5. Minimalisation

In the previous section we have seen that a large collection of functions can be shown to be computable using the operations of substitution and recursion, and operations derived from these. There is a third important operation which generates further computable functions, namely *unbounded minimalisation*, or just *minimalisation*, which we now describe.

Suppose that $f(\mathbf{x}, y)$ is a function (not necessarily total) and we wish to define a function $g(\mathbf{x})$ by

$$g(\mathbf{x}) = \text{the least } y \text{ such that } f(\mathbf{x}, y) = 0,$$

in such a way that if f is computable then so is g. Two problems can arise. First, for some $\boldsymbol{x}$ there may not be any y such that $f(\boldsymbol{x}, y)=0$. Second, assuming that f is computable, consider the following natural algorithm for computing $g(\boldsymbol{x})$. 'Compute $f(\boldsymbol{x}, 0), f(\boldsymbol{x}, 1), \ldots$ until y is found such that $f(\boldsymbol{x}, y)=0$'. This procedure may not terminate if f is not total, even if such a y exists; for instance, if $f(\boldsymbol{x}, 0)$ is undefined but $f(\boldsymbol{x}, 1)=0$.

Thus we are led to the following definition of the *minimalisation operator* μ, which yields computable functions from computable functions.

5.1. *Definition*

For any function $f(\boldsymbol{x}, y)$

$$\mu y(f(\boldsymbol{x}, y)=0)=\begin{cases}\text{the least } y \text{ such that} & \\ \qquad \text{(i) } f(\boldsymbol{x}, z) \text{ is defined, all } z \leq y, \text{ and} & \\ \qquad \text{(ii) } f(\boldsymbol{x}, y)=0, & \text{if such a } y \text{ exists,}\\ \text{undefined,} & \text{if there is no such } y.\end{cases}$$

$\mu y(\ldots)$ is read 'the least y such that ...'. This operator is sometimes called simply the *μ-operator.*

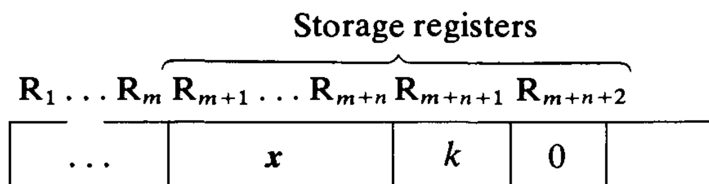
The next theorem shows that $\mathscr{C}$ is closed under minimalisation.

5.2. *Theorem*

Suppose that $f(\boldsymbol{x}, y)$ is computable; then so is the function $g(\boldsymbol{x})=\mu y(f(\boldsymbol{x}, y)=0)$.

Proof. Suppose that $\boldsymbol{x}=(x_1, \ldots, x_n)$ and that F is a program in standard form that computes the function $f(\boldsymbol{x}, y)$. Let $m=\max(n+1, \rho(F))$. We write a program G that embodies the natural algorithm for g: for $k=0, 1, 2 \ldots$, compute $f(\boldsymbol{x}, k)$ until a value of k is found such that $f(\boldsymbol{x}, k)=0$; this value of k is the required output.

The value of $\boldsymbol{x}$ and the current value of k will be stored in registers $R_{m+1}, \ldots, R_{m+n+1}$ before computing $f(\boldsymbol{x}, k)$: thus the typical configuration will be



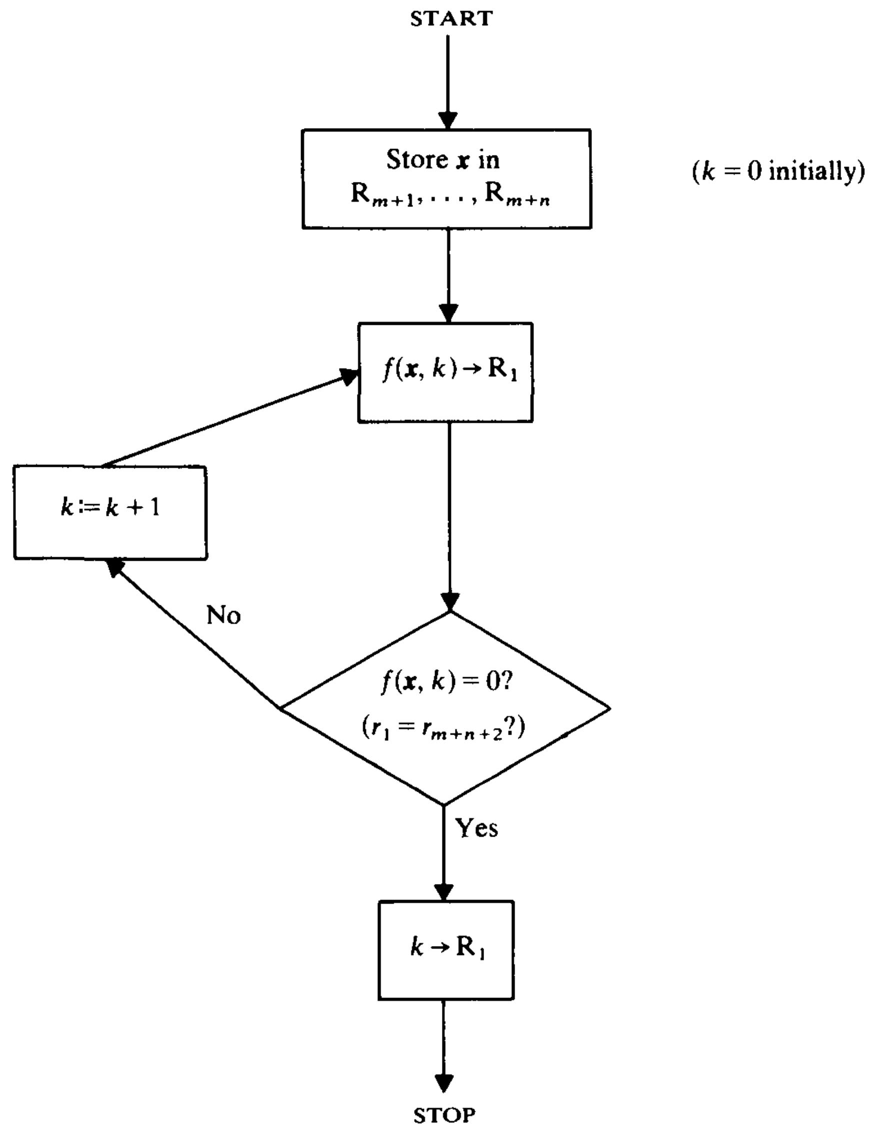
with $k=0$ initially. Note that r_{m+n+2} is always 0.

A flow diagram that carries out the above procedure for g is given in fig. 2d. This translates easily into the following program G for g:

$$
\begin{array}{ll}
 & T(1, m+1) \\
 & \vdots \\
 & T(n, m+n) \\
I_p & F[m+1, m+2, \ldots, m+n+1 \to 1] \\
 & J(1, m+n+2, q) \\
 & S(m+n+1) \\
 & J(1, 1, p) \\
I_q & T(m+n+1, 1)
\end{array}
$$

(I_p is the first instruction of the subroutine $F[m+1, m+2, \ldots \to 1]$.) □

Fig. 2d. Minimalisation (theorem 5.2).



5.3. *Corollary*

Suppose that $R(\mathbf{x}, y)$ is a decidable predicate; then the function

$$g(\mathbf{x}) = \mu y\, R(\mathbf{x}, y)$$
$$= \begin{cases} \textit{the least } y \textit{ such that } R(\mathbf{x}, y) \textit{ holds}, & \textit{if there is such a } y, \\ \textit{undefined} & \textit{otherwise,} \end{cases}$$

is computable

Proof. $g(\mathbf{x}) = \mu y(\overline{\mathrm{sg}}(c_R(\mathbf{x}, y)) = 0)$. □

In view of this corollary, the μ-operator is often called a *search operator.* Given a decidable predicate $R(\mathbf{x}, y)$ the function $g(\mathbf{x})$ searches for a y such that $R(\mathbf{x}, y)$ holds, and moreover, finds the least such y if there is one.

The μ-operator may generate a non-total computable function from a total computable function; for instance, putting $f(x, y) = |x - y^2|$, and $g(x) \simeq \mu y(f(x, y) = 0)$, we have that g is the non-total function

$$g(x) = \begin{cases} \sqrt{x} & \text{if } x \text{ is a perfect square,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Thus, in a trivial sense, using the μ-operator together with substitution and recursion, we can generate from the basic functions more functions than can be obtained using only substitution and recursion (since these operations always yield total functions from total functions). There are also, however, *total* functions for which the use of the μ-operator is essential. Example 5.5 below gives one such function; we present another example in chapter 5. Thus we see that, in a strong sense, minimalisation, unlike bounded minimalisation, cannot be defined in terms of substitution and recursion. It turns out, nevertheless, that most commonly occurring computable total functions can be built up from the basic functions using substitution and recursion only: such functions are called *primitive recursive,* and are discussed further in chapter 3 § 3. In practice, of course, we might establish the computability of these functions by what amounts to a non-essential use of minimalisation, if this makes the task easier.

5.4. *Exercises*

1. Suppose that $f(x)$ is a total injective computable function; prove that f^{-1} is computable.
2. Suppose that $p(x)$ is a polynomial with integer coefficients; show that the function

$f(a)$ = least non-negative integral root of $p(x) - a (a \in \mathbb{N})$ is computable ($f(a)$ is undefined if there is no such root).

3. Show that the function

$$f(x, y) = \begin{cases} x/y & \text{if } y \neq 0 \text{ and } y \mid x, \\ \text{undefined} & \text{otherwise,} \end{cases}$$

is computable.

We conclude this chapter with an example of a function that makes essential use of the μ-operator; it also shows how this operator can be used not only to search for a single number possessing a given property, but to search for finite sequences or sets of numbers, or other objects coded by a single number. The function is a modification by Péter of an example due to Ackermann, after whom it is named. It is rather more complicated than any function we have considered so far.

5.5. *Example* (The Ackermann function)

The function $\psi(x, y)$ given by the following equations is computable:

$$\psi(0, y) = y + 1,$$
$$\psi(x+1, 0) \simeq \psi(x, 1),$$
$$\psi(x+1, y+1) \simeq \psi(x, \psi(x+1, y)).$$

This definition involves a kind of double recursion that is stronger than the primitive recursion discussed in § 3. To see, nevertheless, that these equations do unambiguously define a function, notice that any value $\psi(x, y)$ $(x > 0)$ is defined in terms of 'earlier' values $\psi(x_1, y_1)$ with $x_1 < x$ or $x_1 = x$ and $y_1 < y$. In fact, $\psi(x, y)$ can be obtained by using only a *finite* number of such earlier values: this is easily established by induction on x and y. Hence ψ is computable in the informal sense. For instance, it is easy to calculate that $\psi(1, 1) = 3$ and $\psi(2, 1) = 5$.

To show rigorously that ψ is computable is quite difficult. We sketch a proof using the idea of a *suitable* set of triples S. The essential property of a suitable set S (defined below) is that if $(x, y, z) \in S$, then

(5.6) (i) $z = \psi(x, y)$,

(ii) S contains all the earlier triples $(x_1, y_1, \psi(x_1, y_1))$ that are needed to calculate $\psi(x, y)$.

Definition

A finite set of triples S is said to be *suitable* if the following conditions are satisfied:

(*a*) if $(0, y, z) \in S$ then $z = y+1$,

(*b*) if $(x+1, 0, z) \in S$ then $(x, 1, z) \in S$,

(*c*) if $(x+1, y+1, z) \in S$ then there is u such that $(x+1, y, u) \in S$ and $(x, u, z) \in S$.

These three conditions correspond to the three clauses in the definition of ψ: for instance, (*a*) corresponds to the statement: if $z = \psi(0, y)$, then $z = y+1$; (*c*) corresponds to the statement: if $z = \psi(x+1, y+1)$, then there is u such that $u = \psi(x+1, y)$ and $z = \psi(x, u)$.

The definition of a suitable set S ensures that (5.6) is satisfied. Moreover, for any particular pair of numbers (m, n) there is a suitable set S such that $(m, n, \psi(m, n)) \in S$; for example, let S be the set of triples $(x, y, \psi(x, y))$ that are used in the calculation of $\psi(m, n)$.

Now a triple (x, y, z) can be coded by the single positive number $u = 2^x 3^y 5^z$; a finite set of positive numbers $\{u_1, \ldots, u_k\}$ can be coded by the single number $p_{u_1} p_{u_2} \ldots p_{u_k}$. Hence a finite set of triples can be coded by a single number v say. Let S_v denote the set of triples coded by the number v. Then we have

$$(x, y, z) \in S_v \Leftrightarrow p_{2^x 3^y 5^z} \text{ divides } v,$$

so '$(x, y, z) \in S_v$' is a decidable predicate of x, y, z, v; and if it holds, then $x, y, z < v$. Hence, using the techniques and functions of earlier sections we can show that the following predicate is decidable:

$$R(x, y, v) \equiv \text{'}v \text{ is the code number of a suitable set of triples and } \exists z < v\ ((x, y, z) \in S_v)\text{.'}$$

Thus the function

$$f(x, y) = \mu v R(x, y, v)$$

is a computable function that searches for the code of a suitable set containing (x, y, z) for some z. Hence

$$\psi(x, y) = \mu z ((x, y, z) \in S_{f(x,y)})$$

which shows that ψ is computable.

A more sophisticated proof that ψ is computable will be given in chapter 10 as an application of more advanced theoretical results.

We do not prove here that ψ cannot be shown to be computable using substitution and recursion alone. This matter is further discussed in § 3 of the next chapter.

3
Other approaches to computability: Church's thesis

Over the past fifty years there have been many proposals for a precise mathematical characterisation of the intuitive idea of effective computability. The URM approach is one of the more recent of these. In this chapter we pause in our investigation of URM-computability itself to consider two related questions.

1. How do the many different approaches to the characterisation of computability compare with each other, and in particular with URM-computability?
2. How well do these approaches (particularly the URM approach) characterise the informal idea of effective computability?

The first question will be discussed in §§ 1–6; the second will be taken up in § 7. The reader interested only in the technical development of the theory in this book may omit §§ 3–6; none of the development in later chapters depends on these sections.

1. Other approaches to computability

The following are some of the alternative characterisations that have been proposed:

(*a*) *Gödel–Herbrand–Kleene* (1936). General recursive functions defined by means of an equation calculus. (Kleene [1952], Mendelson [1964].)

(*b*) *Church* (1936). λ-definable functions. (Church [1936] or [1941].)

(*c*) *Gödel–Kleene* (1936). μ-recursive functions and partial recursive functions (§ 2 of this chapter.).

(*d*) *Turing* (1936). Functions computable by finite machines known as Turing machines. (Turing [1936]; § 4 of this chapter.)

(*e*) *Post* (1943). Functions defined from canonical deduction systems. (Post [1943], Minsky [1967]; § 5 of this chapter.)

(*f*) *Markov* (1951). Functions given by certain algorithms over a finite alphabet. (Markov [1954], Mendelson [1964]; § 5 of this chapter.)

(*g*) *Shepherdson–Sturgis* (1963). URM-computable functions. (Shepherdson & Sturgis [1963].)

There is great diversity among these various approaches; each has its own rationale for being considered a plausible characterisation of computability. The remarkable result of investigation by many researchers is the following:

1.1. *The Fundamental result*

Each of the above proposals for a characterisation of the notion of effective computability gives rise to the same class of functions, the class that we have denoted $\mathscr{C}$.

Thus we have the simplest possible answer to the first question posed above. Before discussing the second question, we shall examine briefly the approaches of Gödel–Kleene, Turing, Post and Markov, mentioned above, and we will sketch some of the proofs of the equivalence of these with the URM approach. The reader interested to discover full details of these and other approaches, and proofs of all the equivalences in the Fundamental result, may consult the references indicated.

2. Partial recursive functions (Gödel–Kleene)

2.1. *Definition*

The class $\mathscr{R}$ of *partial recursive functions* is the smallest class of partial functions that contains the basic functions $\mathbf{0}$, $x+1$, U_i^n (lemma 2-1.1) and is closed under the operations of substitution, recursion and minimalisation. (Equivalently, $\mathscr{R}$ is the class of partial functions that can be built up from the basic functions by a finite number of operations of substitution, recursion or minimalisation.)

Note that in the definition of the class $\mathscr{R}$, no restriction is placed on the use of the μ-operator, so that $\mathscr{R}$ contains non-total functions. Gödel and Kleene originally confined their attention to *total* functions; the class of functions first considered was the class $\mathscr{R}_0$ of *μ-recursive functions*, defined like $\mathscr{R}$ above, except that applications of the μ-operator are allowed only if a *total* function results. Thus $\mathscr{R}_0$ is a class of total functions, and clearly $\mathscr{R}_0 \subseteq \mathscr{R}$. In fact, $\mathscr{R}_0$ contains all of the total

functions that are in $\mathcal{R}$, although this is not immediately obvious; see corollary 2.3 below for a proof. Hence $\mathcal{R}$ is a natural extension of $\mathcal{R}_0$ to a class of partial functions.

The term *recursive function* is used nowadays to describe μ-recursive functions; so a recursive function is always total – it is a totally defined partial recursive function. The term *general recursive* function is sometimes used to describe μ-recursive functions, although historically, this was the name Kleene gave to the total functions given by his equation calculus approach ((a) in § 1). It was Kleene who proved the equivalence of general recursive functions (given by the equation calculus) and μ-recursive functions.

We now outline a proof of

2.2. *Theorem*
$\mathcal{R} = \mathscr{C}$.

2.3. *Corollary*

Every total function in $\mathscr{R}$ belongs to $\mathscr{R}_0$.

A predicate $M(\mathbf{x})$ whose characteristic function c_M is recursive is called a *recursive predicate*. In view of theorem 2.2, a recursive predicate is the same as a decidable predicate.

3. A digression: the primitive recursive functions

This is a natural point to mention an important subclass of $\mathscr{R}$, the class of *primitive recursive functions*, although they do not form part of the main line of thought in this chapter. These functions were referred to in chapter 2 § 5.

3.1. *Definition*

(*a*) The class $\mathscr{PR}$ of *primitive recursive functions* is the smallest class of functions that contains the basic functions $\mathbf{0}$, $x+1$, U_i^n, and is closed under the operations of substitution and recursion.

(*b*) A *primitive recursive predicate* is one whose characteristic function is primitive recursive.

All of the particular computable functions obtained in §§ 1, 3, 4 of chapter 2 are primitive recursive, since minimalisation was not used there. We have already noted that the functions c and j used in the proof of theorem 2.2 are primitive recursive. Further, from theorems 2-4.10 and 2-4.12 we see that $\mathscr{PR}$ is closed under bounded sums and products, and under bounded minimalisation. Thus the class of primitive recursive functions is quite extensive.

There are nevertheless recursive functions (or, equivalently, total computable functions) that are not primitive recursive. Indeed, the Ackermann function ψ of example 2-5.5 was given as an instance of such a function. A detailed proof that the Ackermann function is not primitive recursive is rather lengthy, and we refer the reader to Péter [1967, chapter 9] or Mendelson [1964, p. 250, exercise 11]. Essentially one shows that ψ grows faster than any given primitive recursive function. (To see how fast ψ grows try to calculate a few simple values.)

In chapter 5 we will be able to give an example of a total computable (i.e. recursive) function that we shall prove is not primitive recursive.

Our conclusion is that although the primitive recursive functions form a natural and very extensive class, they do *not* include all computable functions and thus fall short as a possible characterisation of the informal notion of computability.