}(0)$	3	$\mathsf{Jump}(0)$
4	$\mathsf{Jump}(4, 1)$	4	$\mathsf{Trans}(4, 1)$
5	$\mathsf{Dec}(4)$	5	$\mathsf{Dec}(4)$

We have added "dummy" instructions at the end of these programs in order to make sure they obey our convention.

First, note that we have $Use(\mathcal{P}) = \{0, 1, 4\}$ and $max(\mathcal{P}) = 4$; thus we find $N_1 = max\{4, 1, 3\} + 1 = 5$. The new program now is the composite of the following programs:

$\mathsf{Copy}(1,6)$	First we copy the input variables x, y, z
$\mathsf{Copy}(2,7)$	to fresh registers R_6, R_7, R_8;
$\mathsf{Copy}(3,8)$	
$\mathsf{Zero}(1)$	Then we clean up the old input registers
$\mathsf{Zero}(2)$	
$\mathsf{Zero}(3)$	
$Shift(\mathcal{Q},5)$	We run $\mathcal{Q}$ on input x, y, z
$\mathsf{Copy}(5,1)$	and move the output $g(x,y,z)$ to R_1
$\mathcal{P}$	Finally, we run $\mathcal{P}$ on input $g(x,y,z)$

Note how the program $\mathcal{Q}$ is included with registers shifted, and that $\mathcal{P}$ is included but with its instructions renumbered. Of course, this program still contains macros, so should be regarded as shorthand. We demonstrate the procedure schematically by running the program on input (x, y, z):

0	x	y	z	0	0	0	0	0	0	$\cdots$	Copy inputs
0	x	y	z	0	0	x	y	z	0	$\cdots$	Erase
0	0	0	0	0	0	x	y	z	0	$\cdots$	Run $Shift(\mathcal{Q},5)$
0	0	0	0	0	z	x	y	0	0	$\cdots$	Copy output
0	z	0	0	0	z	x	y	0	0	$\cdots$	Run $\mathcal{P}$
z	0	0	0	0	z	x	y	0	0	$\cdots$	Halt.

2.2.3 Primitive recursion

The aim of this section is to show that the computable functions are closed under primitive recursion. We first introduce a new macro and a convention to streamline the proof.

First, our convention: suppose $\mathcal{P}$ computes a function $f(x_1, \ldots, x_n)$. All we know is that when $f(x_1, \ldots, x_n) = y$, then the program will give us output y in R_0 after the computation with initial configuration $C[x_1, \ldots, x_n]$ has halted. What we do not know, is what is contained in the rest of the registers. In particular, the values of $R_1, \ldots, R_n$ may have been altered during the computation. Sometimes (especially when we wish to implement $\mathcal{P}$ as part of a larger program) it is convenient to assume that when the computation is finished, the input registers still contain the input variables. So, when the initial configuration was $\langle 0, x_1, \ldots, x_n, 0, \ldots\rangle$, the final configuration reads $\langle f(x_1, \ldots, x_n), x_1, \ldots, x_n, 0, \ldots\rangle$ (provided $f(x_1, \ldots, x_n)\downarrow$ of course).

It is always possible to modify our program in such a way that it will satisfy this extra requirement. We may even get it to clean up the rest of the registers it used. In this section, we shall assume that we are working with programs satisfying this condition.

Convention 2.2.5. *If $\mathcal{P}$ computes $f : \mathbb{N}^k \longrightarrow \mathbb{N}$, then for every input $x_1, \ldots, x_k$, $\mathcal{P}$ will halt only in configurations of the form $\langle f(x_1, \ldots, x_k), x_1, \ldots, x_k, 0, \ldots\rangle$.*

We also introduce one new macro. If $f(x_1, \ldots, x_n)$ is a computable function and $\mathcal{P}$ is a program computing it, then we can introduce the macro

$$\mathcal{P}(i_1, \ldots, i_n) \longrightarrow j$$

The meaning of this macro is: compute the value of $f(R_{i_1}, \ldots, R_{i_n})$ and put the result in register R_j.

Using the ideas from the previous section it should be clear how to expand this macro when we encounter it in a program: first, find ourselves enough room where we are not disturbed by the rest of the program; then, copy the values of $R_{i_1}, \ldots, R_{i_n}$ to there (i.e. into registers $R_{N+1}, \ldots, R_{N+n+1}$ where N is a big enough number). Next, execute $Shift(\mathcal{P}, N)$; the result will be $R_N = f(R_{i_1}, \ldots, R_{i_n})$, provided this is defined. Finally, copy this result to register R_j.

Proposition 2.2.6. *The computable functions are closed under primitive recursion.*

Proof. Suppose we are given functions $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ computed by $\mathcal{P}$ and $g : \mathbb{N}^{k+2} \longrightarrow \mathbb{N}$ computed by $\mathcal{Q}$, and define $h : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ by primitive recursion from f, g. The task is to build a program $\mathcal{R}$ computing h.

The idea behind the program $\mathcal{R}$ is as follows: when presented with input $\overrightarrow{x}, y$, it starts by computing $h(\overrightarrow{x}, 0) = f(\overrightarrow{x})$. If $y = 0$, then we're done. Otherwise, it must compute $h(\overrightarrow{x}, 1) = g(\overrightarrow{x}, 1, h(\overrightarrow{x}, 0))$. If that is done, it checks whether $y = 1$. If so, we're done, if not, we move on to computing of $h(\overrightarrow{x}, 2)$, etc. The way to keep track of where we are in the computation is to use a counter c which counts from 0 to y.

Let $N = k + 3$. Our program is the concatenation of the following:

1 $\mathsf{Copy}(i, N + i) \qquad i = 1, \ldots, k$
2 $\mathcal{P}(N + 1, \ldots, N + k) \longrightarrow N + k + 2$
3 $\mathsf{Jump}(k + 1, N + k + 1, 7)$
4 $\mathcal{Q}(N + 1, \ldots, N + k + 2) \longrightarrow N + k + 2$
5 $\mathsf{Inc}(N + k + 1)$
6 $\mathsf{Jump}(3)$
7 $\mathsf{Copy}(N + k + 2, 0)$

It is left as an exercise to the reader to show that the function computed by this program is indeed h (and in particular, that it has the correct domain of definition). □

We illustrate the working of the program by running it on input $(x, 2)$ (i.e. $k = 1$). Thus, $N = 4$.

0	x	2	0	0	0	0	0	0	⋯	$\mathsf{Copy}(1, 5)$
0	x	2	0	0	x	0	0	0	⋯	$\mathcal{P}(5) \longrightarrow 7$
0	x	2	0	0	x	0	f(x)	0	⋯	$\mathsf{Jump}(2, 6, 7)$
0	x	2	0	0	x	0	f(x)	0	⋯	$\mathcal{Q}(5, 6, 7) \longrightarrow 7$
0	x	2	0	0	x	0	h(x,1)	0	⋯	$\mathsf{Inc}(6)$
0	x	2	0	0	x	1	h(x,1)	0	⋯	$\mathsf{Jump}(3)$
0	x	2	0	0	x	1	h(x,1)	0	⋯	$\mathsf{Jump}(2, 6, 7)$

0	x	2	0	0	x	1	h(x,1)	0	$\cdots$	$\mathcal{Q}(5,6,7) \longrightarrow 7$
0	x	2	0	0	x	1	h(x,2)	0	$\cdots$	$\mathsf{Inc}(6)$
0	x	2	0	0	x	2	h(x,2)	0	$\cdots$	$\mathsf{Jump}(3)$
0	x	2	0	0	x	2	h(x,2)	0	$\cdots$	$\mathsf{Jump}(2,6,7)$
0	x	2	0	0	x	2	h(x,2)	0	$\cdots$	$\mathsf{Copy}(7,0)$
h(x,2)	x	2	0	0	x	2	h(x,2)	0	$\cdots$	Halts

2.2.4 Minimalization

We have shown the basic functions zero, successor and projections to be computable; we have also shown that the computable functions are closed under composition and primitive recursion. It remains to be shown that they are closed under unbounded minimalization.

Proposition 2.2.7. *The computable functions are closed under minimalization.*

Proof. Let $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ be computed by $\mathcal{P}$. We should construct a program computing $h(\overrightarrow{x}) = \mu y.(f(\overrightarrow{x}, y) = 0)$. Try:

1 $\mathcal{P}(1, \ldots, k+1) \longrightarrow k+2$
2 $\mathsf{Jump}(k+2, 0, 6)$
3 $\mathsf{Zero}(k+2)$
4 $\mathsf{Inc}(k+1)$
5 $\mathsf{Jump}(1)$
6 $\mathsf{Copy}(k+1, 0)$

□

We illustrate this again by a simple example; take $f(x, y) = x \ominus y$. Let us run the program on input value 2:

0	2	0	0	$\cdots$	$\mathcal{P}(1,2) \longrightarrow 3$
0	2	0	2	$\cdots$	$\mathsf{Jump}(3,0,6)$
0	2	0	2	$\cdots$	$\mathsf{Zero}(3)$
0	2	0	0	$\cdots$	$\mathsf{Inc}(2)$
0	2	1	0	$\cdots$	$\mathsf{Jump}(1)$
0	2	1	0	$\cdots$	$\mathcal{P}(1,2) \longrightarrow 3$
0	2	1	1	$\cdots$	$\mathsf{Jump}(3,0,6)$
0	2	1	1	$\cdots$	$\mathsf{Zero}(3)$
0	2	1	0	$\cdots$	$\mathsf{Inc}(2)$
0	2	2	0	$\cdots$	$\mathsf{Jump}(1)$
0	2	2	0	$\cdots$	$\mathcal{P}(1,2) \longrightarrow 3$
0	2	2	0	$\cdots$	$\mathsf{Jump}(3,0,6)$
0	2	2	0	$\cdots$	$\mathsf{Copy}(2,0)$
2	2	2	0	$\cdots$	Halts

Combining the results from the previous sections, we have thus proved:

Theorem 2.2.8. *Every recursive function is register machine computable.*

Proof. The basic functions are computable, and the computable functions are closed under composition, primitive recursion and minimalization. □

2.3 Gödel numberings

We now embark on a proof of the converse of Theorem 2.2.8, namely that every computable function is recursive. This is more difficult, since the class of computable functions is not presented to us via an inductive definition. The key idea is to use arithmetization, an encoding of register machine programs and computations. The idea of encoding certain structures as natural numbers in order to reason about them numerically was first used by Gödel.

2.3.1 Coding of programs

We are going to encode the following sets: the set of register machine instructions, the set of programs, the set of states of a program, and the set of finite computations. All this has to be done in an effective way, so that we can reason about the codes using (primitive) recursive functions.

In what follows we will heavily use our coding of finite sequences. The code of a finite sequence $\langle x_1, \ldots, x_n \rangle$ will again be denoted $[x_1, \ldots, x_n]$. We shall assume that the length function $lh(x)$ is primitive recursive. The decoding functions of sequences will also be written $(-)_i$. For convenience, we abbreviate expressions such as $((x)_i)_j$ to $(x)_{ij}$.

We start by numbering instructions and programs. If I is an instruction, we will write $\sharp(I)$ for its code.

- $\sharp(\mathsf{Inc}(n)) := [1, n]$
- $\sharp(\mathsf{Dec}(n)) := [2, n]$
- $\sharp(\mathsf{Jump}(i, j)) := [3, i, j]$

Now a program $\mathcal{P}$ is a sequence of instructions, so if $\mathcal{P} = \langle I_1, \ldots, I_k \rangle$, $k > 0$, then we define

- $\sharp(\mathcal{P}) = [\sharp(I_1), \ldots, \sharp(I_k)]$

In case of the empty program, we put $\sharp(\mathcal{P}) = 0$.

Lemma 2.3.1. *The predicate $Prog(y)$, which expresses that y is the code of a register machine program, is primitive recursive.*

Proof. $Prog(y)$ is true precisely when either $y = 0$, or $y = [x_1, \ldots, x_n]$ and for each $i = 1, \ldots, n$ we have that

- $lh(x_i) = 2$ or $lh(x_i) = 3$
- if $lh(x_i) = 2$, then $(x)_{i1} = 1$ or $(x)_{i1} = 2$, and $(x)_{i2} > 0$
- if $lh(x_i) = 3$, then $(x)_{i1} = 3$ and $(x)_{i3} > 0$.

□

We could even add in the extra requirement that the program satisfies our convention that jump instructions only refer to existing instruction numbers, but for the sequel it makes little difference.

We also have:

Lemma 2.3.2. *The functions $len(y) =$ length of the program coded by y, and $R(y) =$ largest register being used by the program coded by y (or 0 if $y = 0$) are both primitive recursive.*

Proof. The first function simply computes the length of the sequence coded by y. The function $R(y)$ takes each instruction code $(y)_i$ and extracts the second component (which refers to the registers); it then takes the maximum of those, or 0 if the program was empty. Formally: $R(y) = max_{z \leq y}((z)_{z2})$. □

Next, recall that a configuration of a register machine was an assignment of values to each of its registers. Since there are infinitely many registers, this information cannot be encoded as a single number. However, because a program will always access only a finite number of registers during a computation sequence, it is sufficient to simply encode large enough initial segments of configurations. When C is a configuration then we write $C_{|n}$ for the first $n+1$ elements, i.e. $C_{|n} = \langle C(0), \ldots, C(n) \rangle$. Then simply put $\sharp(C_{|n}) := [C(0), \ldots, C(n)]$.

We can now encode the $\mathcal{P}$-states of a machine; these are simply pairs (C, p) of a configuration and an instruction number. When $C_{|n}$ is an initial segment of a state and $p \leq lh(\mathcal{P}) + 1$ a number, we put $\sharp(C_{|n}, p) := [\sharp(C_{|n}), p]$. (Note that in the definition of a state we allowed $p = 0$ and $p = lh(\mathcal{P}) + 1$.)

Lemma 2.3.3. *The following predicates are primitive recursive:*

- $State(e, y) \Leftrightarrow$ *e is the code of a program $\mathcal{P}$ and y is the code of a $\mathcal{P}$-state.*
- $Initial(e, y) \Leftrightarrow$ *e is the code of a program $\mathcal{P}$ and y is the code of an initial $\mathcal{P}$-state*
- $Terminal(e, y) \Leftrightarrow$ *e is the code of a program $\mathcal{P}$ and y is the code of a terminal $\mathcal{P}$-state*

Proof. For the first part, use the fact that

$$\begin{aligned} State(e, y) \quad \Leftrightarrow \quad & (e > 0 \longrightarrow Prog(m) \wedge (y)_1 \leq lh(e) + 1 \wedge lh((y)_0) > R(y)) \\ & \wedge (e = 0 \longrightarrow p = 1) \end{aligned}$$

For the second part, add $(y)_2 = 1$. For the third, ask $p = 0 \vee p = lh(y) + 1$. □

The condition $lh((y)_1) > R(y)$ tells that the length of the initial segment of the coded configuration is long enough, i.e. that it comprises all registers being used by the program.

Finally, we show that finding the successor state of a given state can be done effectively in codes: define a function $Succ(e, y)$ as:

1. $Succ(e, y) = [D_{|n}, q]$ if $e = [\sharp(I_1), \ldots, \sharp(I_k)]$ codes a program $\mathcal{P}$ and y codes a $\mathcal{P}$-state $y = [[C(0), \ldots, C(n)], p]$ which is non-terminal. In that case, we find $D_{|n}$ and q by case distinction:
   a) if $\sharp(I_p) = [1, j]$ then take
   $$q = p + 1 \text{ and } D_{|n} = [C(0), \ldots, C(j + 1), \ldots, C(n)]$$
   b) if $\sharp(I_p) = [2, j]$ then take
   $$q = p + 1 \text{ and } D_{|n} = [C(0), \ldots, C(j) \ominus 1, \ldots, C(n)]$$
   c) if $\sharp(I_p) = [3, u, v]$ then we have subcases:
      i. if $C(u) = 0$ and $v \leq k$ then take $q = v$ and $D_{|n} = C_{|n}$
      ii. if $C(u) = 0$ but $v > k$ then take $q = 0$ and $D_{|n} = C_{|n}$
      iii. if $C(u) \neq 0$ then $q = p + 1$ and $D_{|n} = C_{|n}$
2. $Succ(e, y) = 0$ otherwise (i.e. if e doesn't code a program or y is not a non-terminal state of the program coded by e).

2.3.2 The T-predicate

Having all the encodings at our disposal, we can now define an important predicate, called *Kleene's T-predicate*. It has the following meaning:

$$\begin{aligned} T(e, x, y) \quad \Leftrightarrow \quad & e \text{ codes a program } \mathcal{P} \text{ and} \\ & y \text{ codes a finite computation of } \mathcal{P} \text{ on input } x. \end{aligned}$$

Using the machinery from the previous section, one easily shows:

Proposition 2.3.4. *The T-predicate is primitive recursive.*

Proof. Indeed, we define

$$\begin{aligned} T(e, x, y) \quad \Leftrightarrow \quad & Prog(e) \\ & \wedge\, Initial(e, (y)_1) \\ & \wedge\, lh((y)_{11}) > R(e) \\ & \wedge\, (y)_{112} = x \\ & \wedge\, \forall 1 \leq i < lh(y).(y)_{i+1} = Succ(e, (y)_i) \\ & \wedge\, Terminal(e, (y)_{lh(y)}) \end{aligned}$$

□

We also introduce the function $U : \mathbb{N} \longrightarrow \mathbb{N}$, defined by

$$U(y) = (y)_{lh(y)11}.$$

If y codes a computation $(C_0, p_0), \ldots, (C_n, p_n)$, then $(y)_{lh(y)}$ is the code of the terminal state, and $(y)_{lh(y)1}$ is the code of C_n, the terminal configuration. Thus, $U(y) = (y)_{lh(y)11}$ is the value of the first register in the terminal configuration. The function U is primitive recursive and is called the *outcome function*.

The T-predicate codes computations of programs which act on one input variable. It is straightforward to define, for each $n > 0$, a predicate

$$\begin{aligned} T_n(e, x_1, \ldots, x_n, y) \quad \Leftrightarrow \quad & e \text{ codes a program } \mathcal{P} \text{ and} \\ & y \text{ codes a computation of } \mathcal{P} \text{ on input } x_1, \ldots, x_n. \end{aligned}$$

Similarly, we have outcome functions $U_n(y)$ for such predicates.

We can now put everything together and show:

Theorem 2.3.5. *Every computable function is recursive.*

Proof. Let $f : \mathbb{N} \longrightarrow \mathbb{N}$ be computable. By definition, there is a program $\mathcal{P}$ which computes it, i.e. $f(x) = \Psi_{\mathcal{P}}(x)$. Now $\mathcal{P}$ has a code, say $\sharp(\mathcal{P}) = e$. Then we have

$$f(x) = \Psi_{\mathcal{P}}(x) = U(\mu y.T(e, x, y)).$$

This shows that f is recursive. The proof for functions $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ is similar. □

Our original goal was to prove the above theorem in order to establish that the recursive functions and the computable functions coincide, but we have in fact shown a lot more. Let us reformulate things a bit:

Theorem 2.3.6 (Normal Form Theorem for computations)**.** *For every recursive function f there is a number e such that*

$$f(x) = U(\mu y.T(e, x, y)).$$

Proof. Indeed, f is computable by some program $\mathcal{P}$. □

This is called a normal form result since it shows that every recursive function can be computed using a search over the T-predicate. We will see many more useful applications of the T-predicate later on.

2.4 Intermezzo: busy beavers*

So far, we have not seen any functions which are not computable. We know for cardinality reasons that such functions must exist: there are uncountably many functions on the natural numbers, but countably many computable functions. This observation does not, however, give us any interesting examples of non-computable functions. In this short section, we discuss a nice example relating to the machine model which we have studied in this chapter.

2.4.1 The challenge

Suppose we fix a number $n > 0$, and consider register machine programs of length n. Given such a program $\mathcal{P}$, we may run it on initial configuration $C(i) = 0$, where all registers are set to zero. If it halts, we check what the largest value of any of the registers is.

We translate this idea into a challenge: construct a program $\mathcal{P}$ of length n which halts on input 0 and which produces a very large number in one of the registers.

It is clear that without loss of generality we may ask the program to create this value in R_0, and only use registers up to R_n. However, to participate in the competition it is not sufficient to simply submit your program; you must also tell how many computation steps it will take the program to halt. The reason for this is, that the judges must be able to verify whether your program indeed does what you say it does. They are willing to run your program for long enough time, but not indefinitely.

2.4.2 The busy beaver function

Now for each n, we have a competition of creating the most industrious program of length n. Will this competition always have a winner? Well, we should first note that there are only finitely many programs of length n, once we impose the condition that they only use the registers $R_0, \ldots, R_n$ (as well as our convention that they don't jump to non-existing instructions).

Among those programs, there are those which halt on input 0. Thus we have a non-empty, finite set of programs $\mathcal{P}$ which halt on input 0, and output $\Psi_{\mathcal{P}}(0)$. Let us call this set $M(n)$. We now let

$$\Sigma(n) = max\{\Psi_{\mathcal{P}}(0) | \mathcal{P} \in M(n)\}.$$

Note that there may be several programs $\mathcal{P}$ which are producing this maximum output. These programs are called *busy beavers*, and Σ is called the *busy beaver function*. While some small values of this function are known, it grows extremely fast, and indeed can be shown to be non-computable not only in practice but also in principle.

2.4.3 Uncomputability of the busy beaver function

The proof that the function Σ is not computable has the flavor of a diagonal argument. We first suppose, towards a contradiction, that it were computable by some program, say $\mathcal{P}$. Then we build a new program as the composition of a program computing the function $f(x) = 2x$, $\mathcal{P}$ and of a program computing the successor function. This composite program, let us call it $\mathcal{Q}$, thus computes the function

$$x \mapsto \Sigma(2x) + 1.$$

Let m be the length of this program $\mathcal{Q}$. Now consider the following program:

Inc(1)
⋮ (m times)
Inc(1)
$\mathcal{Q}$

This program, call it $\mathcal{R}$, clearly has $2m$ instructions. Let $N = 2m$.

Now consider what happens if we run $\mathcal{R}$ on input 0. It will first create the value m in input register R_1; then it will run $\mathcal{Q}$ on input m. By construction of $\mathcal{Q}$, the output will be $\Sigma(2m) + 1 = \Sigma(N) + 1$. But $\Sigma(N)$ is the largest possible output for a program with N instructions. Contradiction, and hence Σ is not computable.

2.5 Exercises

Register machines

Exercise 2.1 (*Determinism of computations)**.** Verify that every state of a program has at most one successor state.

Exercise 2.2 (**)**.** Without using macros, give a register machine program which outputs 1 if the input is divisible by 3, and undefined otherwise.

Exercise 2.3 (**)**.** Without using macros, construct a program which computes the function $f(x, y, z) = x + z$.

Exercise 2.4 (**)**.** Without using macros, write a program computing the predicate $x \leq y$.

Exercise 2.5 (**)**.** Let $\mathcal{P}$ be the following program:

1 Jump(2,8)
2 Dec(1)
3 Dec(2)
4 Inc(0)
5 Inc(0)
6 Jump(3,1)
7 Inc(0)

What are $\Psi_{\mathcal{P}}^{(1)}, \Psi_{\mathcal{P}}^{(2)}$ and $\Psi_{\mathcal{P}}^{(3)}$?

Exercise 2.6 (**)**.** Change the program in the previous exercise so that it satisfies the conventions 2.2.2 and 2.2.5.

Exercise 2.7 (**)**.** Prove that for each computable function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ there exist infinitely many programs $\mathcal{P}$ such that $\Psi_{\mathcal{P}}^{(k)} = f$.

Exercise 2.8 (**)**.** Show that if D is a finite set, there exists a program $\mathcal{P}$ such that $\mathcal{P}$ halts exactly on input $x \in D$.

Exercise 2.9 (**)**.** Let $f : \mathbb{N} \longrightarrow \mathbb{N}$ be computable via a program $\mathcal{P}$. We write $\overline{f} : \mathbb{N} \longrightarrow \mathbb{N}$ for the function which is defined by

$$\overline{f}(x) = \begin{cases} x & \text{if } f(x)\downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

The function $\overline{f}$ is sometimes called the *domain* of f. Show that $\overline{f}$ is computable as well by explicitly constructing a program.

Other formulations

Exercise 2.10 (**)**.** Prove that none of the three instructions $\mathsf{Inc}, \mathsf{Dec}, \mathsf{Jump}$ can be defined in terms of the other two.

Exercise 2.11 (***Alternative formalization of register machines)**.** We have allowed for three instructions in register machines. Investigate the following set of instructions and prove or disprove whether it has equal computational strength as our original formalization.

$\mathsf{Inc}(m)$	Increase value of R_m by 1
$\mathsf{Zero}(m)$	Set value of R_m to 0
$\mathsf{Swap}(i,j,k)$	If $R_i = R_j$ then jump to instruction k; otherwise, swap the values of R_i and R_j.

Exercise 2.12 (***Macros)**.** In this exercise we give a precise proof that the macro $\mathsf{Zero}(m)$ can always be expanded.

- Add the instruction $\mathsf{Zero}(m)$ as a basic instruction to the language and adapt the notions of program, state, computation and computable function. (Only indicate the relevant changes.)
- Show for every program in the extended language there is a program in the old language which computes the same function.
- Conclude that both notions of computable function agree.

Exercise 2.13 (**)**.** Is it possible to formulate register machine programs using only two instructions? With only one? (You don't have to prove details.)

Arithmetization and enumeration

Exercise 2.14 (**)**.** Show in detail that the operation $(\mathcal{P}, n) \mapsto Ren(\mathcal{P}, n)$ can be done primitive recursively in codes. This means that there is a PR function $f(e, n)$ which outputs 0 if e is not the code of a program, and outputs the code of the renumbered program $Ren(\mathcal{P}, n)$ if e codes a program $\mathcal{P}$.

Exercise 2.15 (**)**.** Same question for $(\mathcal{P}, n) \mapsto Shift(\mathcal{P}, n)$.

Exercise 2.16 (**)**.** Show that concatenation of programs can be done primitive recursively in codes.

Exercise 2.17 (**)**.** Let us define

$$\phi_e = \begin{cases} \text{the function } \Psi_{\mathcal{P}} \text{ when } \sharp(\mathcal{P}) = e \\ \\ \text{the undefined function otherwise.} \end{cases}$$

Argue that $\phi_0, \phi_1, \ldots$ is an enumeration of the partial recursive functions of one variable, i.e. that every recursive function is of the form ϕ_e for some e.

Exercise 2.18 (***Padding)**.** Referring to the previous exercise, prove that for each e there exists an e' such that $\phi_e = \phi_{e'}$ and $e' > e$. Moreover, prove that there is a primitive recursive function $f(x)$ such that for all e,

$$f(e) > e, \qquad \phi_e = \phi_{f(e)}.$$

(This result is called the Padding Lemma.)

Exercise 2.19 (***)**.** Recall our two conventions 2.2.2 and 2.2.5. Is the predicate

$$Conv(e) \Leftrightarrow \text{ the program with code } e \text{ satisfies both conventions}$$

(primitive) recursive?

Exercise 2.20 (**Busy beavers)**.** Investigate the values of $\Sigma(n)$ for small n.

Chapter 3

Basic recursion theory

The previous two chapters culminated in the proof of the result that the recursive functions are exactly the computable functions. This provides us with a solid foundation and justification for the study of the class of recursive functions. To a certain extent, the work done until now was just preparation! In this chapter our study of recursive functions and recursive sets really begins. The central theme in this chapter, and one of the reasons behind the richness of the theory, is the fact that natural numbers have two faces: on the one hand they are arguments and values of functions, and on the other hand they are codes of recursive functions. It is the interplay between these two which makes the theory fascinating and gives it its unique flavor.

We begin this chapter where we left off: by exploring the consequences of our enumeration of register machine programs and the Normal form theorem for computations. There are two key results on which all of the theory hinges: the Enumeration theorem and the Parameter theorem. In the first section we explain those results; we also briefly address the question to which extent our theory is dependent on the particular enumeration of functions chosen in the previous chapter.

Then we apply these results to prove the famous, but also slightly mysterious Recursion theorems. These are powerful results which have many applications. One of the applications is the fact that the recursive functions are closed under various schemas, such as the double recursion scheme. This gives, in particular, a quick and easy proof that the Ackermann function is recursive. The Recursion theorems also shed light on the special nature of the class of recursive functions, as they illustrate how this class is immune to diagonal arguments. Finally, they will prove to be a useful technical tool for defining functions and sets with specific properties.

Next, we turn to the classes of recursive and recursively enumerable sets (previously, we called those semi-computable, as they are the sets whose semi-characteristic function is computable). We establish the difference between these two classes of sets by introducing the famous Halting problem: this problem, which asks, given a code e and an input x, whether the program with code

e halts on input x, turns out to be undecidable but still recursively enumerable.

We also consider various characterizations of recursively enumerable sets, and various properties which clarify the connections between recursive functions, recursive sets and recursively enumerable sets. Finally, we discuss two important theorems, the existence of recursively inseparable sets and Rice's theorem. The first may be viewed as a generalization of the Halting set to pairs of sets. Rice's theorem gives us the non-recursiveness of an interesting class of sets called index sets. Essentially, such sets describe properties of functions via their codes, and the theorem thus says that properties of functions are non-recursive.

3.1 Enumeration, universality, parametrization

The starting point for the basic theory is the Normal form theorem, proved in the previous chapter. Since we will repeatedly use this result, we restate it here for convenience:

Theorem 3.1.1 (Normal form theorem for computations)**.** *For every recursive function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ there is a code e such that*

$$f(x_1, \ldots, x_k) = U(\mu y.T_k(e, x_1, \ldots, x_k, y)).$$

Here T_k is Kleene's T-predicate in k-variables, and U is the outcome function.

3.1.1 Enumerations of recursive functions

Let us begin by reconsidering our numbering of programs in the previous chapter. To each program we associated a code e. We will now turn this around: given a number e, we will regard it as the code of a recursive function. Thus we define:

Definition 3.1.2. Let $e \in \mathbb{N}$; define the function ϕ_e to be

$$\phi_e(x) = \begin{cases} \Psi_{\mathcal{P}}(x) & \text{if there is a program } \mathcal{P} \text{ with code } e \\ \uparrow & \text{otherwise.} \end{cases}$$

Each of these functions ϕ_e is clearly recursive, since they are either empty or are the functions computed by a program. The sequence

$$\phi_0, \phi_1, \ldots$$

is called an *enumeration* of the (partial) recursive functions (of one variable); indeed, since every recursive function is of the form $\Psi_{\mathcal{P}}$ for a suitable program $\mathcal{P}$, every unary recursive function occurs somewhere in this enumeration. If $f = \phi_e$ then we call e an *index*, or a *code* for f.

Note that we put ϕ_e to be the completely undefined function when $\neg Prog(e)$, i.e. when e is not the code of a program. If we would have devised a bijective coding of programs as number this wouldn't have been necessary.

We may also define enumerations of recursive functions of more than one variable, simply by putting

$$\phi_e^{(n)}(x_1,\ldots,x_n) = \begin{cases} \Psi_{\mathcal{P}}^{(n)}(x_1,\ldots,x_n) & \text{if there is a program } \mathcal{P} \text{ with code } e \\ \uparrow & \text{otherwise.} \end{cases}$$

Often it will be clear what the arity of the function ϕ_e is intended to be, in which case we omit the superscript.

It should be noted that every method of coding programs as numbers gives rise to such an enumeration. Later, we will ask how different such enumerations are related, but for now we first study the important properties of enumerations.

The following result introduces the notion of a universal function, and states their computability. It is also an example of the double function of natural numbers as codes and as inputs.

Theorem 3.1.3 (Enumeration Theorem)**.** *For every $n > 0$ there is a recursive function $\Phi^{(n)}$ of $n+1$ variables with the property that for each recursive function $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ there is a number e such that*

$$\Phi^{(n)}(e, x_1, \ldots, x_n) = f(x_1, \ldots, x_n).$$

The function $\Phi^{(n)}$ in the theorem is called a *universal function* (of n variables). Thus we may reformulate the result by saying that all universal functions are themselves recursive.

Proof. This is an immediate consequence of the Normal form theorem 3.1.1: let $\Phi^{(n)}$ be the function

$$\Phi^{(n)}(e, x_1, \ldots, x_n) = U(\mu y.T_n(e, x_1, \ldots, x_n, y)).$$

□

Since the functions $\Phi^{(n)}$ are recursive, there are programs computing them. A program $\mathcal{P}$ computing $\Phi^{(n)}$ is called a *universal program*. It operates by interpreting the first input variable as the code of a program $\mathcal{Q}$, and by regarding the other inputs as input values for $\mathcal{Q}$. It then simulates the execution of $\mathcal{Q}$ on those input variables and, if this simulation halts, it gives the output $\mathcal{Q}$ would give. Thus, $\mathcal{P}$ acts as a universal interpreter, which can simulate the actions of all programs!

The reader should be warned that in some texts the Enumeration theorem is split in two parts: first the part which states that every recursive function of n variables has a code e (this is the Enumeration part) and then the part which states that the universal functions are recursive (this is then called Universality).

It is convenient in many cases to introduce the following simple notation. Instead of writing $\phi_e(x)$, we write $e \bullet x$, for the e-the function applied to input x. The binary operation $\bullet$ is called *Kleene application* on the natural numbers. The notation is extended to several variables by taking $e \bullet (x_1, \ldots, x_n)$ to mean $\Phi_e^{(n)}(x_1, \ldots, x_n)$. This notation is especially convenient when we wish to avoid nested subscripts, e.g. instead of $\phi_{\phi_e(x)}(y)$ we write $(e \bullet x) \bullet y$.

3.1.2 The parameter theorem

If we have a function $f(x_1, \ldots, x_n, y_1, \ldots, y_m)$ of $n+m$ variables then for each choice of $x_1, \ldots, x_n$ we obtain a function $\tilde{f}_{x_1,\ldots,x_n}$ in the remaining m variables. This process of fixing some of the variables of the function is called *Currying*. Formally, it is an operation

$$[\mathbb{N}^{n+m} \longrightarrow \mathbb{N}] \quad \mapsto \quad [\mathbb{N}^n, [\mathbb{N}^m \longrightarrow \mathbb{N}]]$$

Here, $[A \longrightarrow B]$ denotes the collection of partial functions from A to B, while $[A, B]$ stands for the collection of total functions.

As a simple example, consider the function $f(x, y) = x + y$, regarded as a function of two variables. (Thus, in this example $m = n = 1$.) If we apply the process of Currying to this function we obtain a function $\tilde{f}_x$ of one variable, y. Taking $x = a$, we have that $\tilde{f}_a(y) = f(a, y) = a + y$. In this example, we say that we have regarded x as a parameter and y as a variable in f.

Still considering this example, we may wonder about the following: suppose we are given a code e for the function f, and suppose that we are also given the "parameter" a. How can we find, just in terms of e and a, a code for the Curried function $\tilde{f}_a$? The strategy is to unpack the code e and unravel the program which computes f. Then we build a new program (for computing $\tilde{f}_x$) which does the following: on input y, we copy the input y from register R_1 to register R_2. Then, we place the value a in register R_1. Finally, we run (a suitably renumbered version of) the program with code e.

The above process of taking a code of a program e and a parameter a and turning it into (the code of) a new program which computes the curried version is completely general, in that the numbers e and a are parameters in the construction. Moreover, it is easily verified that the code of the resulting program is a primitive recursive function of those parameters.

The Parameter theorem says exactly this, only generalized to an arbitrary number of parameters and variables. Thus we suppose that we are given a code e of an $n+m$-ary function f, fix the first n parameters of that function and ask for a code of the resulting m-ary function.

Theorem 3.1.4 (Parameter Theorem)**.** *For every $n, m > 0$ there exists a primitive recursive function $S_m^n : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ such that*

$$S_m^n(e, x_1, \ldots, x_n) \bullet (y_1, \ldots, y_m) = e \bullet (x_1, \ldots, x_n, y_1, \ldots, y_m)$$

Proof. The proof is a direct generalization of the informal argument sketched in the example above. We explain first how the function S_m^n should act on input $(e, x_1, \ldots, x_n)$. The result $S_m^n(e, x_1, \ldots, x_n)$ should be the code of a program which does the following (on input $\mathbf{y} = y_1, \ldots, y_m$):

0	$\mathbf{y}$	0	0	$\cdots$	Move inputs
0	$\mathbf{0}$	$\mathbf{y}$	0	$\cdots$	Insert $x_1, \ldots, x_n$
0	$\mathbf{x}$	$\mathbf{y}$	0	$\cdots$	Run program with code e
$e \bullet (\mathbf{x}, \mathbf{y})$	$\mathbf{x}$	$\mathbf{y}$	0	$\cdots$	

Thus the problem is to find, effectively in e and $x_1, \ldots, x_n$, the code of the above program. However, the above program is a concatenation of constant functions (with values) $x_1, \ldots, x_n$) and of e, and its code can be found primitive recursively in those parameters. □

The reader should convince him/herself that the claim made in the last statement of the proof is indeed true. Moreover, with a bit of care, and the knowledge that every recursive function has infinitely many codes, one may in fact find a primitive recursive *injective* function which computes the code of the composite program in terms of $e, x_1, \ldots x_n$.

Many times we will use the Parameter theorem (or the S-m-n theorem or Iteration theorem, as it is also called) by defining a function $f(x)$ by specifying its action on input y. That is, we define $f(x)$ to be a function such that $f(x) \bullet y = h(x, y)$ where the right-hand side is some recursive expression in x, y. What happens is that we let e be a code for h and then put $f(x) = S_1^1(e, x)$.

3.1.3 *Systems of indices

We have introduced an enumeration of the recursive functions and established two key properties, namely the Enumeration theorem and the Parameter theorem. The use of this enumeration will be used heavily in many things to come, so it is pause to stop and ask ourself to which extent the results we proved and will prove depend on the specifics of the enumeration in question.

First, it is clear that our enumeration depends on the choices we have made with regard to our coding of sequences and programs. Second, it is also dependent on the fact that we chose to encode register machine computations, as opposed to, say, Turing machine computations or definitions and computation trees for recursive functions.

Aside from worries about such dependencies one may also have more positive reasons for looking at enumerations from a more abstract point of view. Indeed, we stressed earlier how the enumeration theorem highlights the double role played by the natural numbers; surely we wish to understand what makes this phenomenon possible and what it means for an enumeration of the recursive functions to have these nice properties! These questions have led to a substantial amount of theory on the subject - we only look at the tip of the iceberg here.

A sequence $\psi_0, \psi_1, \ldots$ shall be called an *enumeration of the recursive functions* (of one variable) if $\{\psi_i\}_{i\in\mathbb{N}} = \mathcal{REC}$, i.e. if every unary recursive function occurs somewhere in the enumeration. More generally, a family $\{\psi_i^{(n)}\}_{i\in\mathbb{N},n>0}$ is called a *system of indices* if each $\psi_0^{(n)}, \psi_1^{(n)}, \ldots$ is an enumeration of the recursive functions of n variables. We tend to write simply $\{\psi_i\}_i$ for such a system.

A system $\{\psi_i\}_i$ is said to *satisfy the Enumeration theorem* if for each n there is an i such that

$$\psi_i^{(n+1)}(e, x_1, \ldots, x_n) = \psi_e^{(n)}(x_1, \ldots, x_n).$$

The system is said to *satisfy parametrization* if for each $n, m > 0$ there is a total recursive function s_n^m such that

$$\psi^{(m)}_{s_n^m(e,x_1,\ldots,x_n)}(y_1, \ldots, y_m) = \psi_e^{(n+m)}(x_1, \ldots, x_n, y_1, \ldots, y_m).$$

These are the two desired properties; the following theorem gives a first answer to when they are satisfied.

Theorem 3.1.5 (Rogers)**.** *For a system of indices $\{\psi_i\}_i$, the following are equivalent:*

- *$\{\psi_i\}_i$ satisfies enumeration and parametrization*
- *There are total recursive functions f and g such that for all $n > 0$ and all i we have*
$$\psi_i^{(n)} = \phi_{f(i)}^{(n)}; \qquad \phi_i^{(n)} = \psi_{g(i)}^{(n)}.$$

That is, a necessary and sufficient condition is that one can effectively move back and forth between the indices in the system $\{\psi_i\}_i$ and our standard system $\{\phi_i\}_i$. A system satisfying these conditions is usually called *acceptable*. We will refer to the enumeration $\phi_0, \phi_1, \ldots$ as the *standard system*.

Proof. Let us assume first that $\{\psi_i\}_i$ satisfies enumeration and parametrization. We need to find a total recursive function f such that $\phi_{f(i)}^{(n)} = \psi_i^{(n)}$ for all i. Now since the system satisfies enumeration, the function $(i, x_1, \ldots, x_n) \mapsto \psi_i^{(n)}$ is a recursive function of $n+1$ variables. Hence it has a code in the standard system, say a. Thus we have

$$\psi_i^{(n)}(x_1, \ldots, x_n) = \phi_a^{(n+1)}(i, x_1, \ldots, x_n) = \phi^{(n)}_{S_1^1(a,i)}(x_1, \ldots, x_n).$$

Thus we may let $f(i) = S_1^1(a, i)$. The function g is found similarly.

Conversely, suppose that functions f, g with the stated properties are given. To show enumeration we have to find an index for $\psi_i^{(n)}$ in the system $\{\psi_i\}_i$. But $\psi_i^{(n)} = \phi_{f(i)}^{(n)}$, and the latter is a recursive function of $n+1$ variables, by the

enumeration theorem for the standard system. Thus it is of the form $\psi_j^{(n+1)}$ for some j.

To show parametrization, consider an index a such that

$$\phi_a^{(n+m+1)}(e, \mathbf{x}, \mathbf{y}) = \psi_e^{(n+m)}(\mathbf{x}, \mathbf{y}).$$

This is possible by the enumeration property for $\{\psi_i\}_i$. Now apply the parameter theorem for the standard system and define

$$s_m^n(e, \mathbf{x}) = g(S_m^{n+1}(a, e, \mathbf{x})).$$

Then s_m^n is total recursive and satisfies

$$\psi^{(n)}_{s_m^n(e,\mathbf{x})}(\mathbf{y}) = \psi_{g(S_m^{n+1}(a,e,\mathbf{x}))}(\mathbf{y}) = \phi_{S_m^{n+1}(a,e,\mathbf{x})}(\mathbf{y}) = \phi_a(e, \mathbf{x}, \mathbf{y}) = \psi_e(\mathbf{x}, \mathbf{y}).$$

□

One may now show that acceptable systems of indices share various other properties with the standard system. For example, any such system satisfies the Recursion theorems, to be proved in the next section.

There are various improvements on the above result. One states, that the two functions f and g which relate the two systems may be taken to be bijective and inverse to each other. Thus, in effect, one may say that *any acceptable system of indices is merely a recursive permutation of the standard system.* Another useful consequence is that one can now, without loss of generality assume certain properties of our enumeration. For example, if we want 0 to be a code of the completely undefined function, then we may assume this to be the case, for we just pick a permutation which swaps 0 and a number which actually codes the empty function.

Another result, due to Blum, is the following:

Theorem 3.1.6. *Let $\{\psi_i\}_i$ be an acceptable system of indices. Then there exists a recursive bijective function h such that*

$$\psi_{h(i)}(h(x)) = h(\phi_i(x))$$

for all i, x.

The difference with the earlier result lies in the fact that we now not only transform codes of functions from one system to another, but we also transform the arguments! We may imagine having developed two versions of recursion theory, namely the one based on the standard system, and one based on some alternative system. The result says that these two theories are recursively isomorphic. As long as we agree to be interested in properties which are *recursively invariant* (meaning that they are stable under recursive bijections) this means that the two theories are essentially the same.

The interested reader can find a proof and some related information in Odifreddi's book "Classical recursion theory".

3.2 Recursion theorems and applications

In Chapter 1 we considered an enumeration of primitive recursive functions $f_0, f_1, \ldots$; using a diagonal argument it was then shown that the function $x \mapsto f_x(x) + 1$ could not be primitive recursive. A similar argument applies to total recursive functions: if we had an enumeration $g_0, g_1, \ldots$ of total recursive functions, then we cannot have a universal function which is computable, since that would imply that the function $x \mapsto g_x(x) + 1$ would be total recursive but not in the enumeration.

Given these results, one might wonder why the class of recursive functions escapes this fate. The reason lies in the partiality: when we compare the function $\phi_x(x)+1$ with each the functions ϕ_e we find that we can have $\phi_e(e) = \phi_e(e)+1$ when both are undefined! Thus being undefined on certain elements of the diagonal protects the class of recursive function against diagonal arguments. We are now going to turn this around and use diagonalization in order to define recursive functions with interesting properties.

3.2.1 Kleene's Fixed point theorem

We know that every recursive function has infinitely many codes, corresponding to all the programs which compute it. Can we find a pair of codes for the same function which are related in a special way? For example, could we find a code e and a code e' of the same function such that $e^2 = e'$, or maybe $e' = 2e + 5$? The fixed point theorem guarantees that such codes can indeed be found.

Theorem 3.2.1 (Fixed point theorem)**.** *Let $f(x)$ be a total recursive function. Then there exists an $e \in \mathbb{N}$ such that*

$$\phi_e = \phi_{f(e)}.$$

In this case, e is called a *fixed point* of the function f. However, this is not completely in accordance with common usage of the term "fixed point", which would be a number e such that $f(e) = e$. Of course, many recursive functions, such as the successor function, do not have fixed points in this sense. Rather, what is meant here is that when we regard numbers as codes of functions, then $f(e)$ and e agree. One should also keep in mind that, while we think of f as operating on recursive functions via their codes, it may well happen that i and j code the same function while $f(i)$ and $f(j)$ do not.

Proof. We first define a couple of auxiliary functions. First, let

$$h(u, x) = (u \bullet u) \bullet x$$

Now, using the S-m-n theorem, let $d(u)$ be a total recursive function satisfying

$$d(u) \bullet x = h(u, x).$$

Next, we consider the function fd; this function is also total recursive, and hence has a code, say $fd = \phi_i$. We now diagonalize again, and consider $e = d(i)$:

$$\begin{array}{rcll} e \bullet x & = & d(i) \bullet x & \text{by definition of } e \\ & = & h(i,x) & \text{by definition of } d \\ & = & (i \bullet i) \bullet x & \text{by definition of } h \\ & = & fd(i) \bullet x & \text{since } \phi_i = fd \\ & = & f(e) \bullet x & \text{by definition of } e \end{array}$$

This shows that $\phi_e = \phi_{f(e)}$. □

We note that the proof actually shows that $\phi_e^{(n)} = \phi_{f(e)}^{(n)}$ for any $n > 0$. Sometimes this strengthening is useful.

Let us consider a few illustrations of how one might use this theorem. First, aside from direct applications as mentioned before the theorem, we have a useful corollary:

Corollary 3.2.2. *For any total recursive f there is an e such that $dom(\phi_e) = dom(\phi_{f(e)})$.*

Proof. Pick e such that $\phi_e = \phi_{f(e)}$; in particular, these functions have the same domain. □

It is common practice to write W_e for the domain of the function ϕ_e. Thus, the result gives a code e such that $W_e = W_{f(e)}$. We may now use this to construct domains with certain seemingly self-referential properties.

- There exists an e such that $W_e = \{e\}$. Indeed, define $f(x)$ such that

$$f(x) \bullet y = \begin{cases} x & \text{if } x = y \\ \uparrow & \text{otherwise.} \end{cases}$$

  Thus, we have $W_{f(x)} = \{x\}$. Then choose e such that $W_e = W_{f(e)} = \{e\}$.
- We can find a code i satisfying $W_i = \{0, \ldots, i\}$
- There is a j such that $W_j = \mathbb{N} - \{j\}$
- For some k, we have $W_k = W_{k+1} \cap W_{k+2}$

There are two possible kinds of improvement for the Fixed point theorem. The first states that the "fixed point" e of the function f can be found effectively in a code of f. More precisely:

Theorem 3.2.3 (Fixed point theorem, primitive recursive version)**.** *There is a primitive recursive function m such that for each code i we have*

$$m(i) \bullet x = (i \bullet m(i)) \bullet x$$

for all x.

Proof. Left as exercise to the reader, who should check that the fixed point e found in the proof of the first version is a primitive recursive function of the code of the function f. □

In this formulation, it is important to note that i may not code a total function; if $i \bullet m(i)$ is undefined, then the right-hand side of the equation is undefined, and we ask that $m(i)$ is also the code of an undefined function.

The second generalization introduces parameters in the construction:

Theorem 3.2.4 (Fixed point theorem, parametrized version)**.** *Let $f(x, \mathbf{y})$ be a total recursive function; then there is a total recursive function $e(\mathbf{y})$ such that*

$$\phi_{e(\mathbf{y})} = \phi_{f(e(\mathbf{y}),\mathbf{y})}$$

Proof. Left as exercise to the reader. □

There is also a combination of the two versions, namely a primitive recursive version in parameters.

3.2.2 The Second recursion theorem

We now look at an equivalent formulation of the fixed point theorem. The reason for the appearance of "Second" in the name of the theorem is historical: it appeared after the so-called First recursion theorem in Kleene's work. We will meet the First recursion theorem in a later chapter.

Theorem 3.2.5 (Second recursion theorem)**.** *Let $f(y, \mathbf{x})$ be a recursive function. Then there exists a code e such that*

$$\phi_e(\mathbf{x}) = f(e, \mathbf{x})$$

for all $\mathbf{x}$*.*

Proof. This is easy using the Fixed point theorem: let h be a total recursive function satisfying $h(y) \bullet \mathbf{x} = f(y, \mathbf{x})$ (using the S-m-n theorem as usual) and pick a fixed point e of h, giving

$$\phi_e(\mathbf{x}) = \phi_{h(e)}(\mathbf{x}) = f(e, \mathbf{x}).$$

□

The striking feature of the Second recursion theorem is that it allows us to define recursive functions ϕ_e in terms of their codes. One of the main applications (and indeed the reason for the name of the theorem) is that it allows us to show that the recursive functions are closed under various recursion schemes.

Example 3.2.6. Consider the primitive recursion scheme

$$h(0) = k; \qquad h(n+1) = g(n, h(n)).$$

We may reformulate this as:

$$h(x) = \begin{cases} k & \text{if } x = 0 \\ g(x, h(x-1)) & \text{otherwise.} \end{cases}$$

If h would have a code e it would thus satisfy

$$\phi_e(x) = \begin{cases} k & \text{if } x = 0 \\ g(x, \phi_e(x-1)) & \text{otherwise.} \end{cases}$$

We now let e be a variable in the definition:

$$f(y, x) = \begin{cases} k & \text{if } x = 0 \\ g(x, \phi_y(x-1)) & \text{otherwise.} \end{cases}$$

Clearly, if g is recursive so is f. The Second recursion theorem now tells us that a code e as desired indeed exists, and hence that h is recursive.

The same idea can be applied to other recursion schemes, for example the double recursion scheme. As a consequence, we find that functions defined from recursive functions using double recursion are again recursive; in particular we obtain another proof that the Ackermann function is recursive. In the exercises you will be asked to supply the details.

Again there are two generalizations:

Theorem 3.2.7 (Second recursion theorem, primitive recursive version)**.** *There is a primitive recursive function $n(y)$ such that*

$$n(y) \bullet \mathbf{x} = y \bullet (n(y), \mathbf{x})$$

We leave it as an exercise to the reader to formulate and prove a version with parameters.

We end the discussion of the Recursion theorems with a few observations. First of all, when we consider the equation

$$\phi_e(x) = f(e, x),$$

as a definition of the code e, it seems as if we are committing the old sin of circularity. However, the point is that one should really view the function with code e as the result of a diagonalization argument without the usual punchline! Indeed, in the special case where $f(y, x) = \phi_y(y) + 1$, we will obtain a code e such that $\phi_e(x) = \phi_e(e) + 1$, and we escape the conclusion from the diagonal argument by having $\phi_e(e)$ undefined.

Second, it should be noted that there are in fact infinitely many codes satisfying the equation from the recursion theorem. This is a direct consequence of the fact that every function has infinitely many codes, and the fact that the S-n-m functions, which are used to construct the sought after code, are injective.

Third, the reader who is acquainted with the lambda calculus will perhaps suspect a connection with fixed point combinators. There is indeed a clear analogy, which is worked out in detail in Odifreddi's book.

3.3 Recursive sets and r.e. sets

In the introduction we discussed a number of problems which were not decidable, but for which the strategy of generating possible solutions in a systematic way and testing them one by one will always find a positive answer if it exists. In view of the register machine model, these sets also appear very naturally: for any program we may ask on which inputs the program will halt; given an input we can, in general, not decide whether the program will halt on that input, but we can always run the program and see what happens. If the program comes to a halt then we know the positive answer to our question. However, we will never gain information towards a negative answer.

We now set out to study such sets in more detail. We start by giving a number examples. One of those is the Halting set, which is the paradigmatic example of a set which is recursively enumerable but not recursive. We also offer a number of different viewpoints of recursively enumerability by providing some alternative characterizations. These also explain how these sets are related to recursive sets and to recursive functions and help us understand the relevant properties. Finally, we present the important undecidability and inseparability results, including Rice's theorem. These provide us with a better understanding of why certain problems are undecidable.

3.3.1 Recursively enumerable sets

Let us begin by stating one of the central definitions:

Definition 3.3.1 (Recursively enumerable set)**.** A set $A \subseteq \mathbb{N}^k$ is called *recursively enumerable* if there is a recursive function $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ such that $A = dom(f)$, i.e.

$$(x_1, \ldots, x_k) \in A \Leftrightarrow f(x_1, \ldots, x_k)\downarrow$$

We usually abbreviate "recursively enumerable" to "r.e."; other names encountered in the literature include: semi-recursive set, semi-computable set, semi-decidable set, recursively generated set, computably generated set. In fact, there is a fierce debate on what the correct terminology is, not just for r.e. sets, but also for the entire subject.

Let us look at a few examples. Note that, in order to show that a given set is r.e., it suffices to construct a program which accepts precisely the inputs in that set.

Examples 3.3.2.

1. Any finite set $A = \{a_1, \ldots, a_k\}$ is r.e.: to prove this, consider the program which compares the input x with all the elements a_i. If it is equal to one of them, it will halt. If not, it goes into a loop.
2. Any cofinite set is r.e. (a set is cofinite if its complement is finite).
3. The set of codes e such that the function ϕ_e is defined on input 12 is an r.e. set: consider the function $e \mapsto \phi_e(12)$. By the enumeration theorem this function is recursive in e, and its domain is the set $\{e|\phi_e(12)\downarrow\}$.
4. Let $F(x, x_1, \ldots, x_k), G(x, x_1, \ldots, x_k)$ be polynomials with coefficients in $\mathbb{N}$. Consider the set

$$A = \{x|\exists x_1 \ldots x_k.F(x, x_1, \ldots, x_k) = G(x, x_1, \ldots, x_k)\},$$

   This set is r.e., as witnessed by the program which, on input x, tries all possible value $x_1, \ldots, x_k$ in a systematic manner, computes $F(x, x_1, \ldots, x_k)$ and $G(x, x_1, \ldots, x_k)$ and tests if they are equal. If so, the program halts. If not, the program continues trying values, etc. Sets of this form are called *Diophantine*.

The first two examples are actually a special case of the following simple lemma, which states that the r.e. sets include all recursive sets.

Lemma 3.3.3. *Every recursive set is recursively enumerable.*

Proof. If A is recursive, then by definition the characteristic function χ_A is recursive. Now define the function

$$\psi_A(x) = \begin{cases} 1 & \text{if } \chi_A(x) = 1 \\ \uparrow & \text{otherwise.} \end{cases}$$

This function (which we earlier called the semicharacteristic function of A) is then also recursive and satisfies $dom(\psi_A) = A$. □

The last example given above is a famous one: in 1900, Hilbert presented his list of 23 problems. Number 10 on that list asked for a systematic procedure which would determine the existence of solutions of Diophantine equations with integer coefficients. The importance of this problem becomes clear when we note that, as a special case, we have equations of the form

$$(x+1)^n + (y+1)^n = (z+1)^n.$$

The statement that for $n > 2$ this equation has no solutions (in $\mathbb{N}$) is known as Fermat's last theorem; it took three centuries before a rigorous proof was given. In 1970, Yuri Matiyasevich proved that the procedure Hilbert had requested could not exist. Indeed, he showed the following theorem:

Theorem 3.3.4. *Every r.e. set is Diophantine.*

This is a representation theorem, which establishes a connection between computability theory and number theory. Explicitly, it says that for each r.e. set $A \subseteq \mathbb{N}$ we can find polynomials $F(x, \mathbf{y})$ and $G(x, \mathbf{y})$ such that

$$x \in A \Leftrightarrow \exists \mathbf{y}.F(x, \mathbf{y}) = G(x, \mathbf{y}).$$

If we allow integer coefficients in our polynomials then it suffices to consider only one polynomial. Moreover, it is known that one can without loss of generality consider polynomials of less than 11 variables. The proof of this remarkable theorem is outside the scope of these notes. Since, as we shall shortly see, there are r.e. sets which are not recursive, an algorithm for determining whether there exist solutions to a Diophantine equation cannot exist, and Hilbert's 10th problem cannot have a positive answer.

The Halting problem. Since every computable function has a code, we may describe r.e. sets by referring to these codes. More formally, we define

$$W_e = \{x \in \mathbb{N} | \phi_e(x)\downarrow\}.$$

Thus, the sequence $W_0, W_1, \ldots$, is an enumeration of the r.e. sets. Just as with our enumeration of recursive functions, this enumeration has repetitions: every r.e. set occurs infinitely many times.

Now given two numbers e, x, we may ask whether $x \in W_e$. This is known as the *Halting problem*, since it amounts to asking whether the function/program with code e halts on input x. Put

$$K_0 = \{(e, x) | \phi_e(x)\downarrow\} \subseteq \mathbb{N}^2.$$

This set is r.e.; indeed, it is the domain of the universal function

$$\Phi(e, x) = \phi_e(x).$$

Similarly, one considers the set $\{[e, x] | \phi_e(x)\downarrow\} \subseteq \mathbb{N}$, which is r.e. for much of the same reason.

We may also consider a simpler version, called the *standard problem*: put

$$K = \{x \in \mathbb{N} | \phi_x(x)\downarrow\}.$$

Clearly, K is r.e. as well. Can K be recursive? Well, let's suppose it is. Then the complement $\mathbb{N} - K$ must also be recursive. But by Lemma 3.3.3, that

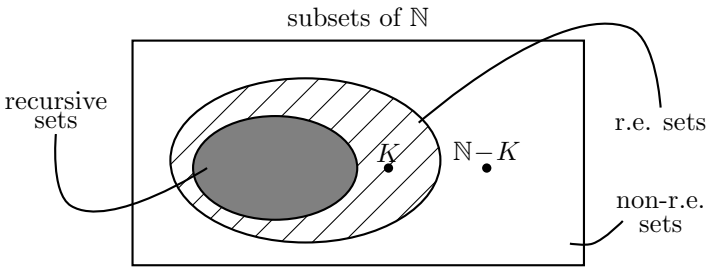


Figure 3.1: Recursive sets and r.e. sets and the Halting set

implies that $\mathbb{N} - K$ is r.e., and thus it is of the form W_i for some i. Note that this set has the property that $x \in W_i \Leftrightarrow x \notin K$.

Now we ask: does i belong to K or to its complement $\mathbb{N} - K$? Let's try:

$$\begin{aligned} i \in K &\Leftrightarrow i \in W_i && \text{by definition of } K \\ &\Leftrightarrow i \notin K && \text{by definition of } W_i \end{aligned}$$

Contradiction. The assumption that K was recursive must be incorrect. We have:

Proposition 3.3.5 (Undecidability of the Halting problem)**.** *The sets* $K = \{x|\phi_x(x)\downarrow\}$ *and* $K_0 = \{(e,x)|\phi_e(x)\downarrow\}$ *both are r.e. but not recursive.*

Proof. We have shown this for K; but $\chi_K(x) = \chi_{K_0}(x,x)$, so a solution to the halting problem would give a solution to the standard problem. □

A different proof for the undecidability of K is suggested in the exercises. Figure 3.1 schematically represents what we know so far about classes of natural numbers.

3.3.2 Characterizations of r.e. sets

We have defined an r.e. set to be a set which is the domain of some recursive function. However, we shall now investigate a few characterizations of r.e. sets which are often useful. The results in this section also highlight further connections between recursive functions, recursive sets and r.e. sets. The first of these is:

Theorem 3.3.6 (Normal form theorem for r.e. sets)**.** *For every r.e. set there exists an* $e \in \mathbb{N}$ *such that* $A = \{x|\exists y.T(e,x,y)\}$.

Proof. This is immediate from the Normal form theorem for computations. □

In particular, this shows that every r.e. set is obtained by existentially quantifying over a primitive recursive set. The result readily extends to r.e. relations of more variables.

However, more is true: the projection of an r.e. relation is again r.e.; indeed, if $R(\mathbf{x}, y)$ is r.e. and we define $A(\mathbf{x}) = \exists y.R(\mathbf{x}, y)$, then we have, for some recursive $S(\mathbf{x}, y, z)$:

$$A(\mathbf{x}) \Leftrightarrow \exists y.R(\mathbf{x}, y) \Leftrightarrow \exists y \exists z.S(\mathbf{x}, y, z) \Leftrightarrow \exists u.S(\mathbf{x}, (u)_1, (u)_2).$$

This shows:

Lemma 3.3.7 (Projection lemma)**.** *The class of r.e. sets are closed under existential quantification.*

The method used in the proof is important: it combines the search for two numbers y, z into the search for a single number which we view as a pair.

If $R(x, y)$ is a relation, then we can always consider functions f such that $Grp(f) \subseteq R$; a function f which in addition satisfies $\exists y.R(x, y) \Leftrightarrow f(x)\downarrow$ is called a *selector function* for R. Set-theoretically, one can (using the axiom of choice) always find a selector function. The following theorem (sometimes called the Uniformization Theorem) says that any r.e. relation contains a *recursive* selector function.

Theorem 3.3.8 (Uniformization/Selection theorem)**.** *Let $R(x, y)$ be an r.e. relation. There exists a recursive function f such that*

$$\exists y.R(x, y) \Leftrightarrow f(x)\downarrow \wedge R(x, f(x)).$$

Proof. Suppose that $R(x, y) \Leftrightarrow \exists z.S(x, y, z)$ for a recursive relation S. Given x, we now need to find y, z such that $S(x, y, z)$, and then we may let $f(x) = y$. We interleave the search for such y, z, by putting

$$h(x) = \mu v.S(x, (v)_1, (v)_2).$$

It now suffices to let $f(x) = (h(x))_1$. □

This result shows how the structured search given by the μ-operator is related to the unordered search arising from existential quantification. The following is an easy consequence.

Lemma 3.3.9. *Let f be a function. Then f is recursive if and only if $Gr(f)$, the graph of f, is recursively enumerable.*

Proof. If f is recursive, then consider

$$g(x,y) = \begin{cases} 1 & \text{if } f(x) = y \\ \uparrow & \text{otherwise.} \end{cases}$$

Then g is recursive and $dom(g) = Gr(f)$.

Conversely, f is the unique selector function for $Gr(f)$, so by the Uniformization theorem f is recursive if $Gr(f)$ is r.e. □

The next result may be regarded as a justification for the terminology "recursively enumerable", because it says that every r.e. set may be enumerated by a recursive function.

Lemma 3.3.10. *Every nonempty r.e. set is the range of a primitive recursive function. Moreover, the range of any recursive function is r.e.*

Proof. For the first part, let $A = dom(\phi_e)$ be r.e.; we have to find g such that $rng(g) = A$. We shall actually show that we may take g to be primitive recursive, unless $A = \emptyset$, in which case we can let g be the empty function. So assume that $A \neq \emptyset$, and pick $n_0 \in A$. Then let

$$g(x,t) = \begin{cases} x & \text{if } \exists s \leq t.T(e,x,s) \\ n_0 & \text{otherwise.} \end{cases}$$

For the second part, if f is recursive and $B = rng(f)$, let e be a code for f and consider the function

$$h(x) = \mu z.(T(e,(z)_0,(z)_1) \wedge U((z)_1) = x.$$

Now if $x \in rng(f)$, then there is x' such that $f(x') = x$. In particular, there is a y such that $T(e,x',y) \wedge U(y) = x$. Thus $h(x)\downarrow$. On the other hand, if $x \notin rng(f)$, such x', y don't exist and $h(x)$ diverges as required. □

It is worth noting that the above result gives rise to another enumeration of the collection of r.e. sets. Define

$$R_e = \{x | \exists y.\phi_e(y) = x\}.$$

Then $R_0, R_1, \ldots$ is also an enumeration of the r.e. sets. A connection between the enumerations $\{W_e\}_{e\in\mathbb{N}}$ and $\{R_e\}_{e\in\mathbb{N}}$ will be explored in the exercises.

Finally, we remark that if $A = rng(f)$ and f recursive, this need not imply that the set A is enumerated in increasing order. Thus, we may have, say, $f(0) > f(1)$. Moreover, one cannot, in general, fix this, as is shown by the following lemma:

Lemma 3.3.11. *A nonempty set $A \subseteq \mathbb{N}$ is recursive if and only if it is the range of a monotone total recursive function.*

Here, by monotone we mean that $x \leq y \Rightarrow f(x) \leq f(y)$.

Proof. First, let a be the smallest element of a. Define:

$$\begin{aligned} f(0) &= a \\ f(n+1) &= \begin{cases} n+1 & \text{if } n+1 \in A \\ f(n) & \text{otherwise.} \end{cases} \end{aligned}$$

Then f is recursive, monotone and $rng(f) = A$.

Conversely, let A be the range of f, where f is monotone total recursive. If A happens to be finite, we are done since any finite set is recursive. Otherwise, A is infinite, and that means that the function

$$h(x) = \mu y.x < f(y)$$

is total recursive. Now consider

$$x \in A \Leftrightarrow \exists y \leq h(x).f(y) = x.$$

This shows that A is recursive. □

Before we explore more properties of r.e. sets, we make an observation: many of the above results are concerned with presentations of r.e. sets. Often it is not just important to know, for example, that a given r.e. set may be presented as the projection of some recursive set, but one wants to obtain a code for that recursive set in terms of a code for the original r.e. set. Moreover, one wants to obtain this code in a (primitive) recursive manner from the original code. In the exercises we explore these strengthened results, which are typically proved by observing that an expression for the new code may be found in terms of the old code, and then applying the parameter theorem.

In general, it is good practice to always ask whether certain results or constructions are effective, i.e. if the can be performed in a recursive manner on the codes of the functions or sets involved.

3.3.3 Lattice-theoretic properties

Having explored some connections between recursive functions, recursive sets and r.e. sets, we now have a brief look into the properties of the partially ordered set of r.e. sets. We shall write $\mathfrak{RE}$ for this poset, where the ordering is given by inclusion.

As a first, trivial observation, we note that $\mathfrak{RE}$ has a least element, $\emptyset$, and a greatest element, $\mathbb{N}$. Next, we note that $\mathfrak{RE}$ has binary meets: given two r.e. sets $A = W_e, B = W_f$, their intersection $A \cap B$ is the domain of the function

$$h(x) = \phi_e(x) + \phi_f(x).$$

Clearly, $h(x)\downarrow$ if and only if both $\phi_e(x)\downarrow$ and $\phi_f(x)\downarrow$, i.e. if and only if $x \in W_e$ and $x \in W_f$.

The collection of r.e. sets is also closed under binary unions: again let $A = W_e, B = W_f$, and consider

$$k(x) = \mu y.(T(e, x, y) \vee T(f, x, y)).$$

Now we have that $k(x)\downarrow$ if and only if one of $\phi_e(x), \phi_f(x)$ is defined, so $x \in W_e$ or $x \in W_f$.

The technique of combining two computations into one by simultaneously considering the approximations to each of them is called *dovetailing*; we will use it many more times below. It helps ensure that we will not get lost when one of the two computations happens to be non-terminating.

We have shown:

Proposition 3.3.12. *The collection $\mathfrak{RE}$ of r.e. sets is a lattice under set-theoretic inclusion. It contains the lattice of recursive sets as a sublattice.*

The following result is known as Post's Theorem. It characterizes the complemented elements in the lattice $\mathfrak{RE}$.

Theorem 3.3.13 (Complementation theorem)**.** *A set A is recursive if and only if both A and $\mathbb{N} - A$ are r.e.*

Proof. One direction is easy: if A is recursive then by Lemma 3.3.3 it is also r.e., and the same will hold for $\mathbb{N} - A$ which is also recursive. In the other direction, suppose we have

$$W_e = A, \qquad W_f = \mathbb{N} - A.$$

Given x, we need to decide whether $x \in A$ or $x \in \mathbb{N} - A$. We know that either $\phi_e(x)\downarrow$ or $\phi_f(x)\downarrow$, but we don't know which. Therefore we alternate the computation steps of $\phi_e(x)$ and $\phi_f(x)$ until one of them holds: put again

$$k(x) = \mu y.(T(e, x, y) \vee T(f, x, y))$$

and then $\chi_A(x)$ may be expressed as

$$\chi_A(x) = \begin{cases} 1 & \text{if } T(e, x, k(x)) \\ 0 & \text{otherwise.} \end{cases}$$

□

Various other notions may be expressed in the language of lattices. For example, an element $A \in \mathfrak{RE}$ is finite if and only if every $A' \subseteq A$ is complemented. We sketch an argument, leaving the details to the reader. Suppose A were infinite but every $A' \subseteq A$ were complemented, i.e. recursive. In particular A is recursive. Pick a total recursive injective function f such that $rng(f) = A$ (this is possible by the results in the previous section). Consider

$f[K] \subseteq A$, the image under f of the halting set. This is a subset of A and hence complemented. But now we can deduce that K itself is complemented. Contradiction.

We may wonder if the lattices $\mathfrak{Rec}$ and $\mathfrak{RE}$ have any more structure. For example, do any infinite suprema exist? It is easy to see that $\mathfrak{RE}$ is not closed under arbitrary suprema: any set is the union of its singleton subsets, each of which is of course recursive. However, the lattice $\mathfrak{RE}$ is closed under r.e. indexed unions, in the sense that for each r.e. set A, the union $\bigcup_{a\in A} W_a$ is r.e. as well. To prove this, we interleave the search for an element $a \in A$ with the computations of $\phi_a(x)$. Explicitly, define

$$f(x) = \mu z.(T(e, (z)_1, (z)_2) \wedge T((z)_1, x, (z)_3)).$$

Now it is readily verified that $f(x)\downarrow$ if and only if then there exists $z = [a, y, y']$ such that $T(e, a, y)$ and $T(a, x, y')$, i.e. $a \in W_e$ and $x \in W_a$.

As a last result in this section, we present a procedure for cutting down the union of two r.e. sets to the union of two disjoint r.e. sets.

Proposition 3.3.14 (Reduction)**.** *Let A, B be two r.e. sets. Then there exist r.e. sets $A' \subseteq A, B' \subseteq B$ such that $A' \cup B' = A \cup B$, and $A' \cap B' = \emptyset$.*

Proof. It is clear that A' should contain at least the elements of $A - B$, and similarly that B' should contain $B - A$; the problem is to decide where the elements in the intersection $A \cap B$ should go. Let $A = W_e, B = W_f$, and define

$$\begin{aligned} x \in A' &\Leftrightarrow \exists y.(T(e, x, y) \wedge \forall z \leq y.\neg T(f, x, z)) \\ x \in B' &\Leftrightarrow \exists y.(T(f, x, y) \wedge \forall z \leq y.\neg T(e, x, z)). \end{aligned}$$

Then clearly $A' \cap B'$ are disjoint and r.e., and their union is equal to $A \cup B$. □

While the lattice-theoretic structure of the recursive sets is completely understood (see Odifreddi for a characterization), the structure of $\mathfrak{RE}$ is more challenging. Some more aspects of this structure will be explored in the exercises. In a later chapter we will investigate $\mathfrak{RE}$ from a topological viewpoint; this leads into the subject called *domain theory*.

3.3.4 Undecidability and inseparability results

Earlier we considered the Halting set, a standard example of a set which is r.e. but not recursive. A result of this kind is called an *undecidability result*, since it tells us that certain problems are not decidable; this guarantees the theory of r.e. sets to be non-trivial. In this section we consider first a strengthening of this result by extending it to pairs of disjoint r.e. sets. We then consider Rice's Theorem, which gives us a large supply of undecidable problems.

Recall that we showed that a set A is recursive if and only if both A and its complement are r.e. This suggests considering pairs of disjoint r.e. sets.

Given a pair A, B of r.e. sets such that $A \cap B = \emptyset$, we may wonder whether they can be *recursively separated*: this means that we look for a third set C such that $A \subseteq C$ and $C \cap B = \emptyset$. Of course, we require C to be recursive. If A and B are complementary, then we know that A is recursive and we may take $C = A$; however, if A and B are not recursive then we cannot, in general, separate them recursively.

Theorem 3.3.15 (Existence of inseparable r.e. sets)**.** *There exist disjoint r.e. sets A, B such that A and B are recursively inseparable.*

Proof. Define A to be the set $\{x|\phi_x(x) = 0\}$ and $B = \{x|\phi_x(x) = 1\}$. Clearly A, B are r.e. and $A \cap B = \emptyset$. Now suppose that C is recursive and $A \subseteq C$, $C \cap B = \emptyset$. Define the function

$$h(x) = \begin{cases} 1 & \text{if } x \in C \\ 0 & \text{otherwise} \end{cases}$$

and consider a code e for h. Now we have

$$e \in C \Leftrightarrow h(e) = 1 \Leftrightarrow \phi_e(e) = 1 \Leftrightarrow e \in B,$$

which is a contradiction since C and B are disjoint. Similarly we find

$$e \notin C \Leftrightarrow h(e) = 0 \Leftrightarrow \phi_e(e) = 0 \Leftrightarrow e \in A,$$

which is again a contradiction since $A \subseteq C$. □

Inseparable sets play an important role in the study of logical systems. In many cases, the sets of theorems (provable sentences) and of refutable sentences are inseparable by recursive means.

We now describe a family of unsolvable problems. It is convenient to introduce some terminology first:

Definition 3.3.16 (Index set)**.** A set $A \subseteq \mathbb{N}$ is called an *index set* if there is a collection F of recursive functions such that

$$e \in A \Leftrightarrow \phi_e \in F.$$

One may reformulate the definition by saying that an index set has the property that if one code for a recursive function is in the set, so are all other codes of that function. Thus, an index set really describes a collection of recursive functions via their codes. In some texts, index sets are called *extenstional sets*.

Examples 3.3.17. The following are index sets:

- $\mathbb{N}$ is an index set, $\emptyset$ is an index set (these are called *trivial*)
- $\mathsf{Tot} = \{e | \phi_e \text{ is a total function}\}$
- $\mathsf{Fin} = \{e | \phi_e \text{ has finite domain}\}$
- $\{e | \exists x.\phi_e(x)\downarrow\}$
- $\{e | \exists x.\phi_e(x)\uparrow\}$.

By contrast, the following are not index sets:

- The set of even numbers (one can find two programs which compute the same function, one having a prime code and the other having a non-prime code).
- Any non-empty finite set (because every function has infinitely many codes)
- $\{e|$the program with code e has more than 12 instructions$\}$.
- K, the standard problem. One possible way to see this is as follows: take a code e such that $W_e = \{e\}$ (using the Fixed point theorem). Since $e \in W_e$, we have $\phi_e(e)\downarrow$, and hence $e \in K$. Now consider the program coded by e and add to that program increasing the length of the program so that the code of this new program will be different from e, while not changing its domain. The new program computes the same function, but its code is not in K.

It is easy to show (exercise for the reader) that a set A is an index set if and only if its complement is an index set. We may now show that index sets (which, in some texts, are called extensional sets) are never recursive, except when they are trivial.

Theorem 3.3.18 (Rice's Theorem)**.** *Let A be an index set such that $A \neq \emptyset$ and $A \neq \mathbb{N}$. Then A is not recursive.*

Proof. Our proof makes use of the Second Recursion theorem. Towards a contradiction, suppose that the set A is recursive. Because A is non-trivial, we may pick a code $e \in A$, and another code $f \notin A$. Now define, using the recursion theorem, a code a such that

$$a \bullet x = \begin{cases} f \bullet x & \text{if } a \in A \\ e \bullet x & \text{otherwise.} \end{cases}$$

Because A is an index set, we now see that

$$a \in A \Rightarrow \phi_a = \phi_f \Rightarrow a \notin A$$

and

$$a \notin A \Rightarrow \phi_a = \phi_e \Rightarrow a \in A.$$

Contradiction, and hence A cannot be recursive. □

This shows at once that the index sets from the examples are not recursive. Be careful: nothing follows about whether these sets are r.e. or not! For example, the set $\{e|\exists x.\phi_e(x)\downarrow\}$ is r.e. because it is the projection of a recursive set, but the sets Tot and Fin are not; in the next chapter we will introduce techniques for establishing these facts.

One often summarizes Rice's theorem by saying that non-trivial properties of recursive functions are undecidable.

3.4 Exercises

Enumerations and indices

Exercise 3.1 (*)**.** Explain in detail why the Enumeration theorem fails for the class of total recursive functions.

Exercise 3.2 (*)**.** Can there be an Enumeration theorem for the recursive sets?

Exercise 3.3 (**Parameter theorem)**.** Show that the Parameter theorem already follows from the special instance $n = m = 1$.

Exercise 3.4 (**Currying of partial functions)**.** In this exercise, let A, B, C be arbitrary sets.

- Show that there is a bijective correspondence between the sets $A^{B\times C}$ and $(A^B)^C$, and that this generalizes the Currying operation described in the text. Here, X^Y stands for the collection of all total functions $X \longrightarrow Y$.
- Investigate what happens when we replace total functions by partial functions. What is the correct operation of Currying for partial functions?

Exercise 3.5 (*)**.** Show that the collection of recursive bijections forms a group under composition.

Exercise 3.6 (***)**.** Show that the primitive recursive bijections do not form a group under composition.

Exercise 3.7 (**)**.** Let $\phi_0, \phi_1, \ldots$ and $\psi_0, \psi_1, \ldots$ be two acceptable systems of indices for the recursive functions.

1. Explain that each of them gives rise to a (partial) binary application operation, just as the standard system gives rise to the Kleene application.

2. Writing $*$ for the first application (induced by ϕ) and $\diamond$ for the second (induced by ψ), show that there exist elements u, v such that for all x, y we have

$$(u * x) * y = x \diamond y \qquad \text{and} \qquad (v \diamond x) \diamond y = x * y.$$

Recursion theorems

Exercise 3.8 (**)**.** Prove the primitive recursive version of the Fixed point theorem.

Exercise 3.9 (**)**.** State and prove a version of the Second Recursion theorem with parameters.

Exercise 3.10 (*)**.** Show that there is a code e such that ϕ_e is the constant function with value e.

Exercise 3.11 (**)**.** Show that there is a code e with the property that ϕ_e is defined on input x whenever x is a multiple of e.

Exercise 3.12 (**)**.** Show that there is a code e such that $(e \bullet x) \bullet y = e$ for all x, y.

Exercise 3.13 (**)**.** Show that choices of codes e, f, g can be made such that

- $W_e = \{0, \ldots, e\}$
- $W_f = \mathbb{N} - \{f\}$
- $W_g = W_{g+1} \cap W_{g+2}$

Exercise 3.14 (**)**.** Show directly that the Fixed point theorem follows from the Second Recursion theorem.

Exercise 3.15 (***)**.** Prove Smullyan's Double Recursion theorem: given recursive functions $f(x, y)$ and $g(x, y)$ there are codes a, b such that $\phi_a = \phi_{f(a,b)}$ and $\phi_b = \phi_{g(a,b)}$.

Exercise 3.16 (**)**.** Show that the recursive functions are closed under the double recursion scheme. Hint: mimic the proof for primitive recursion.

R.e. sets

Exercise 3.17 (*). Show that a set $A \subseteq \mathbb{N}^2$ is recursive if and only if the set $\{p(x,y)|(x,y) \in A\}$ (the image of A under the binary pairing operation) is recursive. Same question for A an r.e. set.

Exercise 3.18 (*Another proof that K is not recursive). Give an alternative proof that the set $\{x|\phi_x(x)\downarrow\}$ is not recursive, by considering the function

$$f(x) = \begin{cases} \phi_x(x) + 1 & \text{if } x \in K \\ 0 & \text{otherwise.} \end{cases}$$

Exercise 3.19 (*Index sets). Determine whether the following sets are index sets or not.

- $\{e|\neg Prog(e)\}$
- $\{e|Prog(e) \wedge e > 623\}$
- $\{e|e$ codes a program without jump-instructions$\}$
- $\{e|e$ codes a program which only halts on prime numbers$\}$
- $\{e|e$ codes a program with less than 7 instructions$\}$
- $\{e|e$ codes a program which accepts infinitely many inputs$\}$
- $\{[e,f]|\phi_e = \phi_f\}$

Exercise 3.20 (*Index sets continued). Determine which of the sets in the previous exercise are recursive.

Exercise 3.21 (*). Is the set $\{e|W_e$ is recursive $\}$ recursive?

Exercise 3.22 (**). Investigate whether the following sets are r.e. or not.

- $\mathbb{N} - K = \{x|\phi_x(x)\uparrow\}$
- $\{e|e \bullet x\uparrow \text{ for all } x\}$
- $\mathsf{Tot} = \{e|\phi_e$ is a total function$\}$.

Aspects of r.e. sets and recursive functions

Exercise 3.23 (**Pullbacks). Let $f : \mathbb{N} \longrightarrow \mathbb{N}$ be recursive, and let $A \subseteq \mathbb{N}$ be a set. Define $f^{-1}(A) = \{x|f(x)\downarrow \,\& f(x) \in A\}$. Suppose that A is r.e.; does it follow that $f^{-1}(A)$ is r.e. as well? Same question for A recursive.

Exercise 3.24 (**Images)**.** Let $f : \mathbb{N} \longrightarrow \mathbb{N}$ be recursive, and let $A \subseteq \mathbb{N}$ be a set. Define $f[A] = \{f(x) | f(x)\downarrow \ \& x \in A\}$. Suppose that A is r.e.; does it follow that $f[A]$ is also r.e.? Same question for A recursive.

Exercise 3.25 (**)**. Let f be a recursive function. Show that f has a *partial section*: this is a recursive function g with the properties that

- $rng(f) = dom(g)$
- for all $x \in dom(g)$: $fg(x) = x$.

Exercise 3.26 (***)**. Let A be an infinite set.

1. Show that A is r.e if and only if A is the range of some injective total recursive function.
2. Show that A is recursive if and only if A is the range of an injective, *monotone*, partial recursive function.
3. Conclude by describing the difference between recursive and r.e. sets in terms of possible ways of enumerating them.

Exercise 3.27 (***)**. Show that every infinite r.e. set contains an infinite recursive subset.

Exercise 3.28 (*)**. Show that there are primitive recursive functions f, g such that

$$W_{f(a,b)} = W_a \cap W_b; \qquad W_{g(a,b)} = W_a \cup W_b.$$

Exercise 3.29 (*)**. Recall that we write R_e for the range of the function ϕ_e. Show that there are primitive recursive functions f, g such that

$$W_a = R_{f(a)}; \qquad R_b = W_{g(b)}.$$

Exercise 3.30 (**)**. Investigate which other characterizations of r.e. sets can be made primitive recursive in codes.

Chapter 4

Classifications of unsolvable problems

We have encountered quite a few unsolvable problems now: first, there is the Halting problem, obtained by diagonalization. Next, we saw how Rice's theorem gives us at once a large supply of non-recursive sets: essentially, any problem which concerns properties of recursive functions is undecidable. A large part of recursion theory is concerned with techniques for comparing and classifiying such unsolvable problems, and now is the time for us to make the first steps into this intricate area.

We begin by introducting the notion of a *reducibility relation*: this is a relation on subsets of the natural numbers (or possibly on functions on the natural numbers) which, in one way or another, expresses that one set is at least as complicated as the other. Such a relation organizes the collection of all subsets of the natural numbers into *degrees*, i.e. sets which have the same level of difficulty. There is a whole spectrum of such reducibility relations: each of them gives a different criterion for what it means to be "at least as difficult".

We look at two reducibility relations in some detail: m-reducibility and T-reducibility. In a certain sense, these two form the extremes of the spectrum of so-called "strong reducibilities". M-reducibility is based on the transformation of problems via computable functions. After considering some basic properties and examples, we shall study *m-complete sets*; these are r.e. sets which are as difficult as possible, in the sense that any other r.e. set m-reduces to it. Various interesting characterizations of such sets will be given, culminating in Myhill's theorem which states that each of these sets are recursively isomorphic.

For any reducibility we can ask if there exist more than two degrees of r.e. sets. For m-reducibility, this means: are there sets which are not recursive and also not m-complete? Such sets indeed exist, and are called *simple sets*. We will look at a couple of examples of such sets. In particular, we will build such a set employing what is arguably the most important technique in recursion theory: the priority argument.

T-reducibility is a weaker reducibility based on the notion of *relative computability*, or *oracle computability* as it is also called. The idea is that an oracle for a set B is an external entity (deity) which we can ask questions to about the

set B during a computation. The sets which are computable with the aid of the oracle for B are then said to be B-recursive. To say that A is B-recursive expresses in some way that A is at most as difficult as B. Almost all of the theory developed so far (about recursive functions and r.e. sets) can be generalized to relative computability.

We will have a first glance at the highly complicated structure of *T-degrees* in the last sections of this chapter. One of the concepts which help us understand this structure is the *Jump operation*: this relativizes the construction of the Halting set, and provides us with a way of building sets of higher and higher complexity. It also helps understand the connections between the various notions we study in this chapter.

Finally, we have a look at the *Arithmetical hierarchy*, which helps us understand the complexity of various problems in terms of nestings of quantifiers. The *Tarski-Kuratowski algorithm* helps us to classify problems in this hierarchy, and we will also see that Post's Theorem creates a link between this hierarchy and relative computability.

4.1 Many-one reduction

We begin by introducing a number of basic concepts which will allow us to organize the collection of unsolvable problems.

4.1.1 Reducibilities and degrees

Definition 4.1.1 (Reducibility relation)**.** A *reducibility relation* $\leq_r$ is a preorder on $\mathcal{P}(\mathbb{N})$ (i.e. a relation which is transitive and reflexive but not necessarily antisymmetric). If $A \leq_r B$ and $B \leq_r A$ then we write $A \equiv_r B$.

We think of $A \leq_r B$ as expressing that the set A is no more complicated than B. Put differently: given a way of solving problem B, we could use that to solve problem B.

Since $\leq_r$ is a preorder, the relation $\equiv_r$ is an equivalence relation on $\mathcal{P}(\mathbb{N})$. We typically denote the collection of equivalence classes by $\mathcal{D}_r$; the elements of this set are called *r-degrees*. The set $\mathcal{D}_r$ is of course now partially ordered by $\leq_r$. Finally, if $A \subseteq \mathbb{N}$, we sometimes write $deg_r(A)$ or $deg(A)$ or for the equivalence class of A under $\equiv_r$.

There are many reducibility relations, but not all of them are particularly interesting from a recursion-theoretic point of view. For example, the relations $A \leq_r B \Leftrightarrow A = B$, $A \leq_r B \Leftrightarrow A \subseteq B$ or simply $\forall A, B. A \leq_r B$ do not impose a meaningful new structure on the collection $\mathcal{P}(\mathbb{N})$. Our first meaningful example of a reducibility relation is given in the following definition:

Definition 4.1.2 (Many-one reduction)**.** Let $A, B \subseteq \mathbb{N}$ be two sets. We say that A *many-one reduces to* B, or that A is *many-one reducible to* B, notation

K	=	$\{x \mid \phi_x(x)\downarrow\}$
K_0	=	$\{[e,x] \mid \phi_e(x)\downarrow\}$
K_1	=	$\{x \mid W_x \neq \emptyset\}$
Tot	=	$\{e \mid \phi_e$ is a total function$\}$
Inf	=	$\{e \mid \phi_e$ has infinite domain$\}$
Fin	=	$\{e \mid \phi_e$ has finite domain$\}$
Cof	=	$\{e \mid \phi_e$ has cofinite domain$\}$
Empty	=	$\{e \mid \phi_e$ has empty domain$\}$
Cons	=	$\{e \mid \phi_e$ is a constant function (possibly partial)$\}$
Rec	=	$\{e \mid W_e$ is recursive$\}$
Ext	=	$\{e \mid \phi_e$ is extendable to a total recursive function$\}$
Ind	=	$\{e \mid W_e$ is an index set$\}$

Table 4.1: Notation for some common sets

$A \leq_m B$, if there exists a total recursive function f such that

$$x \in A \Leftrightarrow f(x) \in B$$

for all $x \in \mathbb{N}$. In this case, we say that $A \leq_m B$ *via* f.

The idea is that we can use the total recursive function f to transform the problem "is $x \in A$?" to the problem "is $f(x) \in B$?". Thus, if we were given a solution to the problem of determining membership in B, then that would imply a solution to the problem of determining membership in A.

Note that it is essential that f is recursive, otherwise there would be no guarantee that the passage from the problem "$x \in A$?" to the problem "$f(x) \in B$?" would be effective. It is also essential that f be total, otherwise the transformation may not be successful.

Lemma 4.1.3. *The relation $\leq_m$ is transitive and reflexive, and hence is a reducibility.*

Proof. Easy exercise for the reader. □

Let us look at some elementary examples first; we recommend that the reader verify their correctness in detail. We have listed a number of important sets in Table 4.1 for reference.

Example 4.1.4.

- Let A be any recursive set, and let B be any set which is not empty and not all of $\mathbb{N}$. Then we have $A \leq_m B$. Indeed, pick any $b \in B$, and any $b' \notin B$ and define
$$f(x) = \begin{cases} b & \text{if } x \in A \\ b' & \text{otherwise.} \end{cases}$$
Because A was assumed to be recursive, so is f, and we have forced
$$x \in A \Leftrightarrow f(x) \in B.$$
- By a simple extension of the argument given in the previous example, any two non-trivial recursive sets are m-equivalent (i.e. have the same m-degree).
- In the previous chapter we considered the halting set K. We may also consider $K_0 = \{[e,x] | \phi_e(x)\downarrow\}$. (We have used the pairing function to make this a subset of $\mathbb{N}$.) We have
$$x \in K \Leftrightarrow [x,x] \in K_0.$$
Since the diagonal function $x \mapsto [x,x]$ is recursive and total, this shows that $K \leq_m K_0$.
- We can also show the converse, namely $K_0 \leq_m K$. To this end, we need a total recursive function f with the property that
$$[e,x] \in K_0 \Leftrightarrow f([e,x]) \in K.$$
That is, f must satisfy
$$\phi_e(x)\downarrow \Leftrightarrow \phi_{f[e,x]}(f[e,x])\downarrow .$$
This can be achieved by defining, using the S-m-n theorem
$$f([e,x]) \bullet y = \phi_e(x).$$
Then $\phi_{f[e,x]}(y)\downarrow \Leftrightarrow \phi_e(x)\downarrow$.
- Consider the set Tot of codes of total functions. We may now show that $K \leq_m \mathsf{Tot}$ as follows: let $g(x) \bullet y = \phi_x(x)$ (again, S-m-n theorem). Then
$$\phi_x(x)\downarrow \Leftrightarrow g(x) \bullet y \downarrow \Leftrightarrow \phi_{g(x)} \text{ is total .}$$
The converse fails, as we shall see later.

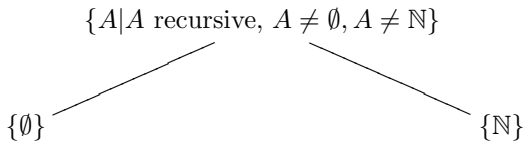
The reader may wish to practice his/her reduction skills by trying to see which other sets in Table 4.1 reduce to each other (warning: some are difficult!).

An important fact about m-reducibility is the following:

Lemma 4.1.5. *If $A \leq_m B$ and B is recursive, then so is A. If B is r.e. then so is A.*

Proof. This was exercise 3.13. □

This allows us to classify the recursive m-degrees (those containing recursive sets): there are exactly three, namely



where the degrees $\{\emptyset\}$ and $\{\mathbb{N}\}$ are strictly less than the degree of nontrivial recursive sets.

This picture certainly becomes more complicated for the r.e. sets! For example, K does not m-reduce to any recursive set (why?). This will be further investigated in the next section. For now, we conclude with a variation on m-reducibility:

Definition 4.1.6. Given $A, B \subseteq \mathbb{N}$, we say that A *1-reduces* to B, notation $A \leq_1 B$, if there is a total recursive injective function f such that

$$x \in A \Leftrightarrow f(x) \in B.$$

The requirement that the reducing function f be injective makes it more difficult to reduce a set to another set. For example, it is no longer possible to reduce an infinite set to a finite one. The reader should go through the above examples to see which of them are in fact examples of 1-reductions.

Trivially, we have $A \leq_1 B \Rightarrow A \leq_m B$. This means that we have an induced order-preserving function from the partially ordered set $\mathcal{D}_1$ of 1-degrees to the poset $\mathcal{D}_m$ of m-degrees.

We conclude this section by introducing two simple operations on sets. The first is the recursive product.

Definition 4.1.7 (Product)**.** Given two sets $A, B \subseteq \mathbb{N}$, define $A \otimes B$ to be the set $\{[a, b] | a \in A, b \in B\}$.

Thus, $A \otimes B$ is the image of $A \times B$ under the pairing operation. This operation is useful if we wish to extend reducibilities from set to general relations.

The next operation is the join of two sets:

Definition 4.1.8 (Join)**.** Given two sets $A, B \subseteq \mathbb{N}$, define $A \oplus B$ to be the set $\{2x | x \in A\} \cup \{2x + 1 | x \in B\}$.

It is readily seen that $A \leq_m A \oplus B$ and that $B \leq_m A \oplus B$. Moreover, given any C such that $A \leq_m C$ (via f, say) and $B \leq_m C$ (say via g) then $A \oplus B \leq_m C$: simply use the function

$$h(x) = \begin{cases} f(x/2) & \text{if } x \text{ even} \\ f((x-1)/2) & \text{if } x \text{ odd.} \end{cases}$$

This proves:

Proposition 4.1.9. *The partially ordered set of m-degrees $\mathcal{D}_m$ has binary joins.*

A poset which has binary joins is sometimes called a join-semilattice. As an exercise, the reader should verify if the above result goes through for 1-degrees.

4.1.2 M-complete sets

Whenever we are given a collection of sets, we may ask if there is one "hardest" problem in that set. More precisely, we ask for a set A to which all other sets in the collection m-reduce. Such a set A is then called m-complete (for that particular collection). In this section we study m-complete sets for the collection of r.e. sets.

Definition 4.1.10 (completeness)**.** A set $A \subseteq \mathbb{N}$ is called *m-complete (1-complete)* if

- A is r.e.
- for every r.e. set B we have $B \leq_m A$ ($A \leq_1 B$).

Does such a set exist? The answer is yes, and we do not even have to look very far:

Theorem 4.1.11. *The Halting set $K = \{x | \phi_x(x)\downarrow\}$ is 1-complete and hence also m-complete.*

Proof. We already know that K is r.e., so we have to show that $B \leq_1 K$ for any r.e. set B. Suppose $B = W_e$; we need a function f such that

$$\phi_e(x)\downarrow \Leftrightarrow \phi_{f(x)}(f(x))\downarrow .$$

Using the S-m-n theorem again, put

$$f(x) \bullet y = \phi_e(x).$$

Then we get $x \in W_e \Leftrightarrow f(x) \in K$, as needed. Moreover, the function f may be taken to be injective. □

Of course, once we have a particular m-complete set A, then all of the sets in $deg(A)$ are m-complete. In particular, the set K_0 is m-complete (one can also easily show this directly). Conversely, any two m-complete sets must have the same m-degree, because they reduce to each other.

We now set out to understand complete sets better. Our first main result states, among other things, that any two 1-complete sets must be recursively isomorphic. Let us first define this notion:

Definition 4.1.12. Let $A, B \subseteq \mathbb{N}$ be two sets.

(i) A recursive function which is also total and bijective is called a *recursive permutation*

(ii) A and B are *recursively isomorphic* if there is a recursive permutation f such that $f[A] = B$. Notation: $A \equiv B$.

It is easily verified that $\equiv$ is an equivalence relation on subsets of $\mathbb{N}$; its equivalence classes are called *recursive isomorphism types*. From an abstract point of view, this is a very natural notion, and it is good practice to always ask if a given class of sets is invariant under recursive isomorphism. For example, the recursive sets are, since if A is recursive and $A \equiv B$, then so is B. The same is true for the r.e. sets, but not, for example, for the index sets (why?).

One of the basic results in set theory is called the "Cantor-Schröder-Bernstein Theorem": it states that if we have injective functions $A \longrightarrow B$ and $B \longrightarrow A$ then $A \cong B$. Myhill's theorem is the recursion-theoretic analogue.

Theorem 4.1.13 (Myhill's Isomorphism theorem)**.** *If $A \leq_1 B$ and $B \leq_1 A$ then $A \equiv B$.*

Proof. This is, like the Cantor-Schröder-Bernstein theorem, a back-and-forth argument. We shall define a recursive function h and its inverse h^{-1} in stages. The part of the function h which we have defined after stage s will be denoted h_s, and its inverse by h_s^{-1}.

Start by putting $h_0 = \emptyset$, the empty function. We are given injective recursive functions f, g such that $A = f^{-1}(B)$ and $B = g^{-1}(A)$. We proceed inductively, assuming that we have defined the finite function h_s (and hence also its inverse). By induction hypothesis, we assume that this function is 1-1 and that it has the property that $y \in A$ if and only if $h(y) \in B$, for all $y \in dom(h_s)$.

Now at an odd stage $s+1 = 2x+1$, we make sure that $h(x)$ will be defined properly. If $h_s(x)$ was already defined, then we let $h_{s+1} = h_s$ and move on to the next stage. Otherwise, we wish to assign a value $h(x)$. This is where we use f: we go back and forth by generating the sequence

$$C_s = \langle f(x), h_s^{-1}(f(x)), f(h_s^{-1}(f(x))), h^{-1}(f(h_s^{-1}(f(x)))), \ldots \rangle$$

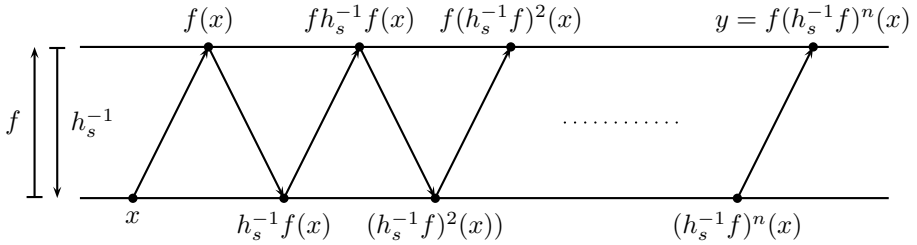


Figure 4.1: Back-and-forth

until we find that an application of f brings us outside the range of h_s (then the generation of the set halts). The picture in Figure 4.1 illustrates this.

We let y be that last element of the sequence; now define h_{s+1} to be the extension of h_s by

$$h_{s+1}(x) = y.$$

At an even stage, we reverse the roles of h and h^{-1}.

Finally, we let $h = \bigcup_{s\in\mathbb{N}} h_s$, the union of the finite functions h_s.

There are a number of things to verify. First, that h is recursive: but the above strategy gives an algorithm for computing $h(x)$. First go through the stages $s = 0, \ldots, 2x$; this involves only manipulation of finite (hence recursive) data, together with applications of the recursive functions f, g. Then at stage $2x + 1$ we find the desired value $h(x)$ (if not earlier). Next, h is injective: at each stage we made sure to define $h(x)$ in such a way that it was not in the range of h_s. Moreover, h is surjective: for any y there is a stage at which we define $h_s^{-1}(y) = x$; this automatically defines $h_s(x) = y$.

Finally, we need to check that $h[A] = B$. If $x \in A$, then the sequence C_s will consist solely of elements alternating between B and A. Thus the last element $h_s(x)$ will be in B. Conversely, if $x \notin A$, then the last element will not be in B. □

Corollary 4.1.14. *Any two 1-complete sets are recursively isomorphic.*

We shall see in the next section that the same holds for the m-complete sets.

4.1.3 Creative sets

While Myhill's theorem shows how closely related complete sets are to each other, we still need to understand what sets apart complete sets from non-complete sets. In this section we discuss a property enjoyed by complete sets which in fact turns out to characterize them.

Consider again K, the halting set. We know that its complement $\mathbb{N} - K$ is not r.e. because that would imply that K were recursive, which is not the case. To say that $\mathbb{N} - K$ is not r.e. is to say that it is different from any set of

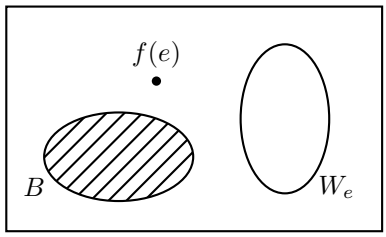


Figure 4.2: B is creative via f

the form W_e. Put yet differently: if we are given a set of the form W_e which is disjoint from K, then there must exist an element y_e which lies outside both K and W_e. This element y_e now witnesses the fact that $\mathbb{N} - K \neq W_e$.

How do we find such an element y_e? Well, suppose $W_e \cap K = \emptyset$. Then the element e cannot lie in W_e, since that would mean $\phi_e(e)\downarrow$, i.e. $e \in K$, contradicting that W_e and K have no elements in common. At the same time, this shows that e cannot be in K. Thus e is the sought-after element y_e outside K and W_e.

Note one more thing: we have found, for each W_e disjoint from K, an element y_e such that y_e is not in K nor in W_e. But more is true: we have found a way of computing y_e effectively from the code e of W_e (namely via the identity function). It is this property of K which we now turn into a definition:

Definition 4.1.15 (Productive and creative sets)**.** Let A be a set. We call A *productive* if there exists a recursive function f such that for each code e we have

$$W_e \subseteq A \Rightarrow f(e) \in A - W_e.$$

In this case, we call f a *productive function* for A, and say that A is productive via f.

Moreover, an r.e. set B is called *creative* if its complement is productive. If the complement of B is productive via f, then B is said to be creative via f.

See figure 4.2 for an illustration.

The above discussion shows that the complement of K is productive and that therefore K is creative.

Just to get a feel for the definition, the reader should verify that an r.e. set can never be productive. From that, it follows that recursive sets are not creative.

The following lemma states a useful property of the creative sets, namely that they form a cone w.r.t. m-reducibility:

Lemma 4.1.16. *If A is creative and $A \leq_m B$ then B is also creative.*

Proof. Let f be a total recursive function f for which

$$x \in A \Leftrightarrow f(x) \in B \tag{4.1}$$

and let A be creative via k.

Now define a function g via

$$g(x) \bullet y = \phi_x(f(y)).$$

Then g has the property that

$$y \in W_{g(x)} \Leftrightarrow f(y) \in W_x. \tag{4.2}$$

Now to show that B is creative, suppose we are given W_e disjoint from B. Now we must have that $W_{g(e)} \cap A = \emptyset$, since $y \in W_{g(e)}$ gives $f(y) \in W_e$ and $y \in A$ gives $f(y) \in B$.

But now we can use the creativeness of A: this means that $k(g(e)) \notin A$ and $k(g(e)) \notin W_{g(e)}$. The first implies, using equation 4.1 that $f(k(g(e))) \notin B$; the second implies, using 4.2, that $f(k(g(e))) \notin W_e$. This proves that the composite function fkg is a creative function for B. □

The following theorem, due again to Myhill, shows that the above abstract property of creativeness indeed captures the essence of m-complete sets:

Theorem 4.1.17 (Myhill)**.** *An r.e. set A is m-complete if and only if it is creative.*

Proof. One direction is implied by the preceding lemma: if A is m-complete then $K \leq_m A$; since we know K to be creative, so is A.

For the other direction we assume that A is creative via f. We wish to show that $K \leq_m A$ via some function h. We first define an auxiliary function g using the recursion theorem:

$$g(z) \bullet y = \begin{cases} \phi_z(z) & \text{if } y = fg(z) \\ \uparrow & \text{otherwise.} \end{cases} \tag{4.3}$$

This function g has the property that if $z \in K$ then $W_{g(z)} = \{fg(z)\}$, and $W_{g(z)} = \emptyset$ otherwise.

Now let $h(x) = fg(x)$. We claim that $K \leq_m A$ via h.

Suppose first that $x \in K$. By the above, that means that $W_{g(x)} = \{fg(x)\}$. We wish to show that $fg(x) \in A$; if this were not the case then we would have $\{fg(x)\} \cap A = \emptyset$. By creativity of A this would imply that $f(g(x))$ is an element outside A and $W_{g(x)}$. Contradiction, since $W_{g(x)} = \{fg(x)\}$.

On the other hand if $fg(x) \in A$, then we wish to show $x \in K$. Suppose $x \notin K$; then we have $W_{g(x)} = \emptyset$, and hence trivially that $A \cap W_{g(x)} = \emptyset$. Creativity of A gives an element $f(g(x))$ outside A. Again we have reached a contradiction.

This completes the proof. □

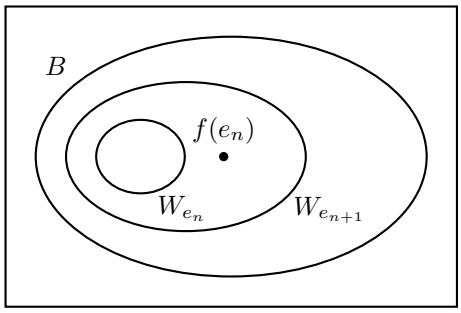


Figure 4.3: Production of an infinite set inside B

4.1.4 1-complete sets

Looking again at the proof of theorem 4.1.17, we see that the function constructed to show that $K \leq_m A$ is a composite of two functions; the first is the creative function for A, and the second is obtained by the S-m-n theorem. We would like to improve on the result by showing that in fact $K \leq_1 A$. Since the S-m-n theorem may always be taken to produce injective functions, the issue is if we can find an injective creative function for our creative set A. We will show that this is indeed possible, but first we need to understand one feature of creative and productive sets.

Let B be a productive set, say with productive function f. We know B cannot be r.e. and hence must be infinite. Therefore we can consider infinite subsets of B. In particular we may ask: does there exist an infinite r.e. set $C \subseteq B$?

Let us construct such C in stages, adding one element to the set at each stage. Start by letting $C_0 = W_{e_0} = \emptyset$, where e_0 is a code for the empty function; since it is contained in B, the productive function f gives an element $f(e_0)$, which lies in B.

Now let e_1 be a code of the set $\{f(e_0)\}$, and define $C_1 = W_{e_1}$. Since $C_1 \subseteq B$, we may apply f again to obtain an element $f(e_1) \in B$, which must be different from $f(e_0)$ by productivity. We continue in this style, by letting at stage $n+1$:

$$e_{n+1} = \text{ a code for the finite set } C_n \cup \{f(e_n)\},$$

and letting $C_{n+1} = W_{e_{n+1}}$.

This gives a sequence of sets $C_0 \subsetneq C_1 \subsetneq \ldots$ and we may put $C = \bigcup_n C_n$. Clearly C is infinite. By induction it is easily proved that $C \subseteq B$. Moreover, the construction of C is recursive and hence C is r.e. as desired. Figure 4.3 illustrates the construction of the set C.

We have shown:

Lemma 4.1.18. *For every creative set A there exists an infinite r.e. set disjoint from A.*

But in fact, we have more: not only does there exist an infinite r.e. set, we can in fact construct it in an effective manner from a code of the creative function. We will now use the same idea to construct a creative function which is injective.

Suppose that f is a creative function for A. Then there is a recursive function t such that

$$W_{t(x)} = W_x \cup \{f(x)\}.$$

(We understand here that $t(x)$ is defined if and only if $f(x)$ is defined.) Because f is a creative function we have

$$W_n \cap A = \emptyset \Rightarrow W_{t(n)} \cap A = \emptyset.$$

Moreover, we have that $W_n \cap A = \emptyset$ implies that all elements of the sequence $f(n), f(t(n)), f(t^2(n)), \ldots$ are different.

We wish to define a new creative function g which is injective. We start by letting $g(0) = f(0)$. Now we proceed inductively: in order to define $g(n)$, we assume we have define $g(0), \ldots, g(n-1)$ in such a manner that they are all different and such that for all $i < n$: if $W_i \cap A = \emptyset$ then $g(i) \notin A \cup W_i$.

We find the element $g(n)$ as follows: start enumerating the sequence

$$f(n), f(t(n)), f(t^2(n)), \ldots.$$

Now there are two possibilities:

1. for some i: $f(t^i)(n) = f(t^{i+1}(n))$ (the sequence stagnates). In that case we know that we couldn't have had $W_n \cap A = \emptyset$. Therefore it doesn't matter what the value of $g(n)$ is, as long as it is different from the previous values. We choose to set

$$g(n) = max\{g(0), \ldots, g(n-1)\} + 1.$$

2. if the sequence does not stagnate, then at some point we must encounter an element y different from all of $g(0), \ldots, g(n-1)$. (Of course, this happens after maximally n steps.) Then we let $g(n) = y$. Because $y = f(t^i(n))$ for some i, this ensures that $W_n \cap A = \emptyset \Rightarrow y \notin A \cup W_n$.

This proves:

Proposition 4.1.19. *Every creative set has an injective creative function.*

By the discussion at the beginning of this section, we therefore also have:

Corollary 4.1.20 (Characterization of complete sets)**.** *For a set A, the following are equivalent:*

(i) *A is m-complete*

(ii) *A is 1-complete*

(iii) *A is creative*

Because of the fact that all 1-complete sets are recursively isomorphic (Corollary 4.1.14) we conclude that all the creative, m-complete and 1-complete sets form a single recursive isomorphism type.

4.2 Incomplete sets

We have seen that all non-trivial recursive sets form a single m-degree; we have also shown that there is an m-degree consisting of the creative sets and that this degree consists exactly of those sets recursively isomorphic to the Halting set. Both these degrees are called *r.e. degrees* because they consist of r.e. sets.

We now ask the following question:

Question 4.2.1 (Post's Problem for m-degrees)**.** Are there m-degrees strictly between the degree of K and the degree of recursive sets?

Put slightly differently, do there exist sets which are not recursive but also not m-complete?

This question can be asked for any kind of reducibility relation, and has become known as "Post's Problem", after Emil Post who, in 1944, asked the question for Turing reducibility (a reducibility which we shall meet later in this chapter). In its original form, the question turned out to be difficult; Post himself solved the problem for a number of stronger reducibilities (including m-reducibility), but it took 12 years before the original problem was solved independently by Friedberg (1957) and Muchnik (1956). The technique used to solve the problem is called a *priority argument*, although it is really a construction, not an argument. Priority arguments have since become the hallmark technique of recursion theory.

Post's Problem for m-degrees is fortunately much less complicated, and we shall now present a number of solutions. The first is rather ad hoc, but easy to grasp. The second solution is based on an uncomplicated priority argument, and this will be a good opportunity to get acquainted with these arguments. The third solution is more conceptual, and is based on the notion of a random number.

4.2.1 Simple sets

The starting point for our attack on Question 4.2.1 is our observation that every m-complete set must not only have an infinite complement, but that that complement must contain an infinite r.e. subset. If we could build a set A with the following two properties:

1. A is r.e.
2. $\mathbb{N} - A$ is infinite but contains no infinite r.e. subsets

then we would be done: first, A cannot be recursive because that would imply that its complement would also be recursive. Moreover, such A cannot be creative, and hence not m-complete.

A set with the above two properties is called a *simple* set. We may slightly reformulate the definition: a set A is simple if it is r.e., has infinite complement, and we have

$$W_e \text{ infinite } \Rightarrow W_e \cap A \neq \emptyset.$$

That is, a simple set must intersect every infinite r.e. set! We may therefore think of simple sets as sets which have a very "thin" complement.

Our problem has become: find a simple set! The reader should note that the construction of such a set A needs to balance two requirements: first, for each e, we wish to ensure that

$$W_e \text{ infinite } \Rightarrow A \cap W_e \neq \emptyset.$$

Thus, we have to make sure to put an element of W_e into A whenever W_e is infinite. And second, we have to make sure not to put too many elements into A, because we wish A to have infinite complements. The first requirements are called *positive requirements*, the second *negative*.

Post's construction of a simple set. This construction is fairly easy. We will construct our set A as the range of a recursive function h, and thus it will automatically be r.e. The function h will have the property that whenever it is defined on some input x, we will have $h(x) > 2x$. This guarantees that for each number e, there are at most e elements of the form $h(x)$ in the set $\{0, 1, \ldots, 2e\}$. Thus the complement of A will be infinite.

Define the functions g, h by

$$g(x) = \mu z.(T(x, (z)_1, (z)_2) \wedge (z)_1 > 2x) \qquad h(x) = (g(x))_1.$$

Suppose that, for a number e, we have that $h(e)$ is defined. Then it follows that

- $h(e) > 2e$
- $h(e) \in W_e$.

It remains to be verified that if W_e is infinite, we indeed get $A \cap W_e \neq \emptyset$. However, if W_e is infinite, then $h(e)$ must be defined, since W_e will contain an element $y > 2e$. By construction, we then have $h(e) \in W_e \cap A$.

4.2.2 The priority construction

We will now present a construction of a simple set which is based on a priority argument. We shall first explain the general idea behind such constructions, and give a useful way to visualise them.

Requirements and priority. In a priority construction one constructs a set (or more generally a collection of sets) with certain desired properties. These properties (which tend to be infinitary in nature) are split up into countably many *requirements*, each of which is manageable. For example, the defining properties of a simple set can be split up into the following requirements:

$$\begin{aligned} P_e: &\quad W_e \text{ infinite } \Rightarrow W_e \cap A \neq \emptyset \\ N_e: &\quad \mathbb{N} - A \text{ contains at least } e \text{ elements.} \end{aligned}$$

Clearly, if a set A satisfies these requirements for each e, then it must be simple. Note that it wouldn't be hard to build a set satisfying only the positive requirements, or only the negative ones. The problem is to balance the two, and one way of doing that is by assigning priorities to them. In this case, we do that as follows:

$$P_0 > N_0 > P_1 > N_1 > \cdots$$

This means that P_0 has highest priority, N_0 second highest, etc. During the actual construction, that means we shall always look to satisfy requirements of highest possible priority. (I.e. we're not going to satisfy P_9 if we can satisfy N_8 first, etcetera.)

In a priority argument, one builds the desired set in stages, approximating the final result A by finite sets $A_0, A_1, \ldots$. We begin at stage 0 by letting $A_0 = \emptyset$; at stage $s+1$ of the construction we have built finite sets $A_0 \subseteq A_1 \subseteq \cdots \subseteq A_s$ and we explain how to find A_{s+1} from A_s. Typically this is done by looking for the requirement with highest priority which hasn't been satisfied yet and by adding an element to A_s in such a way that the requirement is satisfied. It's not always possible to satisfy a requirement, and in that case we let $A_{s+1} = A_s$. In the end we shall let $A = \bigcup_{s \in \mathbb{N}} A_s$.

The important point is not only that each A_s will be finite (and hence recursive) but also that the process of finding A_{s+1} in terms of A_s is recursive, uniformly in s. This guarantees that the resulting set A will be r.e., being definable as

$$x \in A \Leftrightarrow \exists s. x \in A_s.$$

Movable markers. In order to visualize this process of gradually adding elements to the set A, we use what are sometimes called "movable markers". We imagine that we have a countable set of markers $\Gamma_0, \Gamma_1, \ldots$, which we can use to label certain elements. This may be pictured as follows:

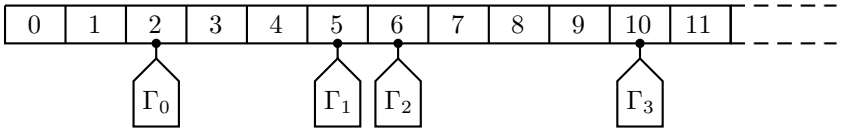
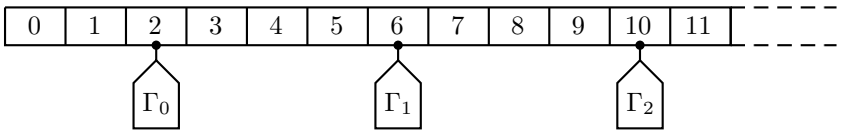


Figure 4.4: Movable markers

In figure 4.4 we have used the 0-th marker Γ_0 to label the number 2, the 1-st marker Γ_1 to label number 5, etcetera. We write $\Gamma_n = m$ to indicate that the n-th marker is at position m. One should imagine that in figure 4.4 the natural numbers in the boxes are fixed in position, but that one can slide the markers sideways. However, one cannot have two markers at the same position: essentially, the markers specify a subset of $\mathbb{N}$ in a 1-1 fashion.

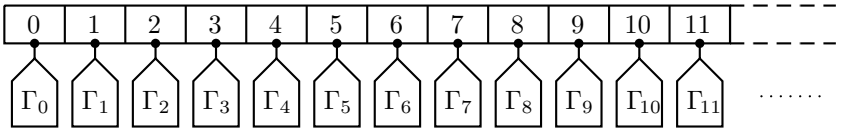
The idea is that, at stage s of our construction, the markers form an enumeration of the elements which are currently *not* in A. Thus, the elements which we have already put in A are not marked. If at stage s we decide to put an element in A, then we have to remove its marker and move all higher markers one to the right. For example, if at stage s our markers would be in the position of Figure 4.4 (and thus $A_s = \{0, 1, 3, 4, 7, 8, 9, 11, \ldots\}$), and we would decide to put the element 5 into A, then we would obtain the following:



Note is that in this process, we only move markers to the right, never to the left; also, if we are adding an element y to our set, then the markers to the left of y do not move.

Because we will move the markers during the construction (several times perhaps) we sometimes use the following notation: $\Gamma_n^s = m$ indicates that at stage s, the n-th marker is at position m.

The construction. Since we start with the empty set A_0, initially every $n \in \mathbb{N}$ belongs to the complement $\mathbb{N} - A_0$, and thus we have $\Gamma_n^0 = n$ for all n. Thus in the initial stage things look as follows:



We assume we have carried out the first s stages of the construction and are given a finite set A_s. At stage $s+1$ we wish to take care of a requirement P_e, with $e \leq s$ which is not yet satisfied. However, the question "is W_e infinite?" cannot be answered recursively. We therefore only consider those elements in W_e for which the computation has not too big a code:

$$W_{e,s} = \{x | \exists t \leq s . T(e, x, t)\}.$$

Clearly each $W_{e,s}$ is finite, and we have that $W_e = \bigcup_{s \in \mathbb{N}} W_{e,s}$. Moreover we have $W_{e,s} \subseteq W_{e,s+1}$.

We now search the for the least $e \leq s$ such that

- $W_{e,s} \cap A_s = \emptyset$
- there is $n \geq e$ such that $\Gamma_n^s \in W_{e,s}$.

The first condition $W_{e,s} \cap A$ means that at this stage requirement P_e looks unsatisfied (since W_e could be infinite). We will try to take satisfy P_e by choosing an element of $W_{e,s}$ to add to A_s, but we wish to do this in such a way that we do not endanger the negative requirements. Therefore we leave the first e marked elements alone, and only consider marked elements Γ_n^s in $W_{e,s}$ for which $n \geq e$. If such n exists, then we take the least such and add it to A_s. Thus we put $A_{s+1} = A_s \cup \{y\}$; and

$$\Gamma_n^{s+1} = \begin{cases} \Gamma_n^s & \text{if } n < y \\ \Gamma_{n+1}^s & \text{if } y \leq n. \end{cases}$$

Thus the markers to the left of the element Γ_n^s stay where they are, the rest moves to the next position.

Finally, if no e with the required properties exists, then we do nothing and go to the next stage.

It remains to prove that the construction is correct. We start by proving that the resulting set A has infinite complement. To this end, it suffices to show that *every marker Γ_n moves only finitely many times*. Indeed, if Γ_n does not move after stage s, then we know that Γ_n will never be in A, and thus must be an element of $\mathbb{N} - A$. If this is the case for each Γ_n, then the complement will in the end contain all the marked elements.

Now if we, at a certain stage, satisfy requirement P_e, then we do not move the markers $\Gamma_0, \ldots, \Gamma_{e-1}$. Thus, a marker Γ_n moves only if we satisfy a requirement P_e with $e \leq n$. Since each requirement is satisfied only once, this means that the marker Γ_n moves at most $n+1$ times, namely when we satisfy $P_0, \ldots, P_n$.

Finally, the constructed set A is simple. We show the following statement by induction: for each e there is a stage s such that all requirements P_i are satisfied at stage s, for $i < e$. Suppose that we are at a stage s_0 where we have satisfied all requirements up to P_{e-1}. We need to show that there is a stage where we have satisfied P_e as well.

Suppose we never satisfy P_e, that is, we have $W_e \cap A = \emptyset$. Then at each stage $s \geq s_0$, there cannot exist an $n \geq e$ with $\Gamma_n^s \in W_{e,s}$, otherwise we would put such Γ_n^s into A and satisfy P_e. However, only finitely elements $y \in W_{e,s}$ can be of the form Γ_n, with $n < e$. (To be precise, at most e elements can be of this form.) Thus if W_e is infinite, at some stage s we must find an $n \geq e$ with $\Gamma_n^s \in W_{e,s}$.

4.2.3 *Random numbers

Our third example of a simple set will not be based on a technical construction, but will be more mathematical in nature. Let us begin with a definition

Definition 4.2.2 (Kolmogorov complexity)**.** For a number x, define its *Kolmogorov complexity* to be the least code e which, on input 0, gives output x. That is:

$$K(x) = \mu e.\phi_e(0) = x.$$

One should think of $K(x)$ as the shortest possible description of x. For example, the Kolmogorov complexity of 2 is the code of the shortest possible program which outputs 2; this is the program consisting of two instructions Inc0. It should be clear that for some larger numbers, one can come with efficient programs. For example, using the recursion theorem we can pick a code e such that $\phi_e(x) = e + 1$ for each x.

The reader who enjoyed the busy beaver problem may wish to think about how this problem is related to the complexity function K.

Definition 4.2.3 (Random numbers)**.** A number x is *random* if $x \leq K(x)$.

Intuitively, this means that x is its own shorter description. We first observe that the set of non-random numbers is r.e.: we have

$$x \text{ is not random } \Leftrightarrow \exists e < x.\phi_e(0) = x.$$

We shall now show that the set of non-random numbers intersects every infinite r.e. set; equivalently, there cannot exist an infinite r.e. set of random numbers. So suppose that B is an infinite set of random numbers, and define

$$h(x) \bullet 0 = \mu z.(x < z \wedge z \in B).$$

Note that $h(e) \bullet 0 > e$ for all e. Since h is total recursive, by the Fixed point theorem there exists a fixed point e, so that we have

$$\phi_e = \phi_{h(e)}.$$

Now consider $\phi_e(0) = \phi_{h(e)}(0)$. By definition of h, this number is in B, and therefore is random, and hence must be smaller than e. Contradiction.

We conclude:

Proposition 4.2.4. *The set of non-random numbers is simple.*

4.3 Relative computability

We shall now turn to a weaker kind of reducibility, called Turing reducibility. From a computational point of view this is a very natural notion, since it makes precise which functions are computable *assuming that we have access to the values of a given function.*

4.3.1 Oracles

We have defined the recursive functions to be the least class containing the initial functions and closed under primitive recursion, composition and minimalization. We may choose to add a function f to the initial functions before closing under the given operations. That is, we consider all functions which are recursive *relative* to the given f (which need not be recursive). Clearly, if f was recursive, then this doesn't give anything new. But if f was non-recursive, then we have extended our class of recursive functions.

Definition 4.3.1 (Relative computability)**.** Let f be a given function. A function g is *recursive relative to* f, or *recursive in* f, or simply f*-recursive*, when it can be obtained from f and the initial functions using finitely many applications of composition, primitive recursion and minimalization.

We write $g \leq_T f$ to express that g is recursive in f. It is easily verified that the relation $\leq_T$ is a preorder, i.e. is reflexive and transitive. We extend this notation to sets, identifying sets with their characteristic functions. Thus $A \leq_T B$ stands for $\chi_A \leq_T \chi_B$, and this is a reducibility relation on subsets of natural numbers.

The formalism of register machines allows for a more colorful interpretation of relative computation. Imagine that our register machine is connected to an external database, which contains all the values of the function f. Also imagine that we have an extra instruction in our language of the form $R_i \leftarrow f(R_j)$, which takes the value of the j-th register, queries the database for the value $f(R_j)$, and places the result of the query in register R_i. We agree that this query will only be successful if f is indeed defined at that particular value; otherwise the computation diverges.

Now the class of functions computable by this machine (which we call the functions computable relative to f) consists exactly of the functions recursive in f: one may still show that these functions are closed under composition,

recursion and minimalization. Conversely one can still encode programs as numbers; the resulting T-predicate will now be f-recursive.

The database for f is usually referred to as an *oracle* for f, and we say that, during the computation, the program queries the oracle.

Examples 4.3.2.

1. Any recursive function is recursive relative to any f.
2. The function
$$g(x) = \begin{cases} 1 & \text{if } f(x) = 0 \\ \uparrow & \text{otherwise.} \end{cases}$$
is recursive in f.
3. By contrast, the function
$$h(x) = \begin{cases} 1 & \text{if } f(x) = 0 \\ 0 & \text{otherwise.} \end{cases}$$
is, in general, only recursive in f when f is total.
4. Consider the function
$$k(x) = \begin{cases} 1 & \text{if } f \text{ has infinite range} \\ 0 & \text{otherwise.} \end{cases}$$
We are not allowed to ask the oracle whether f has infinite range or not - the only questions we can ask the oracle are of the form "What is $f(x)$?", for one value of x at a time. So in the current form, we cannot, even recursively in f, say what the function k is. However, it is a constant function in either case.
5. By contrast, the function
$$l(x) = \begin{cases} 1 & \text{if } f(x) \in K \\ 0 & \text{otherwise.} \end{cases}$$
is in general not constant; we cannot decide *recursively* in f whether $f(x) \in K$ or not. Therefore l is not, in general, recursive in f.

Of course, one should distinguish the general situation from particular examples: if, for a particular function f, we happen to know, say, that f always takes values in, codes of total functions, then we may conclude that h is recursive in f.

4.3.2 Relativised recursion theory

We may now redo all of the theory developed so far, systematically replacing "recursive" by "recursive in f". We shall not do so in detail since most of this is straightforward, but we shall state a few of the key results, both to get the reader acquainted with this extra generality and to establish notation.

Theorem 4.3.3 (Normal form theorem, relativised version)**.** *For any function f, there exists a predicate $T^f(e,x,y)$ and a function $U^f(y)$ such that*

- *T^f is primitive recursive in f*
- *U^f is primitive recursive*
- *for each function g recursive in f there is a code e such that*

$$g(x) = U^f(\mu y.T^f(e,x,y)).$$

Note that T^f is PR in f since it needs to verify whether the instructions $R(i) \leftarrow f(R_j)$ were executed correctly, but that the outcome function is still PR, as it merely decodes the computation.

From the relativised Normal form theorem we get an enumeration of the n-ary functions recursive in f:

$$\phi_0^{f,(n)}, \phi_1^{f,(n)}, \ldots.$$

Of course, for unary functions we simply write ϕ_e^f. In case of a characteristic function of a set A we shall write ϕ_e^A for the e-th function recursive in A.

We obtain:

Theorem 4.3.4 (Enumeration theorem, relativised version)**.** *For each $n > 0$ there is an f-recursive function $\Phi^{f,(n)}$ such that for each n-ary f-recursive g there is a code e such that*

$$\Phi^{f,(n)}(e, x_1, \ldots x_n) = g(x_1, \ldots, x_n).$$

and of course also

Theorem 4.3.5 (Parameter theorem, relativised version)**.** *For each $n, m > 0$ there is a primitive recursive function S_m^n such that for all e we have*

$$\Phi^{f,(m)}(S_m^n(e, x_1, \ldots, x_n), y_1, \ldots, y_m) = \Phi^{f,(n+m)}(e, x_1, \ldots, x_n, y_1, \ldots, y_m).$$

Since the S_m^n functions merely reorganize programs, they do not need to access the oracle and hence are still primitive recursive.

We may also relativise the notion of a recursively enumerable set:

Definition 4.3.6. A set A is *recursively enumerable relative to f* when there is an f-recursive function g such that $A = dom(g)$. We shall say that A is *r.e. in f*.

We have an enumeration of the sets r.e. in f:

$$W_0^f, W_1^f, \ldots.$$

The results from chapter 3 generalize immediately. For example, we have the complementation theorem:

Theorem 4.3.7. *If A is r.e. in f and $\mathbb{N}-A$ is r.e. in f, then A is f-recursive.*

Remark 4.3.8. While most of the time a mere inspection of a proof shows that a result relativises, this is not automatic! There is no general principle which states that all results relativise, so for each result, we need to inspect the proof. (Indeed, certain results fail to relativise, although we shall not be concerned with those.)

4.3.3 Turing degrees

When we restrict our attention to sets of natural numbers, the relation $\leq_T$ is a reducibility relation. We now set out to study its basic properties. We use a notation similar to that for m-degrees: we write $A \equiv_T B$ for $A \leq_T B$ and $B \leq_T A$, and the equivalence classes w.r.t. are called *Turing degrees*, or simply *T-degrees*. The partially ordered set of T-degrees will be denoted by $\mathcal{D}_T$.

We start by making a few basic observations about T-reducibility.

Lemma 4.3.9.

(i) *for any set A we have $A \equiv_T \mathbb{N} - A$*

(ii) *if A is recursive then $A \leq_T B$ for any set B*

(iii) *if $A \leq_T B$ and B is recursive then so is A*

Proof. Left as an exercise to the reader. □

The first two properties give us an idea of how T-reducibility differs from m-reducibility. The third property holds for m-reducibility alike; note however that the statement "If $A \leq_T B$ and B is r.e. then so is A" is false: consider the Halting set and its complement for a counterexample.

In spite of the differences there are, however, various connections between the two relations. The most important of those is:

Lemma 4.3.10. *If $A \leq_m B$ then also $A \leq_T B$.*

Proof. If $x \in A \Leftrightarrow f(x) \in B$ for some total recursive f, then the characteristic function of A is clearly recursive in the characteristic function of B. □

Thus, m-reducibility is a stronger relation than T-reducibility, and consequently the m-degrees are finer than the T-degrees.

One way of understanding the difference is by considering limitations on how computations can query the oracle. For m-reducibility, in order to compute whether $x \in A$, we compute $f(x)$ (no oracle needed there) and then present the answer to the oracle for χ_B. Thus, for each input, we ask precisely one question to the oracle. By contrast, if $A \leq_T B$, then many questions may be needed. Moreover, it may depend on the particular input value how many questions: the number of questions may not even be recursive in the input.

Remark 4.3.11. T-reducibility and m-reducibility form two ends of a spectrum of reducibility relations, called *strong reducibilities*. Intermediate relations are obtained by imposing certain restrictions on how often, and in what ways, the computation can query the oracle. There are also weaker reducibilities; we shall not look into those.

In the previous chapter we saw that the m-degrees had binary joins; the following lemma strengthens this result to the T-degrees.

Lemma 4.3.12. *Write $A \oplus B$ for the set*

$$A \oplus B = \{2x | x \in A\} \cup \{2x+1 | x \in B\}.$$

Then $A \oplus B$ is the least upper bound of A and B with respect to $\leq_T$.

Proof. Since $A \leq_m A \oplus B$ and $B \leq_m A \oplus B$ it follows that $A \oplus B$ is an upper bound for A, B. If $A \leq_T C$ and $B \leq_T C$, then we get $A \oplus B \leq_T C$ as follows: given an even number $2x$, solve (with help of the oracle for C) the problem whether $x \in A$. Given an odd number $2x+1$, use the fact that B is recursive in C to solve the problem whether $x \in B$. □

Thus, $\mathcal{D}_T$ is a join-semilattice. It can be proved that $\mathcal{D}_T$ does not admit binary meets, but the proof is outside the scope of these lectures.

The jump operation. We now look at a way of producing sets with higher and higher degrees of unsolvability. The idea is to extend the basic result that K, the halting set, is not recursive.

Given a function f let

$$K^f = \{x | \phi_x^f(x)\downarrow\}.$$

The set K^f is called the Halting set relative to f. As we expect, we have:

Theorem 4.3.13 (Unsolvability of the Halting problem, relativised version)**.** *The set K^f is f-r.e. but not f-recursive.*

Proof. Analogous to the absolute case. □

There is a question here: suppose we let f be the characteristic function of the Halting set. Thus we have access to the oracle for the Halting set, and we can solve the Halting problem with help of this oracle. Now why does that not contradict the above result, which states that the Halting problem is not recursive relative to f? The answer is simple: in solving the old Halting problem (by allowing an oracle for it) we have created a new Halting problem, which is even harder than the previous one. The theorem states that this new problem cannot be solved, not even using the oracle for the old problem.

We now use this idea to create more and more complicated sets:

Definition 4.3.14 (Jump operation)**.** Let A be a set. Define

$$A' = K^A = \{x | \phi_x^A(x)\downarrow\}.$$

The set A' is called the *jump of* A.

From theorem 4.3.13 we immediately get:

Theorem 4.3.15. *The set A' is A-r.e. but not A-recursive.*

This makes precise the sense in which A' is strictly more complicated than A. In the proof of the main theorem about the jump operation, the following fact will be useful: if a set A is r.e. in B and $B \leq_T C$ then also A is r.e. in C. Indeed, to say that A is r.e. in B means that $A = dom(f)$ for some $f \leq_T B$. By transitivity of $\leq_T$, we also find $f \leq_T C$, and hence A is the domain of some C-recursive function f.

The following theorem connects the jump operation, Turing reducibility and m-reducibility:

Theorem 4.3.16 (Jump theorem)**.**

(i) *A is r.e. in B if and only if $A \leq_m B'$*

(ii) *$A \leq_T B$ if and only if $A' \leq_m B'$*

Proof. Part *(i)*: assume first that A is r.e. in B. Now B' is not only B-r.e., but it is m-complete in the class of B-r.e. sets. (This is just the statement that K is m-complete, relativised to B.) Therefore $A \leq_m B'$. Conversely, if $A \leq_m B'$ then, since B' is r.e. in B, so is A. (This is just the fact that if V is r.e. and $U \leq_m V$ then also U is r.e., relativised to B.)

Part *(ii)*: if $A \leq_T B$ then, since A' is r.e. in A, we have that A' is also r.e. in B. By part *(i)* it follows that $A' \leq_m B'$, and hence $A' \leq_T B'$.

Conversely, if $A' \leq_m B'$, first note that $A \leq_m A'$, so that $A \leq_m B'$. By part *(i)*, this implies that A is r.e. in B. But we also have that $\mathbb{N} - A \leq_m A'$: since A is recursive in A, so is its complement $\mathbb{N} - A$ (relativisation of the fact that recursive sets are closed under taking complements). If $\mathbb{N} - A$ is recursive in A, then it is also r.e. in A, which, by part *(i)* again, gives that $\mathbb{N} - A \leq_m A'$.

Now by transitivity of $\leq_m$, we find that $\mathbb{N} - A \leq_m B'$, whence, again by *(i)*, $\mathbb{N} - A$ is also r.e. in B.

We have proved that both A and $\mathbb{N} - A$ are r.e. in B. By the relativised complementation theorem, this means that A is actually recursive in B, i.e. that $A \leq_T B$. □

There are a few important consequences: first, the operation $A \mapsto A'$ is well-defined and monotone on degrees, i.e. if $A \leq_T B$ then $A' \leq_T B'$. Moreover, A' always has strictly higher degree than A. This proves at once:

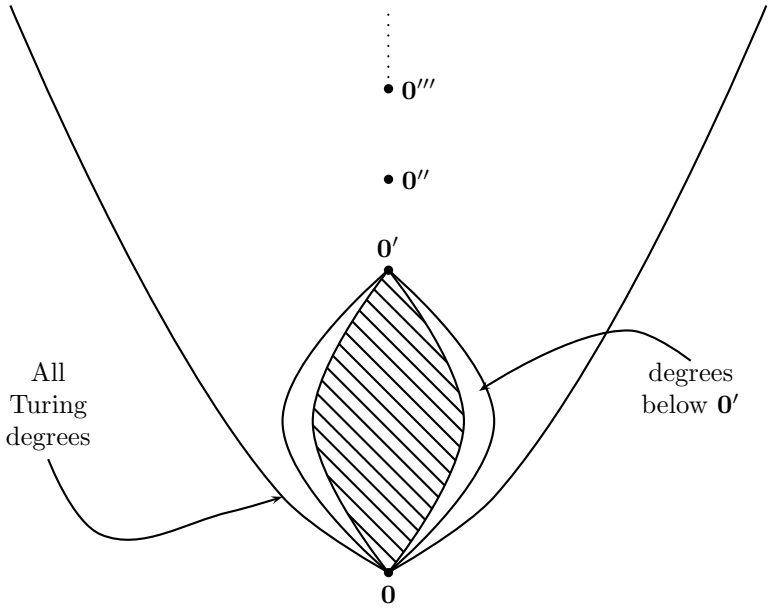


Figure 4.5: The semilattice of Turing degrees, with r.e. degrees highlighted

Proposition 4.3.17. *The semilattice $\mathcal{D}_T$ has no maximal elements.*

We usually write $A^{(n)}$ for the n-th iterate of the jump of A. Thus $A^{(0)} = A, A^{(n+1)} = A^{(n)\prime}$. It is also common to write $\mathbf{0}$ for the degree of the recursive sets, $\mathbf{0}'$ for the degree of $K = \emptyset'$, etc.

Finally, let us say that a degree is r.e. if it contains an r.e. set. At first one might be tempted to follow the analogy with m-degrees and say that a degree is r.e. if it contains only r.e. sets, but that is not correct: the T-degrees are closed under taking complements, and the r.e. sets are not. For example, the degree $\mathbf{0}'$ of the Halting set is r.e. but contains the non-r.e. set $\mathbb{N} - K$. Also note that if $A \leq_T K$ it does not follow that A is r.e. Thus there are degrees below $\mathbf{0}'$ which are not r.e. (at the moment we don't have an example of such a set A). The bottom part of the poset $\mathcal{D}_T$ is sketched in Figure 4.3.3.

4.4 The Arithmetical Hierarchy

While reducibility relations aim at organizing the world of unsolvable problems by imposing a criterion for comparing them, there is another very useful approach, namely via the use of hierarchies. When one defines a hierarchy, one begins with a well-defined class of sets which is considered elementary. From there, one gradually considers more and more complicated sets by allowing certain constructions.

In this section we consider the most common hierarchy in recursion theory, namely the Arithmetical Hierarchy. Its name derives from the fact that the sets in this hierarchy are precisely those which are definable in Peano Arithmetic. This is one of many interesting connections between recursion theory and neighboring branches of mathematical logic.

4.4.1 Definition and first properties

We will define, for each $n \in \mathbb{N}$, three classes of sets (relations), denoted $\Sigma_n, \Pi_n, \Delta_n$. This is done by induction on n.

Definition 4.4.1 (Arithmetical hierarchy)**.**

(i) We let $\Sigma_0 = \Pi_0 = \{\text{all recursive relations}\}$.

(ii) Inductively, a relation $A \subseteq \mathbb{N}^k$ is in Σ_{n+1} if there is a relation $B \subseteq \mathbb{N}^{k+1}$ in Π_n for which
$$\mathbf{x} \in A \Leftrightarrow \exists y.B(\mathbf{x}, y).$$

(iii) Dually, a relation $A \subseteq \mathbb{N}^k$ is in Π_{n+1} if there is a relation $B \subseteq \mathbb{N}^{k+1}$ in Σ_n for which
$$\mathbf{x} \in A \Leftrightarrow \forall y.B(\mathbf{x}, y).$$

(iv) A relation A is in Δ_n if it is both in Σ_n and in Π_n.

(v) A relation is *arithmetical* if it is in Σ_n for some n.

One should think back to the result that existential quantification (projection) over a recursive set yields an r.e. set: in terms of the classes just defined, this says that any r.e. set is in the class Σ_1. Moreover, since we showed that any r.e. set may be expressed as the projection of a recursive relation, we find

$$\Sigma_1 = \{\text{all r.e. relations}\}.$$

For an example of a Π_1 relation, consider the set $\mathsf{Empty} = \{x | W_x = \emptyset\}$ of codes of functions with empty domain. We have

$$e \in \mathsf{Empty} \Leftrightarrow \forall y.\neg T(e, (y)_1, (y)_2).$$

Since the part $\neg T(e, (y)_1, (y)_2)$ is recursive and hence in Σ_0, it follows that $\mathsf{Empty} \in \Pi_1$.

Note that a relation A is in Π_1 precisely when it is the complement of an r.e. set. Indeed, suppose that $x \in A \Leftrightarrow \forall y.B(x, y)$ for some recursive B. Then we also have, by plain predicate logic, that $x \in A \Leftrightarrow \neg\exists y \neg B(x, y)$, hence $x \in \mathbb{N} - A \Leftrightarrow \exists y \neg B(x, y)$, which expresses the complement $\mathbb{N} - A$ as the projection of the recursive relation $\neg B$. The converse is similar.

We may now deduce that

$$\Delta_1 = \{\text{all recursive relations}\}.$$

For, $\Delta_1 = \Sigma_1 \cap \Pi_1$ by definition, and a set is r.e. and co-r.e. precisely when it is recursive (by Post's Complementation Theorem).

For another example, consider $\mathsf{Tot} = \{x | \phi_x \text{ is total}\}$. We have

$$e \in \mathsf{Tot} \Leftrightarrow \forall x \exists y. T(e, x, y).$$

Since the relation $\exists y. T(e, x, y)$ is r.e. and hence Σ_1 (it is the domain of the universal function) this proves that $\mathsf{Tot} \in \Pi_2$.

The following lemma is easily proved by induction on n:

Lemma 4.4.2. *A relation A is Σ_n if it is of the form*

$$x \in A \Leftrightarrow \exists x_1 \forall x_2 \cdots Q x_n. R(x, x_1, \ldots, x_n)$$

where Q is either $\forall$ (if n is even) or $\exists$ (if n is odd). Dually, a relation A is Π_n if it is of the form

$$x \in A \Leftrightarrow \forall x_1 \exists x_2 \cdots Q x_n. R(x, x_1, \ldots, x_n)$$

where Q is either $\exists$ (if n is even) or $\forall$ (if n is odd).

As a consequence, we have:

Lemma 4.4.3. *A relation is Π_n if and only if it is the complement of a relation in Σ_n.*

The word "hierarchy" suggests that as we move up, our classes get increasingly large. This is confirmed by the following results.

Lemma 4.4.4. *For each n, we have $\Sigma_n \subseteq \Pi_{n+1}$ and $\Pi_n \subseteq \Sigma_{n+1}$.*

Proof. Let A in Σ_n, and consider the set $A \times \mathbb{N} = \{(x, z) | x \in A\}$. Then $A \times \mathbb{N}$ is again in Σ_n: if B is π_{n-1} such that $x \in A \Leftrightarrow \exists y. B(x, y)$ then also $(x, z) \in A \times \mathbb{N} \Leftrightarrow \exists y. B(x, y)$.

Now observe that $x \in A \Leftrightarrow \forall y. (x, y) \in A \times \mathbb{N}$. This shows that A is Π_{n+1}. The dual statement is analogous. □

But more is true: we have that $\Sigma_n \subseteq \Sigma_{n+1}$ and $\Pi_n \subseteq \Pi_{n+1}$. This is easily proved by induction on n, and left as an exercise to the reader. We conclude:

Proposition 4.4.5. *For each n, we have inclusions $\Sigma_n \subseteq \Delta_{n+1}$ and $\Pi_n \subseteq \Delta_{n+1}$.*

A sketch of the arithmetical hierarchy can be found in Figure 4.6.

In section 4.4.3 we shall prove that the inclusions in the hierarchy are strict. For now, we conclude with some closure properties of the classes Σ_n and Π_n.

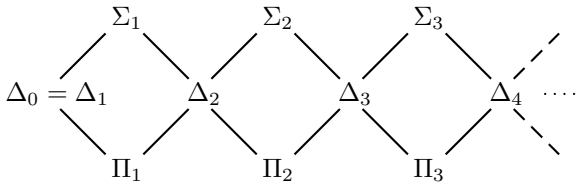


Figure 4.6: The Arithmetical Hierarchy

Lemma 4.4.6. *The classes Σ_n and Π_n are closed under binary unions and intersections.*

Proof. We prove closure under intersections using lemma 4.4.2. Suppose that $A, B \in \Sigma_n$, i.e. that there are recursive R, S such that

$$A = \{x | \exists x_1 \forall x_2 \cdots . R(x, x_1, \ldots, x_n)\} \qquad B = \{x | \exists x_1 \forall x_2 \cdots . S(x, x_1, \ldots, x_n)\}.$$

Then

$$\begin{aligned} A \cap B &= \{x | \exists x_1 \forall x_2 \cdots . R(x, x_1, \ldots, x_n) \wedge \exists y_1 \forall y_2 \cdots . S(x, y_1, \ldots, y_n)\} \\ &= \{x | \exists x_1 \exists y_1 \forall x_2 \forall y_2 \cdots . R(x, x_1, \ldots, x_n) \wedge S(x, y_1, \ldots, y_n)\} \\ &= \{x | \exists u_1 \forall v_1 \cdots . R(x, (u_1)_1, \ldots, (u_n)_1) \wedge S(x, (u_1)_2, \ldots, (u_n)_2)\}. \end{aligned}$$

In the second step, we have used the fact (from ordinary predicate logic) that $\exists v(\phi \wedge \psi) \leftrightarrow (\exists v.\phi) \wedge \psi$ if v is not free in ψ. In the last step we have used *quantifier contraction*, a trick to combine two similar quantifiers into one.

The remaining cases are left as an exercise. □

The idea of quantifier contraction can also be used to show the following:

Lemma 4.4.7. *For $n > 0$, the classes Σ_n are closed under existential quantification, and Π_n are closed under universal quantification.*

Note that this generalizes the fact that the r.e. sets are closed under projections (existential quantification).

Finally, we show that Σ_n and Π_n are closed under bounded quantification.

Lemma 4.4.8. *The classes Σ_n and Π_n are closed under bounded quantification.*

Proof. Let us look at the case of a set $A(x, y)$ in Σ_1; this is of the form

$$(x, y) \in A \Leftrightarrow \exists z.R(x, y, z)$$

where R is recursive. Define

$$B = \{(x, y) | \forall w \leq y. A(x, w)\}$$

we must verify that B is again Σ_1. Note that $(x, y) \in B$ if and only if $\forall w \leq y \exists z. R(x, w, z)$. That means that, given x, we have a finite function σ, which is defined on all $w \leq y$ and outputs a z for which $R(x, w, z)$. Thus we may reformulate:

$$(x, y) \in B \Leftrightarrow \exists \sigma \forall w \leq y. R(x, w, \sigma(w)).$$

Now σ may be encoded as a single number $[\sigma(0), \ldots, \sigma(y)]$, and therefore

$$(x, y) \in B \Leftrightarrow \exists u. (lh(u) = y + 1 \wedge \forall w \leq y. R(x, w, (u)_w)).$$

The part inside the existential quantifier is recursive, hence B is r.e.

Now this idea of permuting a bounded universal quantifier and an existential quantifier may be repeated; when we combine it with the fact (from plain predicate logic) that bounded universal quantifiers commute with universal quantifiers, the result follows by induction on the quantifier depth. □

4.4.2 The Tarksi-Kuratowski algorithm

Lemma 4.4.2 suggests a way of proving that a given set is in a certain level of the hierarchy: try to express the given set in the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n. R(x, x_1, \ldots, x_n)$$

where R is recursive and where each Q_i is either $\forall$ or $\exists$.

There is indeed a procedure which purports to do exactly this, and it is called the *Tarski-Kuratowski algorithm*. Given a set A (or a relation), we do the following:

- Write A as a logical combination of recursive predicates (using the logical connectives $\wedge, \vee, \longrightarrow, \neg$ and the quantifiers $\exists, \forall$).
- Bring the resulting expression in *prenex normal form* (PNF), where a formula P is in PNF if it is of the form

  $$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n. R \tag{4.4}$$

  where R is quantifier-free. The part R is sometimes called the *matrix* of the expression.
- Count the number k of *quantifier alternations*; if $Q_1 = \exists$, then $A \in \Sigma_{k+1}$; if $Q_1 = \forall$, then $A \in \Pi_{k+1}$. If the formula is quantifier-free, then A is recursive, i.e. $A \in \Sigma_0 = \Pi_0$.

A few remarks are in order. As for the first step, note that this may or may not be possible (some sets simply cannot be expressed in this way). Indeed, this is the main difficulty when using the algorithm in practice: to find a suitable expression of the set at hand. Moreover, there is nothing algorithmic about

$(\exists x.\phi) \wedge \psi \longleftrightarrow \exists x.(\phi \wedge \psi)$	if x not free in ψ.
$(\forall x.\phi) \wedge \psi \longleftrightarrow \forall x.(\phi \wedge \psi)$	if x not free in ψ.
$(\exists x.\phi) \vee \psi \longleftrightarrow \exists x.(\phi \vee \psi)$	if x not free in ψ.
$(\forall x.\phi) \vee \psi \longleftrightarrow \forall x.(\phi \vee \psi)$	if x not free in ψ.
$(\exists x.\phi) \longrightarrow \psi \longleftrightarrow \forall x.(\phi \longrightarrow \psi)$	if x not free in ψ.
$(\forall x.\phi) \longrightarrow \psi \longleftrightarrow \exists x.(\phi \longrightarrow \psi)$	if x not free in ψ.
$\forall x.\neg\phi \longleftrightarrow \neg\exists x.\phi$	
$\exists x.\neg\phi \longleftrightarrow \neg\forall x.\phi$	
$A \longrightarrow B \longleftrightarrow \neg A \vee B$	

Table 4.2: Rules for bringing a formula in PNF

this step! We typically need mathematical insight to come up with a good reformulation.

As for the second step, this is of course based on the rules of predicate logic, in particular the manipulation of quantifiers. For convenience, we have listed some useful rules of predicate logic in Table 4.2.

In applying these rules, it is understood that one avoids variable clashes by renaming bound variables as necessary.

Third, when we have our expression in PNF, then we count the number of alternations of quantifiers, because adjacent quantifiers of the same type may be contracted (as in Lemma 4.4.7). Then the conclusion of the algorithm is correct by Lemma 4.4.2.

Fourth, in some texts one asks not just for a recursive matrix R, but for a *primitive recursive* matrix. The reason is that, from the point of view of arithmetic, the primitive recursive predicates are better behaved. That this is not a restriction at all, follows from the Normal form theorem for recursive sets: every recursive set A may be written as a projection of the (primitive recursive) T-predicate. In practice, one often uses the T-predicate in the matrix anyway.

Let us look at a few examples to see the algorithm in action.

Examples 4.4.9.

(i) Consider the set $\mathsf{Con} = \{x | \phi_x \text{ codes a constant function}\}$. (With the understanding that a constant function may be partial.) First, write this in logical form:

$$\begin{aligned} e \in \mathsf{Con} &\Leftrightarrow \forall x \forall x'.(\phi_e(x)\downarrow \wedge \phi_e(x')\downarrow \longrightarrow \phi_e(x) = \phi_e(x')\downarrow \\ &\Leftrightarrow \forall x \forall x' \forall y \forall y'.(T(e,x,y) \wedge T(e,x',y')) \longrightarrow U(y) = U(y'). \end{aligned}$$

This is the correct format since T and U are recursive.

Next, this formula is already in PNF, so it remains to observe that there is no quantifier alternation and that the formula starts with a universal quantifier. Hence we conclude that $\mathsf{Con} \in \Pi_1$.

(ii) Consider $\mathsf{Inf} = \{x | W_x \text{ is infinite}\}$. This may be written as

$$\begin{aligned} e \in \mathsf{Inf} &\Leftrightarrow \forall x \in W_e \exists y \in W_e . x < y \\ &\Leftrightarrow \forall x (\exists z . T(e,x,z) \longrightarrow \exists y \exists t . T(e,y,t) \wedge x < y). \end{aligned}$$

Bring in PNF:

$$\begin{aligned} e \in \mathsf{Inf} &\Leftrightarrow \forall x (\exists z . T(e,x,z) \longrightarrow \exists y \exists t . T(e,y,t) \wedge x < y) \\ &\Leftrightarrow \forall x \forall z . (T(e,x,z) \longrightarrow \exists y \exists t . T(e,y,t) \wedge x < y) \\ &\Leftrightarrow \forall x \forall z . (\neg T(e,x,z) \vee \exists y \exists t . T(e,y,t) \wedge x < y) \\ &\Leftrightarrow \forall x \forall z \exists y \exists t . (\neg T(e,x,z) \vee (T(e,y,t) \wedge x < y)). \end{aligned}$$

Again, we have one quantifier alternation, and the formula starts with $\forall$, hence $\mathsf{Inf} \in \Pi_2$.

(iii) Consider $\mathrm{Fin} = \{x | W_x \text{ is finite}\}$. We have:

$$\begin{aligned} e \in \mathsf{Fin} &\Leftrightarrow \exists n \forall x > n . \phi_e(x) \uparrow \\ &\Leftrightarrow \exists n \forall x . (x > n \longrightarrow \neg \exists y . T(e,x,y)). \end{aligned} \tag{4.5}$$

Bring into PNF:

$$\begin{aligned} \exists n \forall x . (x > n \longrightarrow \neg \exists y . T(e,x,y)) &\Leftrightarrow \exists n \forall x . (x > n \longrightarrow \forall y . \neg T(e,x,y)) \\ &\Leftrightarrow \exists n \forall x . (x \leq n \vee \forall y \neg T(e,x,y)) \\ &\Leftrightarrow \exists n \forall x \forall y . (x \leq n \vee \neg T(e,x,y)) \end{aligned}$$

Thus $\mathsf{Fin} \in \Sigma_2$.

Of course, we could also have deduced the conclusion of the last example from the second, since Fin and Inf are complementary.

A few more remarks: we have asked that the matrix R be quantifier free, but we could allow for bounded quantifiers in the matrix (since the recursive relations are closed under these). Alternatively, we may simply state that in determining the complexity of a set, we may ignore bounded quantifiers.

Second, note that the algorithm does not necessary give an optimal answer: indeed, sometimes different ways of bringing a formula in PNF give rise to different conclusions. For example, look at $A = \{e | \phi_e \text{ total and bounded}\}$. We have

$$\begin{aligned} e \in A &\Leftrightarrow [\forall x \exists z . T(e,x,z)] \wedge [\exists n \forall x \forall z . (T(e,x,z) \longrightarrow U(z) \leq n)] \\ &\Leftrightarrow [\forall x \exists z . T(e,x,z)] \wedge [\exists n \forall u \forall v . (T(e,u,v) \longrightarrow U(v) \leq n)] \\ &\Leftrightarrow \exists n \forall u \forall v . [(\forall x \exists z . T(e,x,z)) \wedge (T(e,u,v) \longrightarrow U(v) \leq n)] \\ &\Leftrightarrow \exists n \forall u \forall v \forall x \exists z . [T(e,x,z) \wedge (T(e,u,v) \longrightarrow U(v) \leq n)] \end{aligned}$$

which shows that A is Σ_3. But on the other hand we could write

$$\begin{aligned} e \in A &\Leftrightarrow [\forall x \exists z.T(e,x,z)] \wedge [\exists n \forall x \forall z.(T(e,x,z) \longrightarrow U(y) \leq n)] \\ &\Leftrightarrow [\forall x \exists z.T(e,x,z)] \wedge [\exists n \forall u \forall v.(T(e,u,v) \longrightarrow U(v) \leq n)] \\ &\Leftrightarrow \forall x \exists z.[T(e,x,z) \wedge (\exists n \forall u \forall v.(T(e,u,v) \longrightarrow U(v) \leq n))] \\ &\Leftrightarrow \forall x \exists z \exists n \forall u \forall v.[T(e,x,z) \wedge (T(e,u,v) \longrightarrow U(v) \leq n)] \end{aligned}$$

showing that A is Π_3. We may thus conclude that $A \in \Delta_3$.

Note the following features of the algorithm:

- It does not necessarily give the sharpest possible results
- It only classifies sets as belonging to Σ_n or Π_n, but not as Δ_n.
- It cannot be used to show that a set does *not* belong to a certain class.

4.4.3 The Hierarchy theorem

We now wish to prove that the arithmetical hierarchy does not collapse, i.e. that for each n we have that the inclusions $\Sigma_n \subseteq \Delta_{n+1}$ and $\Pi_n \subseteq \Delta_{n+1}$ are strict. This will be achieved by exhibiting, for each $n > 0$, a relation $A \in \Sigma_n - \Pi_n$. There are various possibilities for such a set, but it is best to think first to the case Σ_1, where we already know a good candidate: $K \in \Sigma_1$ but $K \notin \Pi_1$. Of course, this also gives that $\mathbb{N} - K \in \Pi_1 - \Sigma_1$.

We thus need a set in Σ_n which plays a similar role in the class Σ_n as K plays for the r.e. sets. But we know which role that is: K is m-complete in Σ_1.

By the above reasoning, we are led to the following generalization of the notion of m-completeness:

Definition 4.4.10 (Complete sets in the arithmetic hierarchy)**.** For each $n > 0$, say that a set A is Σ_n-*complete* if

- $A \in \Sigma_n$
- for each $B \in \Sigma_n$ we have $B \leq_m A$.

Similarly, one defines Π_n-completeness.

That it is not unreasonable to consider m-reducibility in the context of the arithmetic hierarchy is supported by the following proposition:

Proposition 4.4.11. *Let $n > 0$, let $A \in \Sigma_n$ and suppose that $B \leq_m A$. Then also $B \in \Sigma_n$. Similarly, if $A \in \Pi_n$ and $B \leq_m A$ then $B \in \Pi_n$.*

Since in general the elements of Σ_n are relations, we have to use a generalization of m-reducibility (see also exercise 4.1): for relations $A \subseteq \mathbb{N}^k, B \subseteq \mathbb{N}^m$,

say $B \leq_m A$ if there exists a k-tuple of total recursive functions $f_1, \ldots, f_k$ such that

$$(x_1 \ldots, x_m) \in B \Leftrightarrow (f_1(x_1, \ldots, x_m), \ldots, f_k(x_1, \ldots, x_m)) \in A.$$

With that understanding, we can prove the result as follows.

Proof. Induction on n. For $n = 1$ this is the statement that if A is r.e. and $B \leq_m A$ then B is also r.e., which has been proved in Lemma 4.1.5 (and can be easily generalized to relations). We thus assume we have proved the statement up to n, and consider $A \in \Sigma_{n+1}$ and $B \leq_m A$. Write A as $A = \{x | \exists y. R(x, y)\}$ for some $R \in \Pi_n$, and let f be such that $x \in B \Leftrightarrow f(x) \in A$. Then

$$x \in B \Leftrightarrow \exists y. R(f(x), y).$$

The relation $S(x, y) \Leftrightarrow R(f(x), y)$ m-reduces to $R(x, y)$ since

$$(x, y) \in S \Leftrightarrow (f(x), y) \in R.$$

By IH for the class Π_n, this implies that $S \in \Pi_n$. Thus $B \in \Sigma_n$ as required. □

This proposition is extremely useful, because it often reduces the task of showing that a certain set is in Σ_n (Π_n) by showing that it reduces to a set which we already know to be in Σ_n (Π_n). After we prove the hierarchy theorem, we shall illustrate this with examples.

The hierarchy theorem will be an immediate consequence of the following theorem, due to Post. Not only does it exhibit sets which are m-complete in a given level of the hierarchy, it also draws an valuable connection between the arithmetic hierarchy and the jump operation, thereby bringing, perhaps unexpectedly, relative computability into the picture.

One part of the proof relies on a certain feature of oracle computations which we have not yet discussed; the reader is adviced to skip this part on first reading.

Theorem 4.4.12 (Post's Theorem)**.** *Let $n \in \mathbb{N}$.*

(i) *$A \in \Sigma_{n+1} \Leftrightarrow A$ is r.e. in some Σ_n set $\Leftrightarrow A$ is r.e. in some Π_n set.*

(ii) *The set $\emptyset^{(n)}$ (jump n times) is Σ_n-complete. Dually, the set $\mathbb{N} - \emptyset^{(n)}$ is Π_n-complete.*

(iii) *$A \in \Delta_{n+1}$ if and only if $A \leq_T B$ for some $B \in \Pi_n$ or $B \in \Sigma_n$.*

(iv) *$A \in \Delta_{n+1}$ if and only if $A \leq_T \emptyset^{(n)}$.*

Proof.

(i) If $A \in \Sigma_{n+1}$ then it is of the form $A = \{x | \exists y. R(x, y)\}$ for some R in Π_n. Thus A is r.e. in some Π_n set. But then A is also r.e. in the complement of R, which is in Σ_n.

Conversely, if A is r.e. in some Π_n set B then it is the domain of some B-recursive function ϕ_e^B. Thus $x \in A$ if and only if $\phi_e^B(x)\downarrow$, if and only if, for some finite initial segment $\sigma \subset B$, we have $\exists s.T^\sigma(e,x,s)$. Since σ is finite, the part $T^\sigma(e,x,s)$ is recursive; moreover, the part $\sigma \subset B$ may be reexpressed as

$$\begin{aligned} \sigma \subset B &\Leftrightarrow \forall i \leq lh(\sigma).[(\sigma)_i = \chi_B(i)] \\ &\Leftrightarrow \forall i \leq lh(\sigma).[((\sigma)_i = 0 \wedge i \in B) \vee ((\sigma)_i = 1 \wedge i \notin B)] \end{aligned}$$

Now the part $(\sigma)_i = 0 \wedge i \in B$ is Π_n since $B \in \Pi_n$; and the part $(\sigma)_i = 1 \wedge i \notin B$ is in Σ_n, since $\mathbb{N} - B \in \Sigma_n$. Thus both parts are in Σ_{n+1}, and hence the part $\sigma \subset B$ is Σ_{n+1}. Finally, we find that the expression $\exists\sigma\exists s.(\sigma \subset B \wedge T^\sigma(e,x,s))$ is Σ_{n+1}, as required.

(ii) Induction on n. For $n = 1$, this is just the m-completeness of the Halting set K. For $n+1$, observe that $\emptyset^{(n+1)} = (\emptyset^{(n)})'$ is the jump of $\emptyset^{(n)}$, which by IH is Σ_n-complete. In particular, $\emptyset^{(n)} \in \Sigma_n$; since $\emptyset^{(n+1)}$ is r.e. in $\emptyset^{(n)}$, it follows by part *(i)* that $\emptyset^{(n+1)} \in \Sigma_{n+1}$. Moreover, given any other set $A \in \Sigma_{n+1}$, we have, again by part *(i)*, that A is r.e. in some Σ_n-set B. By IH, we have $B \leq_m \emptyset^{(n)}$. Thus A is also r.e. in $\emptyset^{(n)}$, and by the Jump-theorem this gives that $A \leq_m \emptyset^{(n+1)}$.

(iii) If $A \in \Delta_{n+1}$ then by definition $A \in \Sigma_{n+1}$ and $\mathbb{N} - A \in \Sigma_{n+1}$. By part *(ii)*, this means that both A and $\mathbb{N} - A$ are r.e. in the complete set $\emptyset^{(n)}$. By the relative version of the complementation theorem this implies that A is recursive in $\emptyset^{(n)}$.

Conversely, if $A \leq_T B$ for some Σ_n set B then also $A \leq_T \emptyset^{(n)}$. Again by the relativised complementation theorem, this implies that both A and $\mathbb{N} - A$ are r.e. in $\emptyset^{(n)}$. Thus A is Σ_{n+1} and $\mathbb{N} - A$ is also Σ_{n+1}, and hence $A \in \Delta_{n+1}$.

(iv) This follows from an inspection of the proof of part *(iii)*.

□

Corollary 4.4.13 (Hierarchy Theorem)**.** *All inclusions in the Arithmetical Hierarchy are strict.*

Proof. We have shown that $\emptyset^{(n+1)}$ is Σ_{n+1}-complete. It suffices to show that $\emptyset^{(n+1)} \notin \Pi_{n+1}$. If it were, then $\emptyset^{(n+1)} \in \Delta_{n+1}$. By part *(iv)* of the theorem, this would imply that $\emptyset^{(n+1)} \leq_T \emptyset^{(n)}$, which contradicts the Jump theorem. □

We end this section with an application of these results. We have seen before that Tot is in Π_2. However, we haven't proved yet that this is the best possible classification. We shall now show that Tot is *strictly* Π_2; in fact, we shall show that Tot is Π_2-complete. It turns out that many index sets, such as $\mathsf{Fin}, \mathsf{Cof}, \mathsf{Rec}$ (see Figure 4.1), are all complete on a certain level of the

arithmetical hierarchy. For the easier ones, classifications can often done by hand. For the harder ones one needs more advanced machinery.

We shall prove that given any Π_2 set A, we have $A \leq_m \mathsf{Tot}$. First write $A = \{z | \forall x \exists y. R(x, y, z)\}$ for some recursive relation R. Then define

$$f(z) \bullet x = \mu y. R(x, y, z).$$

It now easily follows that $z \in B \Leftrightarrow f(z) \in \mathsf{Tot}$.

4.5 Exercises

M-degrees

Exercise 4.1 (*Reduction of relations)**.** Define m-reducibility for arbitrary relations (subsets of $\mathbb{N}^k$) and show that it agrees with reducibility for sets when we use pairing to encode relations as sets.

Exercise 4.2 (**Recursive 1-degrees)**.** Completely describe the partially ordered set of recursive 1-degrees.

Exercise 4.3 (**Another m-complete set)**.** Show that $K_1 \equiv_1 K$. (See Table 4.1 for the definition.)

Exercise 4.4 (*)**.** Show that if $A \leq_m B$ then $\mathbb{N} - A \leq_m \mathbb{N} - B$. Also show that in general $A \leq_m \mathbb{N} - A$ fails.

Exercise 4.5 (***)**.** Show directly that $\mathsf{Inf} \equiv_m \mathsf{Tot}$.

Exercise 4.6 (***)**.** Show directly that $\mathsf{Tot} \equiv_m \mathsf{Con}$.

Exercise 4.7 (***)**.** Give an example of a recursive function which cannot be extended to a total recursive function.

Exercise 4.8 (**)**.** Show directly that K_1 is creative.

Exercise 4.9 (*)**.** Show that a productive set is not r.e.; use this to show that Empty is not r.e.

Exercise 4.10 (***Rice's Theorem improved)**.** Prove that if A is a non-trivial index set, then either $K \leq_1 A$ or $\mathbb{N} - K \leq_1 A$. (Hint: pick a code e of the empty function. If $e \in A$, then show $K \leq_1 A$, otherwise show $\mathbb{N} - K \leq_1 A$.)

Exercise 4.11 (***Cylinders)**.** In this exercise we explore another connection between m-reduction and 1-reduction. We say that a set A is a *cylinder* if $B \leq_m A$ implies $B \leq_1 A$.

1. Show that any index set is a cylinder.
2. Show that any set of the form $A \otimes \mathbb{N}$ is a cylinder.
3. Show that A is a cylinder if and only if $A \equiv_1 B \otimes \mathbb{N}$ for some $\mathbb{N}$.

Exercise 4.12 (**)**.**

1. Prove that the intersection of two simple sets is again simple. Is the same true for unions?
2. Prove that if A is a simple set and W_e is infinite, then $A \cap W_e$ is also infinite.

Exercise 4.13 (*Effective simplicity)**.** A set A is called *effectively simple* if there is a recursive function such that for each $W_e \cap A = \emptyset$, the set W_e has at most $f(e)$ elements.

Prove that such a set is simple. Also show that Post's simple set and the simple set obtained via the priority argument are effectively simple.

Exercise 4.14 (**More effective simple sets)**.** Show that a set A is effectively simple if and only if there is a recursive function g such that

$$W_e \text{ infinite } \Rightarrow g(e) \in W_e \cap A.$$

Exercise 4.15 (***Effectively non-recursive sets)**.** A set A is called *effectively non-recursive* if it is r.e. and there is a recursive function f such that

$$f(x) \in A \Leftrightarrow f(x) \in W_x.$$

(Compare this with creativeness.)

- Prove that K is effectively non-recursive.
- Prove that if $A \leq_m B$ and A is effectively non-recursive, so is B.
- Prove that A is m-complete if and only if A is effectively non-recursive.

Exercise 4.16 (*Kolmogorov complexity)**.** Is the function $K(x)$ recursive?

Relative computability

Exercise 4.17 (*)**.** Prove lemma 4.3.9.

Exercise 4.18 (**)**.** State and prove a relative version of the Fixed point theorem.

Exercise 4.19 (**)**.** State and prove a relative version of the Second recursion theorem.

Exercise 4.20 (**)**.** Prove that given any Turing degree, there are at most countably many degrees below the given degree. Also prove that every degree contains at most countably many sets.

Exercise 4.21 (**Countable joins of degrees?)**.** Given a family of sets $\{A_n | n \in \mathbb{N}\}$ define

$$\oplus_{n\in\mathbb{N}} A_n = \{[x, n] | x \in A_n\}.$$

- Show that for each i we have $A_i \leq_T \oplus_n A_n$.
- Show that in general $\oplus_n A_n$ is not a least upper bound for the A_i. Hint: consider a set A which is not recursive, and put

$$A_n = \begin{cases} \mathbb{N} & \text{if } n \in \\ \emptyset & \text{otherwise.} \end{cases}$$

  What is the degree of each A_i? What is the degree of $\oplus_n A_n$?

Exercise 4.22 (**Jump)**.** Improve the Jump-theorem by showing that

$$A \leq_T B \Leftrightarrow A' \leq_1 B'.$$

Exercise 4.23 (**Infinitary jumps)**.** Let A be any set, and define

$$A^{(\omega)} = \oplus_n A^{(n)}.$$

(The notation refers to exercise 4.21.)

- Show that if $A \leq_T B$ then $A^{(\omega)} \leq_T B^{(\omega)}$. (Thus, the infinitary jump is well-defined on Turing degrees.)
- Show that the infinitary jump is not injective on degrees.

Exercise 4.24 (**Jumps and Joins)**.** Let A and B be sets. Prove that

$$A' \oplus B' \leq_T (A \oplus B)'.$$

Arithmetic hierarchy

Exercise 4.25 (**)**.** Complete the proof of Lemma 4.4.8.

Exercise 4.26 (**Tarski-Kuratowski algorithm)**.** Using the algorithm, show that the following sets are all Σ_3:

- Cof
- Rec
- Ext

Exercise 4.27 (**)**.** Show that the set $\mathsf{Comp} = \{x | W_x \text{is m-complete}\}$ is Σ_4.

Exercise 4.28 (***)**.** Show that the set $\{e | W_e \text{ is a singleton}\}$ is Δ_2. Also show that this is strict, i.e. that the set is not Σ_1 nor Π_1.

Exercise 4.29 (***)**.** Give an exact classification of the sets Con and Fin.

Exercise 4.30 (*)**.** Give an example of a set which is not arithmetical. Hint: exercise 4.23.

Exercise 4.31 (**)**.** Show that the sets $\emptyset^{(n)}$ are in fact 1-complete in Σ_n.

Exercise 4.32 (***)**.** Adapt the methods of section 4.1 to show that all Σ_n-complete sets are recursively isomorphic.

Index