

Chapter 4

Recursive Enumerability

First of all, let's summarize some of the results of the preceding chapters, and establish the terminology that will be used henceforth.

We have seen that the class of general recursive partial functions is exactly the same as the class of register-machine computable partial functions. The fact that two such different approaches yield the same class of functions is evidence that we have here a "natural" class. The members of this class will be called *computable partial functions* (or *recursive partial functions*—the two names are both in common use). The adjective "partial" covers both the total and non-total functions; it can be omitted in cases where we know that the function is total. Church's thesis is the assertion that the concept of being a computable partial function is the correct formalization of the informal idea of being an effectively calculable partial function.

The class of computable partial functions includes all of the primitive recursive functions. Moreover, the class is closed under composition, the μ -operator, and definition by cases, among other things.

We define a relation R on \mathbb{N} to be a *computable relation* (or a *recursive relation*) if its characteristic function (which is always total) is a computable function. The class of computable relations includes all of the primitive recursive relations. Moreover, the class is closed under unions, intersections, complements, and bounded quantification. Church's thesis tells us that the concept of being a computable relation corresponds to the informal idea of being a decidable relation.

Moreover, we have found the following basic result.

Enumeration theorem: For each positive n , there is an $(n + 1)$ -place computable partial function $\Phi^{(n)}$ with the property that for every n -place computable partial function f there exists a number e such that

$$\Phi^{(n)}(e, \vec{x}) = f(\vec{x})$$

for all n -tuples \vec{x} , where '=' has the usual meaning—either both sides are undefined or they are both defined and are the same.

Consequently, we can let $\llbracket e \rrbracket^{(n)}$ be the n -place partial function defined by the equation

$$\llbracket e \rrbracket^{(n)}(\vec{x}) = \Phi^{(n)}(e, \vec{x})$$

and obtain a complete enumeration (with repetitions)

$$\llbracket 0 \rrbracket^{(n)}, \quad \llbracket 1 \rrbracket^{(n)}, \quad \llbracket 2 \rrbracket^{(n)}, \quad \dots$$

of all of the n -place computable partial functions. When $n = 1$, we can omit the superscript.

The enumeration theorem lets us define the “diagonal” partial function

$$d(x) = \llbracket x \rrbracket(x) + 1.$$

This is a 1-place computable partial function; its domain is the set

$$K = \{x \mid \llbracket x \rrbracket(x) \downarrow\}.$$

Because the diagonal function d is a computable partial function, we know that $d = \llbracket e \rrbracket$ for some index e . That is, $\llbracket x \rrbracket(x) + 1 = d(x) = \llbracket e \rrbracket(x)$ for all x . In particular, taking $x = e$, we have $\llbracket e \rrbracket(e) + 1 = d(e) = \llbracket e \rrbracket(e)$. This looks like bad news; we *almost* have proved that $1 = 0$. But remember that ‘=’ means that either both sides are defined and equal, or both sides are undefined. Here it is the latter. A modification of the argument gives us the following result:

Theorem: The diagonal function d has no extension to a computable total function. (That is, there is no computable total function f with the property that whenever $d(x)$ is defined then $f(x) = d(x)$.)

Proof: Suppose that $\llbracket e \rrbracket$ is a computable partial function extending d . (Maybe it even *is* d .) We will show that $e \notin \text{dom } \llbracket e \rrbracket$, thereby showing that $\llbracket e \rrbracket$ is not total.

If to the contrary $e \in \text{dom } \llbracket e \rrbracket$, then $e \in K$ and $d(e) \downarrow$. But then we have

$$\begin{aligned} \llbracket e \rrbracket(e) &= d(e) && \text{because } \llbracket e \rrbracket \text{ extends } d \\ &= \llbracket e \rrbracket(e) + 1 && \text{by the definition of } d \end{aligned}$$

and these are defined. Hence $0 = 1$, a contradiction. \dashv

The same argument would apply to the function $\hat{d}(x) = 1 \div \llbracket x \rrbracket(x)$.

Corollary: The set K is not a computable set.

Proof: The function

$$f(x) = \begin{cases} d(x) & \text{if } x \in K \\ 0 & \text{if } x \notin K \end{cases}$$

is a total extension of d and therefore is not a computable function. But if K were a computable set, then f would have been a computable function (by definition by cases). \dashv

Unsolvability of the halting problem: The relation

$$H = \{\langle x, y \rangle \mid \llbracket x \rrbracket(y) \downarrow\}$$

is not a computable relation.

Proof: We have $x \in K \iff \langle x, x \rangle \in H$. \dashv

Thus the halting problem, despite being a precisely formulated problem, is unsolvable. We will see other such problems (i.e., other non-computable relations). Moreover, there are unsolvable problems in other parts of mathematics. In Chapter 5, we will see that problem of deciding, given a sentence in arithmetic, whether it is true or false, is unsolvable.

Digression: “Hilbert’s tenth problem” is the problem of deciding, given a polynomial equation in many variables with integer coefficients, whether or not it has solution in the integers. (For example, the equation $x^2 = 9y^2 + 18y + 28$ has the solution $x = 10, y = 2$, but the equation $x^2 = 9y^2 + 18y + 10$ has no solution in the integers.) It is now known (through work of Martin Davis, Yuri Matiyasevich, Hilary Putnam, and Julia Robinson) that this problem is unsolvable.

For a very different example, in symbolic logic, the problem of deciding, given a formula in symbolic logic, whether or not it is true under all interpretations of its symbols, is an unsolvable problem. This result is known as Church’s theorem.

In contrast to the halting problem, the ternary relation

$$\{\langle x, y, t \rangle \mid \llbracket x \rrbracket(y) \downarrow \text{ in } \leq t \text{ steps}\}$$

is primitive recursive:

$$\llbracket x \rrbracket(y) \downarrow \text{ in } \leq t \text{ steps} \iff (\text{snap}(x, y, t))_0 \geq \text{lh } x$$

Call this ternary relation T , so that

$$\begin{aligned} T(x, y, t) &\iff \llbracket x \rrbracket(y) \downarrow \text{ in } \leq t \text{ steps} \\ &\iff (\text{snap}(x, y, t))_0 \geq \text{lh } x. \end{aligned}$$

And here we can replace y by \vec{y} . That is, for each n , we define the $(n+2)$ -ary $T^{(n)}$

$$\begin{aligned} T^{(n)}(x, \vec{y}, t) &\iff \llbracket x \rrbracket^{(n)}(\vec{y}) \downarrow \text{ in } \leq t \text{ steps} \\ &\iff (\text{snap}^{(n)}(x, \vec{y}, t))_0 \geq \text{lh } x. \end{aligned}$$

and this relation is primitive recursive. (The relation is closely related to one that is often called “the Kleene T -predicate.”)

Recursively enumerable relations

The set K , while non-computable, is not all bad. Although we lack a decision procedure for it, we do have an “acceptance” procedure. That is, its semi-characteristic function

$$c_K(x) = \begin{cases} 1 & \text{if } x \in K \\ \uparrow & \text{if } x \notin K \end{cases}$$

is a computable partial function, because

$$c_K(x) = 0 \cdot \mu t T(x, x, t) + 1.$$

(Here we are exploiting the fact that the product $0 \cdot \mu t T(x, x, t)$ is undefined unless both factors are defined.) So despite the fact that K is not a decidable set, it is at least semi-decidable.

We will be interested in other such sets. The following theorem lists four ways of characterizing them.

Theorem: For an m -ary relation R on \mathbb{N} , the following conditions are equivalent:

- (a) The semi-characteristic function of R

$$c_R(\vec{x}) = \begin{cases} 1 & \text{if } \vec{x} \in R \\ \uparrow & \text{if } \vec{x} \notin R \end{cases}$$

is a computable partial function. (Informally, this condition tells us that we have an effective “acceptance procedure” for R , so that R is an effectively recognizable relation.)

- (b) R is the domain of some computable partial function.
(c) For some $(m + 1)$ -ary computable relation Q ,

$$\vec{x} \in R \iff \exists y Q(\vec{x}, y).$$

(We say that R is a Σ_1 relation if this condition holds. We can think of y as providing “evidence” that \vec{x} belongs to R .)

- (d) For some k and some $(m + k)$ -ary computable relation Q ,

$$\vec{x} \in R \iff \exists y_1 \cdots \exists y_k Q(\vec{x}, y_1, \dots, y_k).$$

Proof: To show equivalence of the conditions, it suffices to obtain four implications, forming a loop. But instead, we will obtain six.

- (a) \Rightarrow (b): Easy; $R = \text{dom } c_R$.

(b) \Rightarrow (a): $c_{\text{dom } f}(\vec{x}) = 0 \cdot f(\vec{x}) + 1$. That is, whenever f is a computable partial function, then the function mapping \vec{x} to $0 \cdot f(\vec{x}) + 1$ is a computable partial function with the same domain and with range at most $\{1\}$. (By the rules for composition of partial functions, a product such as $0 \cdot f(\vec{x})$ is defined only if both factors are defined.)

(b) \Rightarrow (c): Assume that R is the domain of the computable partial function $\llbracket e \rrbracket^{(n)}$. Apply the normal form theorem:

$$\begin{aligned} \vec{x} \in \text{dom } \llbracket e \rrbracket^{(n)} &\iff \exists t [\llbracket e \rrbracket^{(n)}(\vec{x}) \downarrow \text{ in } \leq t \text{ steps}] \\ &\iff \exists t T^{(n)}(e, \vec{x}, t) \end{aligned}$$

This shows a bit more than (c) states: It shows that in (c) we can get Q to be not only computable but even primitive recursive. And later on, we will want to make use of this extra bit of information. (Here the “evidence” that \vec{x} belongs to R is the *time* at which we discover the fact.)

(c) \Rightarrow (b): Assume that $R(\vec{x}) \Leftrightarrow \exists y Q(\vec{x}, y)$ and define

$$f(\vec{x}) = \mu y Q(\vec{x}, y).$$

Then f is a computable partial function and its domain is R .

(c) \Rightarrow (d): Obvious.

(d) \Rightarrow (c): We use the following technique to “collapse quantifiers”:

$$\exists y_1 \cdots \exists y_k Q(\vec{x}, y_1, \dots, y_k) \iff \exists y Q(\vec{x}, (y)_1, \dots, (y)_k).$$

The $(m + 1)$ -ary relation

$$\{(\vec{x}, y) \mid Q(\vec{x}, (y)_1, \dots, (y)_k)\}$$

is computable by the substitution property from page 218. \dashv

If R meets the conditions listed in this theorem, we say that R is *computably enumerable*, abbreviated c.e., or that R is *recursively enumerable*, abbreviated r.e. Both the “c.e.” and the “r.e.” terminologies are in common use. When said aloud, the phrase “r.e.” is more euphonious than the phrase “c.e.” is. Church’s thesis tells us the concept of being a recursively enumerable relation corresponds to the informal idea of being a semi-decidable relation. Whenever x belong to an r.e. set W_e , then we can effectively verify this fact by running program number e on input x until it halts, as it eventually must. (But if $x \notin W_e$, this procedure will run forever, leaving us waiting in vain for an answer, never sure whether to give up or to wait just a bit more.)

For example, any computable relation (and by now we know many of these) is also recursively enumerable. (It might be a good idea to stop and check that for a computable relation R , each of the conditions (a)–(d) of the preceding theorem holds.) Beyond that, the set K is a recursively enumerable set, and the halting relation H is a recursively enumerable binary relation. (Right?)

The enumeration theorem gives us an enumeration of the r.e. relations. That is, define

$$W_e^{(n)} = \text{dom } \llbracket e \rrbracket^{(n)}.$$

Then

$$W_0^{(n)}, \quad W_1^{(n)}, \quad W_2^{(n)}, \quad \dots$$

is a complete list (with repetitions) of all the recursively enumerable n -ary relations. As usual, when $n = 1$, we can omit the superscript. Thus

$$W_0, \quad W_1, \quad W_2, \quad \dots$$

is a complete list of all the recursively enumerable sets of natural numbers.

For a non-example, consider the set

$$\overline{K} = \{x \mid \llbracket x \rrbracket(x) \uparrow\}.$$

This is the complement of an r.e. set (such sets are sometimes called co-r.e. sets), but the set \overline{K} is *not* recursively enumerable.

Proposition: For any recursively enumerable subset W_e of \overline{K} , we have $e \in \overline{K} \setminus W_e$.

That is, whenever $W_e \subseteq \overline{K}$, then the number e itself is a witness to the fact that equality does not hold. So the proposition immediately implies that \overline{K} is not r.e.

Proof: We have $W_e \subseteq \overline{K}$. It is not possible to have $e \in K$ because that would imply that $e \in W_e \subseteq \overline{K}$. Hence we can be sure that $e \in \overline{K}$. And from that we get $e \notin W_e$. \dashv

The fact that \overline{K} is not r.e. also follows from the following result.

Kleene's theorem: A relation is computable if and only if both it and its complement are recursively enumerable.

We have encountered this theorem before; see page 109. The result appeared in a 1943 paper by Kleene; it was observed independently by Post and by Andrzej Mostowski.

Proof: In the one direction, assume that R is a computable relation. Then we know that its complement (with respect to \mathbb{N}^n) \overline{R} is also a computable relation. And because all computable relations are also r.e., it follows that both R and \overline{R} are recursively enumerable.

For the more serious direction, we assume that R is an n -ary relation such that both R and \overline{R} are recursively enumerable. Thus each is Σ_1 : For some $(n + 1)$ -ary computable relations P and Q , we have both

$$\begin{aligned} \vec{x} \in R &\iff \exists y P(\vec{x}, y) \quad \text{and} \\ \vec{x} \in \overline{R} &\iff \exists y Q(\vec{x}, y). \end{aligned}$$

Let

$$f(\vec{x}) = \mu y [\text{either } P(\vec{x}, y) \text{ or } Q(\vec{x}, y)].$$

Thus f searches for evidence, one way or the other. Then f is computable and total (because there always is such a number y). And

$$\vec{x} \in R \iff P(\vec{x}, f(\vec{x}))$$

which tells us that R is a computable relation (by using substitution). \dashv

Example: We know that the halting relation H

$$\langle x, y \rangle \in H \iff \llbracket x \rrbracket(y) \downarrow$$

is *not* computable. But H is a recursively enumerable relation, because

$$\llbracket x \rrbracket(y) \downarrow \iff \exists t T(x, y, t).$$

Applying Kleene's theorem, we can conclude that the non-halting relation \overline{H}

$$\langle x, y \rangle \in \overline{H} \iff \llbracket x \rrbracket(y) \uparrow$$

is *not* recursively enumerable.

It is not hard to see that the union of two n -ary recursively enumerable relations is again recursively enumerable. And the same is true for intersections. (See Exercise 4.) But the complement of an r.e. relation is not r.e., unless the relation is computable.

For an n -place partial function f , its *graph* is the $(n + 1)$ -ary relation

$$\{\langle \vec{x}, y \rangle \mid f(\vec{x}) = y\}.$$

In fact, a standard procedure is to define a function to be a set of ordered pairs with a certain single-valuedness property. In this approach, a function simply *is* its graph. We will ignore this point.

Theorem: A partial function is a computable partial function if and only if its graph is a recursively enumerable relation.

We observed earlier (page 218) that the graph of a *total* computable function is a computable relation. This can fail in the non-total case; for example the graph of c_K , the semi-characteristic function of K , is a non-computable binary relation, because

$$x \in K \iff \langle x, 1 \rangle \in \text{the graph of } c_K.$$

Proof: In one direction, assume that f is a partial function whose graph is the Σ_1 relation

$$\{\langle \vec{x}, y \rangle \mid \exists z R(\vec{x}, y, z)\}$$

where R is a computable relation. Then given \vec{x} , we need a “two-dimensional” search: we want to locate both the answer y and the evidence z . The μ -operator does the search; the dimensionality is easy to deal with:

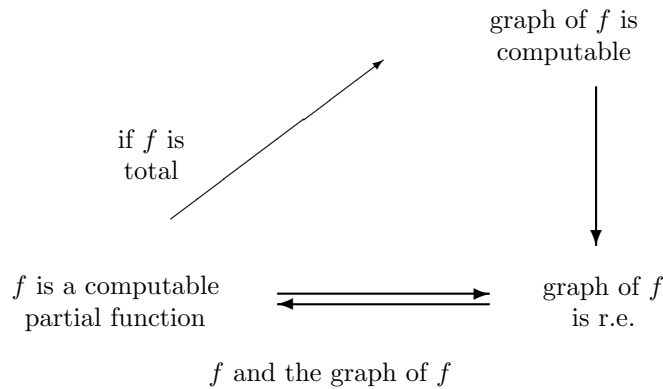
$$f(\vec{x}) = (\mu t R(\vec{x}, (t)_0, (t)_1))_0$$

That is, we search for y and z , and then we ignore z and return y . This equation shows that f is a computable partial function.

In the other direction, consider the computable partial function $\llbracket e \rrbracket^{(n)}$. We apply the normal form theorem:

$$\begin{aligned} \langle \vec{x}, y \rangle \in \text{the graph of } \llbracket e \rrbracket^{(n)} &\iff \exists t [\llbracket e \rrbracket^{(n)}(\vec{x}) = y \text{ in } \leq t \text{ steps}] \\ &\iff \exists t [T^{(n)}(e, \vec{x}, t) \ \& \ ((\text{snap}(e, \vec{x}, t)_1)^* = y)] \end{aligned}$$

Observe that this relation is Σ_1 . \dashv



In the foregoing proof, we have made use of the fact that

$$T^{(n)}(e, \vec{x}, t) \quad \& \quad ((\text{snap}(e, \vec{x}, t)_1))^*_0 = y$$

is a primitive recursive condition on e, \vec{x}, t, y that says “ $\llbracket e \rrbracket^{(n)}(\vec{x}) = y$ in $\leq t$ steps.” The fact that this condition is primitive recursive will be useful later, as well.

We know that the domain of a computable partial function is r.e. The same is true of the range:

Theorem: The range of any computable partial function is a recursively enumerable subset of \mathbb{N} .

Proof 1: The range of a function f is the set

$$\{y \mid \exists \vec{x} \langle \vec{x}, y \rangle \in \text{the graph of } f\}.$$

By the preceding theorem, we can express the graph of f as a Σ_1 relation

$$\{\langle \vec{x}, y \rangle \mid \exists t Q(\vec{x}, y, t)\}$$

where Q is a computable relation. Then

$$y \in \text{ran } f \iff \exists \vec{x} \exists t Q(\vec{x}, y, t)$$

which shows that the range is r.e. \dashv

Proof 2: The same argument can be condensed into one line:

$$y \in \text{ran } f \iff \exists \vec{x} \exists t [f(\vec{x}) = y \text{ in } \leq t \text{ steps}]$$

\dashv

What is especially “enumerable” about recursively enumerable sets? The following theorem (which also provides a converse to the preceding theorem) gives an answer.

Theorem: A subset of \mathbb{N} is recursively enumerable if and only if it either is empty or is the range of a total computable 1-place function.

Proof: In one direction, the preceding theorem applies: The range of a total computable 1-place function is r.e., as is the range of *any* computable partial function. And the empty set is both computable and recursively enumerable.

It is the other direction that requires proof. Assume that A is a non-empty computable enumerable subset of \mathbb{N} ; say c is some particular member of A . Then A is Σ_1 ; say

$$x \in A \iff \exists y Q(x, y)$$

for computable Q . Then a *two*-place function with range equal to A is the function f_2 :

$$f_2(x, y) = \begin{cases} x & \text{if } Q(x, y) \\ c & \text{otherwise} \end{cases}$$

But we want a one-place function. Define

$$f(t) = \begin{cases} (t)_0 & \text{if } Q((t)_0, (t)_1) \\ c & \text{otherwise.} \end{cases}$$

(Informally, if t says to us, “I have found a member of A and here it is and here is the evidence,” then we act accordingly.) Then f is a total computable function (by definition-by-cases) and its range is A . In fact, because we can get Q to be primitive recursive, we can get f to be a primitive recursive function. (In particular, this shows that the range of a primitive recursive function need *not* be a primitive recursive set, or even a computable set.) \dashv

In this theorem (in the more interesting direction), we have

$$A = \{f(0), f(1), f(2), \dots\}.$$

That is, the function f gives us an effective *enumeration* of the set A . (Julia Robinson suggested using the word “listable” instead of “enumerable.”) It is not, in general, possible to get f to enumerate the members of A in increasing order, unless A is computable. (See the exercises.)

The preceding theorems show that

$$\text{ran } [0], \quad \text{ran } [1], \quad \text{ran } [2], \quad \dots$$

give a complete list (with repetitions) of exactly the recursively enumerable subsets of \mathbb{N} . The advantage of using the domain instead of the range (as we did in defining W_e) is that it extends to recursively enumerable relations that are subsets of \mathbb{N}^k for larger k .

Among the computable partial functions, the ones that are *total* have a particular “user-friendly” status: When you give an input to a total function, you get an output back. Define the set

$$\text{Tot} = \{e \mid [e] \text{ is total}\}$$

of indices of total 1-place computable functions. This is not a computable set, as shown in the exercises. And moreover, it is not even a recursively enumerable set. The following theorem proves an even stronger fact.

Theorem: Assume that A is a recursively enumerable subset of Tot. Then there is some total 1-place computable function that does not equal $\llbracket a \rrbracket$ for any number a belonging to A .

Proof: We can obviously assume that A is nonempty. Hence by the preceding theorem A is the range of some total computable 1-place function g . Define the following function:

$$f(x) = \llbracket g(x) \rrbracket(x) + 1$$

(Alternatively, we can just as well use $f(x) = 1 \div \llbracket g(x) \rrbracket(x)$.) This is a computable partial function. Moreover, it is total, because $g(x) \in \text{Tot}$ for all x . Could f equal $\llbracket a \rrbracket$ for some a in A ? We know that $a = g(t)$ for some t . But then

$$\llbracket a \rrbracket(t) = \llbracket g(t) \rrbracket(t) \text{ and } f(t) = \llbracket g(t) \rrbracket(t) + 1$$

and these are defined. So f cannot be the same as $\llbracket a \rrbracket$; the two functions differ at t . \dashv

To illustrate this theorem, suppose that you are teaching a beginning programming course. Your students keep turning in programs for non-total functions, which you find annoying. So you give them a set of rules such that the rule-following programs always compute total functions. Assume that the set of rule-following programs is computable, or at least recursively enumerable (so we can recognize a rule-following program when we see one). Then it follows from the above theorem that some total function will have *no* rule-following program. That is, imposing the rules has had the unintended side effect of limiting the class of total programmable functions. (Programming rules may, indeed, be a good thing. But one should be aware that there is a cost—there may be some total functions that can be programmed *only* by breaking the rules.)

For a second illustration, suppose we have a program that we are pretty sure computes a total function. But to be certain, we want a *correctness proof*. Will there necessarily be a proof that the program halts on all possible inputs?

This question leads us to ask: What is a proof? Since a proof should be a finite string of symbols, we can code a proof by a number, in much the same way as we coded programs by numbers. Suppose that we have formalized a notion of “proof of totality” in such a way that the following two assumptions are met.

1. Proofs don’t lie. That is, whenever there exists a proof that program number y is total, then $\llbracket y \rrbracket$ is really total.
2. We can effectively recognize proofs. That is, the relation

$$\{(y, z) \mid z \text{ codes a proof that program } y \text{ is total}\}$$

is recursively enumerable. (The idea behind this assumption is that if a proof is to *convince* someone, then that person must at the very least be able to *verify* the proof in an effective way.)

Then apply the theorem to the set

$$B = \{y \mid \exists z(z \text{ codes a proof that program } y \text{ is total})\}$$

of provably total programs (in this proof system). By the first assumption, $B \subseteq \text{Tot}$. By the second assumption, B is recursively enumerable. We conclude that there exist total computable functions such that *no* programs for those functions are provably total in this system!

(The fact that Tot is not r.e. implies that some total programs are not provably total in the system. The result here is stronger. It states that there are computable total functions *all* of whose programs fail to be provably total in the system.)

There is a connection here to Gödel's incompleteness theorem, which we will encounter in Chapter 5.

Next we want to show that the class of recursively enumerable relations is closed under substitution of computable functions.

Proposition: (a) If A is a recursively enumerable subset of \mathbb{N} and f is a total computable function, then $\{x \mid f(x) \in A\}$ is also recursively enumerable.

(b) If R is a recursively enumerable n -ary relation and f_1, \dots, f_n are total computable m -place functions, then the m -ary relation

$$\{\vec{x} \mid R(f_1(\vec{x}), \dots, f_n(\vec{x}))\}$$

is also recursively enumerable.

In fact, the functions here do not even need to be total, if, for example, $\{x \mid f(x) \in A\}$ is understood as the set of x 's for which $f(x)$ is defined and belongs to A . So let's restate the proposition of greater generality. We write " $f(x) \downarrow$ " to indicate that $f(x)$ is defined (i.e., $x \in \text{dom } f$).

Proposition: (a) If A is a recursively enumerable subset of \mathbb{N} and f is a computable partial function, then $\{x \mid f(x) \downarrow \text{ and } f(x) \in A\}$ is also recursively enumerable.

(b) If R is a recursively enumerable n -ary relation and f_1, \dots, f_n are computable partial m -place functions, then the m -ary relation

$$\{\vec{x} \mid \langle f_1(\vec{x}), \dots, f_n(\vec{x}) \rangle \downarrow \text{ and belongs to } R\}$$

is also recursively enumerable.

Proof: (b) We know that the semi-characteristic function c_R of R is a computable partial function. The semi-characteristic function of the new relation is

$$\vec{x} \mapsto c_R(f_1(\vec{x}), \dots, f_n(\vec{x}))$$

which is a composition of computable partial functions. \dashv

But note that for a *computable* set A and a computable partial function f , the set $\{x \mid f(x) \downarrow \text{ and } f(x) \in A\}$ is not computable in general. For example, if $A = \mathbb{N}$, then this set is simply the domain of f , which might not be computable.

For a computable relation Q and a computable partial function g , we have seen that the partial function g^Q defined by cases

$$g^Q(\vec{x}) = \begin{cases} g(\vec{x}) & \text{if } \vec{x} \in Q \\ 0 & \text{if } \vec{x} \notin Q \end{cases}$$

is again a computable partial function. If we weaken the assumption on Q to assume only that Q is recursively enumerable, then g^Q might fail to be a computable partial function (for example, if $Q = K$ and $g(x) = 1$). But we do have the following result.

Proposition (Definition by semi-cases): Assume that g is a computable partial n -place function and Q is a recursively enumerable n -ary relation. Then the function

$$(g \upharpoonright Q)(\vec{x}) = \begin{cases} g(\vec{x}) & \text{if } \vec{x} \in Q \\ \uparrow & \text{if } \vec{x} \notin Q \end{cases}$$

is a computable partial function.

Note that $(g \upharpoonright Q)(\vec{x})$ is undefined unless both $\vec{x} \in Q$ and $g(\vec{x}) \downarrow$. Informally, the procedure for computing $(g \upharpoonright Q)(\vec{x})$ involves first trying to verify that $\vec{x} \in Q$, and then computing $g(\vec{x})$.

Proof 1: $(g \upharpoonright Q)(\vec{x}) = g(\vec{x}) + 0 \cdot c_Q(\vec{x})$. \dashv

Proof 2: $(g \upharpoonright Q)(\vec{x}) = y \Leftrightarrow \vec{x} \in Q \ \& \ g(\vec{x}) = y$, so the graph of $g \upharpoonright Q$ is the intersection of two $(n + 1)$ -ary r.e. relations \dashv

All of the non-computable sets are non-computable, but some are more non-computable than others. One way to make sense out of that statement is to look at ways in which membership questions about one set might be “reduced” to membership questions about another.

More specifically, for sets A and B of natural numbers, we say that A is *many-one reducible* to B (in symbols, $A \leq_m B$) if there exists some total computable function such that

$$x \in A \iff f(x) \in B$$

for all x . That is, the function f is, in a sense, effectively reducing the question whether $x \in A$ to a question about B .

Note that if $A \leq_m B$, then it is automatically true that $\overline{A} \leq_m \overline{B}$, by using the same function.

Digression: The name “many-one reducible” is not particularly informative. The name “mapping reducible” has been suggested as an alternative. And it retains the feature of starting the letter ‘m’ so that the symbol \leq_m does not need to be altered.

Proposition: Assume that A and B are sets of natural numbers with $A \leq_m B$.

- (a) If B is a computable set, then A is also computable.
- (b) If B is recursively enumerable, then A is also recursively enumerable.

Proof: Part (a) is a substitution rule, which we already have seen back on page 218: $A = \{x \mid f(x) \in B\}$.

Part (b) follows from an earlier proposition. If f is a computable function that many-one reduces A to B , then

$$A = \{x \mid f(x) \in B\}$$

and so if B is recursively enumerable, then so is A . \dashv

For example, in order to show that some set is not recursively enumerable, it suffices to show that \overline{K} is many-one reducible to it.

Exercises

1. Obviously $\text{Tot} \subseteq K$. Show that there is no computable set A with $\text{Tot} \subseteq A \subseteq K$. Suggestion: Consider the function defined by the equation:

$$f(x) = \begin{cases} \llbracket x \rrbracket(x) + 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

2. (a) Assume that f is a total computable 1-place non-decreasing function (that is, $f(x) \leq f(x+1)$ for all x). Further assume that the range of f is infinite. Show that the range of f is a computable set.

(b) Suppose that in part (a), we drop the assumption that the range is infinite. Show that the same conclusion still holds.

3. Assume that A is an infinite recursively enumerable set of natural numbers. Show that there is a total computable strictly increasing function g (that is, $g(x) < g(x+1)$ for all x) whose range is included in A . (It follows from this exercise that every infinite recursively enumerable set has an infinite computable subset.)

4. (a) Show that the intersection of two n -ary recursively enumerable relations is again recursively enumerable.

(b) Show that the union of two n -ary recursively enumerable relations is again recursively enumerable.

5. Assume that f is a total computable function. Show that

$$\bigcup_{n \in \mathbb{N}} W_{f(n)}$$

is recursively enumerable. (That is, a computably indexed union of r.e. sets is r.e.)

6. (a) Show that there is a computable partial function f such that whenever $W_x \neq \emptyset$, then $f(x) \downarrow$ and $f(x) \in W_x$. (That is, the function f finds *some* member of W_x , provided there is a member.)

(b) Assume that R is a recursively enumerable binary relation. Construct a computable partial function f such that whenever $\exists y R(x, y)$, then $f(x) \downarrow$ and $R(x, f(x))$. (That is, the function f finds *some* y such that $R(x, y)$, provided there is one.)

7. Show that the set $\{x \mid \llbracket x \rrbracket(x) = 0\}$ (i.e., the set of x 's for which $\llbracket x \rrbracket(x)$ is defined and equals 0) is recursively enumerable but not computable.

8. Let

$$A = \{x \mid \llbracket x \rrbracket(x) = 0\} \quad \text{and} \quad B = \{x \mid \llbracket x \rrbracket(x) = 1\}.$$

(a) Show that A and B are disjoint r.e. sets.

(b) Show that A and B are *computably inseparable*, that is, there is no computable set D with $A \subseteq D$ and $B \subseteq \bar{D}$. Suggestion: Suppose we had such a set D ; let d be an index of its characteristic function. What can you say about $\llbracket d \rrbracket(d)$?

9. Give an alternative proof of Kleene's theorem as follows. Assume that R is a relation for which both R and \bar{R} are recursively enumerable. Show that C_R , the characteristic function of R , has a recursively enumerable graph.

Parameters

Next we want to turn our attention to the subject of *calculating programs*. That is, suppose we want a program that will meet some particular need. Possibly we have good reason to know that a program *exists* that meets the need. But we might want more than that; we might want actually to *find* such a program.

For example, we know that every total constant function is computable. (As noted in item **2** in Chapter 2, every such function is primitive recursive.) But even more is true. Given a constant k , we can actually compute an index of the one-place function that is constantly equal to k . See Exercise 4 in Chapter 3.

For another example, we have seen recently that the range of any computable partial 1-place function $\llbracket e \rrbracket$ is an r.e. set, and therefore is W_y for some y . That is, there exists some index y of a function whose domain is the set we want. But a stronger fact is true: Given e , we can actually compute such a number y . (We will see a proof of this later.) That is, there is a computable function f such that

$$\text{ran } \llbracket e \rrbracket = W_{f(e)}$$

for every e .

For a third example, suppose we have a two-place computable partial function f . Then clearly the one-place function g obtained by holding the second variable fixed as a parameter, say

$$g(x) = f(x, 3),$$

is a computable partial function; we have merely applied composition with a computable (and constant) function. Or in terms of register machines, we can

make a program for g that increments register 2 three times, and then follows the program for f .

But by standing back and looking at the previous paragraph, we perceive a more subtle fact. We were able to find explicitly a program for g , given the parameter 3 and a program for f . So there should be a computable function ρ that, given an index e for f and given the parameter 3, will compute an index of the function g :

$$f(x, 3) = \llbracket \rho(e, 3) \rrbracket(x)$$

Not only does there *exist* a program to compute g , but, given the parameter 3, we can actually lay our hands on such a program.

Parameter theorem: There is a primitive recursive function ρ such that the equation

$$\llbracket e \rrbracket^{(2)}(x, y) = \llbracket \rho(e, y) \rrbracket(x)$$

holds for all e , x , and y . (Here equality has the usual meaning: either both sides are undefined, or both sides are defined and are the same.) Moreover, ρ is one-to-one.

The idea is that if we have a computable partial two-place function $\llbracket e \rrbracket^{(2)}$ and we want to hold the last variable fixed as a parameter, then ρ will actually calculate an index for the resulting one-place function.

Proof, first try: The program that increments register 2 exactly y times has Gödel number

$$[\#I2, \#I2, \dots, \#I2].$$

The Gödel number of the instruction I2 is $[0, 2] = 2^1 \cdot 3^3 = 54$. So the Gödel number of the program incrementing register 2 exactly y times is

$$[54, 54, \dots, 54] = \prod_{t < y} p_t^{55}$$

which is a primitive recursive function of y ; call it $k_2(y)$. This suggests that we might be able to take

$$\rho(e, y) = k_2(y) * e$$

using the concatenation function $*$ from item **21** in Chapter 2. This will indeed work for “nice” values of e . But difficulties can arise if e is the Gödel number of a program that makes bad negative jumps (i.e., jumps to a point before the start of the program). Also, when e is not the Gödel number of a program at all, then we are less certain that this equation for ρ will give us the theorem we are after.

Proof, second try: There is a way to circumvent the difficulties that arise in the first try. We use a universal function instead. The universal partial function

$$\Phi^{(2)}(e, x, y) = \llbracket e \rrbracket^{(2)}(x, y)$$

is a computable partial function; fix some register-machine program \mathcal{Q} that computes it. (By the way we defined the verb “computes,” the program \mathcal{Q} always either runs forever or comes to a good halt. Thus we can avoid the difficulties mentioned earlier.) Define

$$\rho(e, y) = k_2(y) * k_3(e) * q$$

where q is the Gödel number of the program \mathcal{Q} . Here k_3 is the function like k_2 , but it uses register 3. Clearly ρ is primitive recursive. (Here q is a fixed constant.)

To check that ρ works, let’s calculate $\llbracket \rho(e, y) \rrbracket(x)$. Here $\rho(e, y)$ is the Gödel number of a program, and we know what that program does, given the input x :

- First it inserts y into register 2.
- Secondly, it inserts e into register 3.
- Lastly, it runs \mathcal{Q} to try to find $\Phi^{(2)}(e, x, y) = \llbracket e \rrbracket^{(2)}(x, y)$, if this is defined.

That is, we get exactly $\llbracket e \rrbracket^{(2)}(x, y)$ if this is defined, and we get nothing if this is undefined. So the equation

$$\llbracket e \rrbracket^{(2)}(x, y) = \llbracket \rho(e, y) \rrbracket(x)$$

holds.

The function ρ is one-to-one, because different values of e and y will result in different programs, and hence different Gödel numbers. \dashv

As an example of an application of the parameter theorem, we can “uniformize” an earlier result. We have seen that the range of any computable partial function $\llbracket r \rrbracket$ is an r.e. set. The “uniformized” version of this statement is that there is some total computable (in fact primitive recursive) function f such that

$$\text{ran } \llbracket r \rrbracket = W_{f(r)}$$

for every r . That is, the function f goes out and finds an “r.e. index” for the range of $\llbracket r \rrbracket$.

We know that

$$\begin{aligned} y \in \text{ran } \llbracket r \rrbracket &\iff \exists x \llbracket r \rrbracket(x) = y \\ &\iff \exists x \exists t \llbracket r \rrbracket(x) = y \text{ in } t \text{ steps.} \end{aligned}$$

Look at the function that searches for x and t here:

$$g(y, r) = \mu s [\llbracket r \rrbracket((s)_0) = y \text{ in } (s)_1 \text{ steps}].$$

Then g is a computable partial function, so it is $\llbracket e \rrbracket^{(2)}$ for some e . Parameterize out r :

$$\llbracket \rho(e, r) \rrbracket(y) = \llbracket e \rrbracket^{(2)}(y, r) = g(y, r) = \mu s [\llbracket r \rrbracket((s)_0) = y \text{ in } (s)_1 \text{ steps}].$$

This quantity is defined if and only if there is some s to be found, which happens if and only if $y \in \text{ran } \llbracket r \rrbracket$. That is,

$$\text{ran } \llbracket r \rrbracket = W_{\rho(e,r)}$$

which is what we wanted.

Here is another application of the parameter theorem:

Lemma: Assume that S is an r.e. set of natural numbers, and that f is a computable partial one-place function. Then there is a one-to-one primitive recursive function g such that for any y :

(i) If $y \in S$, then $\llbracket g(y) \rrbracket$ is the partial function f .

(ii) If $y \notin S$, then $\llbracket g(y) \rrbracket$ is the empty function, that is, the function that is undefined for all inputs.

So the function g produces indices of functions. In fact, $g(y)$ is an index either for f or for the empty function. And which is these two alternatives occurs is determined by whether or not $y \in S$.

The proof involves looking at the two-place partial function

$$h(x, y) = \begin{cases} f(x) & \text{if } y \in S \\ \uparrow & \text{if } y \notin S \end{cases}$$

which can be computed by the one-line instruction,

“First verify that $y \in S$ and secondly find $f(x)$.”

The point is that given y , we can effectively find the above one-line instruction. There is a primitive recursive function that, given y , produces the above line with the value y filled in. The actual proof cleans this argument up.

Proof: Applying definition by semi-cases, we obtain a two-place computable partial function h :

$$h(x, y) = \begin{cases} f(x) & \text{if } y \in S \\ \uparrow & \text{if } y \notin S \end{cases}$$

So h is $\llbracket e \rrbracket^{(2)}$ for some number e . Now parameterize out the y . We get

$$\llbracket \rho(e, y) \rrbracket(x) = \llbracket e \rrbracket^{(2)}(x, y) = \begin{cases} f(x) & \text{if } y \in S \\ \uparrow & \text{if } y \notin S. \end{cases}$$

Now let $g(y) = \rho(e, y)$ for this number e . Then g is primitive recursive, one-to-one, and

$$\llbracket g(y) \rrbracket(x) = \llbracket \rho(e, y) \rrbracket(x) = \begin{cases} f(x) & \text{if } y \in S \\ \uparrow & \text{if } y \notin S. \end{cases}$$

Thus the partial function $\llbracket g(y) \rrbracket$ either is f (if $y \in S$) or else is the empty function (if $y \notin S$). \dashv

Application: We can show that $K \leq_m \text{Tot}$. In the lemma, take S to K and take f to the identity function I_1^1 (or any total computable function). We obtain a primitive recursive function g such that whenever $y \in K$ then $g(y) \in \text{Tot}$, because $g(y)$ is an index of the total function I_1^1 . And whenever $y \notin K$ then $g(y) \notin \text{Tot}$, because $g(y)$ is an index of the empty function, which is certainly not total.

It follows that $K \leq_m \text{Tot}$ under this function g . And so automatically $\overline{K} \leq_m \overline{\text{Tot}}$ under the same function. Consequently, $\overline{\text{Tot}}$ is not recursively enumerable. (We saw earlier that Tot itself is not recursively enumerable.)

Similarly, the set of indices of the empty function

$$\{e \mid \llbracket e \rrbracket \text{ is empty}\} = \{e \mid W_e = \emptyset\}$$

is not recursively enumerable, because \overline{K} is many-one reducible to it by the same function g .

Application: We can show that $S \leq_m K$ for any r.e. set S . Apply the lemma where f again is I_1^1 or some other total computable function. We obtain a one-to-one primitive recursive function g such that for any y ,

$$\begin{aligned} y \in S &\Rightarrow \llbracket g(y) \rrbracket \text{ total} \Rightarrow g(y) \in K \\ y \notin S &\Rightarrow \llbracket g(y) \rrbracket \text{ empty} \Rightarrow g(y) \notin K. \end{aligned}$$

Thus $S \leq_m K$ under g .

Definition: A *complete* recursively enumerable set is a recursively enumerable subset C of \mathbb{N} with the property that

$$A \leq_m C$$

for every recursively enumerable subset A of \mathbb{N} .

The preceding application proves the following result.

Proposition: The set K is a complete recursively enumerable set.

Here is another complete r.e. set obtained in a more direct way:

$$C = \{x \mid (x)_0 \in W_{(x)_1}\}$$

Then C is recursively enumerable because

$$x \in C \iff \exists t T((x)_1, (x)_0, t).$$

And C is complete because

$$x \in W_a \iff [x, a] \in C.$$

We can “vectorize” the parameter theorem as follows. (In this form, the theorem is commonly called the “ S - m - n theorem,” for no very good reason.)

Parameter theorem: For each m and n , there is a $(n+1)$ -place primitive recursive function ρ_{mn} such that the equation

$$\llbracket e \rrbracket^{(m+n)}(\vec{x}, \vec{y}) = \llbracket \rho_{mn}(e, \vec{y}) \rrbracket^{(m)}(\vec{x})$$

for all e , all m -tuples \vec{x} , and all n -tuples \vec{y} . (Here equality has the usual meaning: either both sides are undefined, or both sides are defined and are the same.) Moreover, ρ_{mn} is one-to-one.

Proof: We proceed as before. Here \vec{x} is $\langle x_1, \dots, x_m \rangle$ and \vec{y} is $\langle y_1, \dots, y_n \rangle$.

$$\rho_{mn}(e, y_1, \dots, y_n) = k_{m+1}(y_1) * \dots * k_{m+n}(y_n) * k_{m+n+1}(e) * q$$

where q is the Gödel number of our program that computes $\Phi^{(m+n)}$. (Here m , n , and q are fixed; e , \vec{x} , and \vec{y} are the variables.) \dashv

For example, we have long known that the composition $f \circ g$ of two computable partial functions is a computable partial function. But now we can obtain a “uniform” version of that statement: There is a total computable function h such that

$$\llbracket h(x, y) \rrbracket = \llbracket x \rrbracket \circ \llbracket y \rrbracket$$

for all x and y . That is, not only does a program for the composition exist, but we can effectively find it.

The property we need h to satisfy is

$$\llbracket h(x, y) \rrbracket(t) = \llbracket x \rrbracket(\llbracket y \rrbracket(t))$$

for all t . So look at the right-hand side as a function all the variables: it is a computable partial function of t , x , and y . So there is some e for which

$$\llbracket e \rrbracket^{(3)}(t, x, y) = \llbracket x \rrbracket(\llbracket y \rrbracket(t))$$

for all t , x , and y . We proceed to parameterize out the x and y :

$$\llbracket e \rrbracket^{(3)}(t, x, y) = \llbracket \rho_{12}(e, x, y) \rrbracket(t).$$

So we can take $h(x, y) = \rho_{12}(e, x, y)$ for this e . Then h is a primitive recursive function.

The following result was published by H. Gordon Rice in 1953.

Rice’s theorem: Let \mathcal{C} be a set of one-place computable partial functions and let $I_{\mathcal{C}} = \{e \mid \llbracket e \rrbracket \in \mathcal{C}\}$ be the set of all indices of members of \mathcal{C} . Then $I_{\mathcal{C}}$ is non-computable except in the two trivial cases: $\mathcal{C} = \emptyset$ (in which case $I_{\mathcal{C}} = \emptyset$) or \mathcal{C} is the set of *all* one-place computable partial functions (in which case $I_{\mathcal{C}} = \mathbb{N}$).

Proof: First we look to see where the empty function (the function that is undefined everywhere), call it \emptyset , is.

Case I: The empty function \emptyset is *not* in \mathcal{C} . We are given that \mathcal{C} has some function in it; say $\psi \in \mathcal{C}$. We can apply a recent lemma to obtain a computable total function g with the following properties:

- (i) Whenever $y \in K$, then $\llbracket g(y) \rrbracket$ is the function ψ .
- (ii) Whenever $y \notin K$, then $\llbracket g(y) \rrbracket$ is the empty function \emptyset .

Then $K \leq_m I_{\mathcal{C}}$ under the function g :

$$y \in K \iff g(y) \in I_{\mathcal{C}}$$

Hence $I_{\mathcal{C}}$ is not computable.

Case II: The empty function \emptyset *is* in \mathcal{C} . We proceed much as before. We are given that not everything is in \mathcal{C} ; say ψ is a computable partial function not in \mathcal{C} . As before, we can get a computable total function g with the following properties:

- (i) Whenever $y \in K$, then $\llbracket g(y) \rrbracket$ is the function ψ .
- (ii) Whenever $y \in \overline{K}$, then $\llbracket g(y) \rrbracket$ is the empty function \emptyset .

Then $\overline{K} \leq_m I_{\mathcal{C}}$ under the function g :

$$y \in \overline{K} \iff g(y) \in I_{\mathcal{C}}$$

Hence $I_{\mathcal{C}}$ is not recursively enumerable. \dashv

In more informal terms, we can restate Rice's theorem as follows: Assume we have in mind some *property* of one-place computable partial functions. Further assume that this property is non-trivial, in the sense that at least one computable partial function has the property, but not all do. Then we can conclude that the problem of determining whether or not a given number is the index of a partial function with the property is undecidable.

For example, suppose we focus attention on a particular computable function, such as the doubling function $x \mapsto 2x$. The doubling function can be computed by a very simple program, but it is also computed by some convoluted programs. Rice's theorem tells us that we cannot always decide of a given number whether it is an index of the doubling function or not. (And in particular, this shows that the doubling function has infinitely many indices, which is not surprising.)

For another example, Rice's theorem shows that

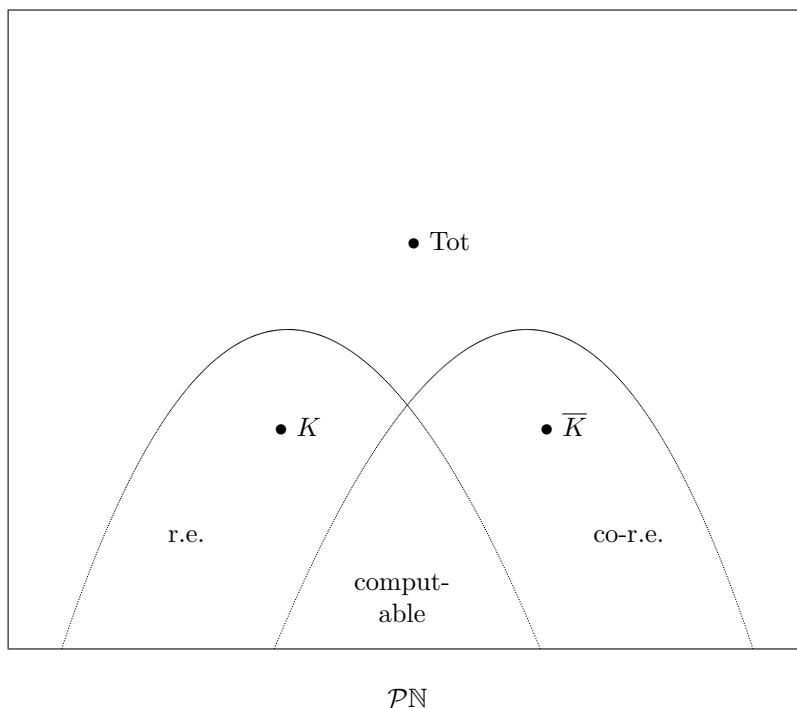
$$\{x \mid W_x \text{ is the set of primes}\}$$

is not computable. We take \mathcal{C} to be the set of computable partial functions f for which $\text{dom } f$ is the set of primes.

Our proof of Rice's theorem actually shows a bit more than the theorem states. On the one hand, it shows that when the empty function is in \mathcal{C} , then $I_{\mathcal{C}}$ is not r.e. And on the other hand, it shows that if the empty function is not in \mathcal{C} , then $\overline{K} \leq_m \overline{I_{\mathcal{C}}}$ and hence $\overline{I_{\mathcal{C}}}$ is not r.e. (that is, $I_{\mathcal{C}}$ is not "co-r.e."; it is not the complement of an r.e. set).

For example, consider the set Tot of indices of total functions. We previously saw that Tot is not r.e. We now can see that it is not co-r.e., either. Taking \mathcal{C}

to be the collection of total computable functions, we have $I_C = \text{Tot}$. Since the empty function is not total, we conclude that Tot is not co-r.e.



Another consequence of the parameter theorem is the following result, which is due to Kleene.

Recursion theorem: For any computable partial function g , we can find an e such that

$$\llbracket e \rrbracket(x) = g(e, x)$$

for all x .

Again, x can be replaced by an n -tuple \vec{x} . The proof of the recursion theorem is very like the argument generally used in logic to prove Gödel's incompleteness theorem; see Exercise 20. We will not pursue any of these topics right now.

Exercises

10. We know that the product $f \cdot g$ of computable partial functions is a computable partial function. Show that there is a total computable function h such that the equation

$$\llbracket h(x, y) \rrbracket(t) = \llbracket x \rrbracket(t) \cdot \llbracket y \rrbracket(t)$$

holds for all t, x and y .

11. We know that the union of two recursively enumerable sets is recursively enumerable. Show that there is a total computable function g such that

$$W_{g(x,y)} = W_x \cup W_y$$

for all x and y .

12. Show that

$$\{t \mid \llbracket t \rrbracket(0) = 0\} \leq_m \{y \mid \llbracket y \rrbracket(0) = 7\}.$$

13. (a) Show that $K \leq_m \{x \mid W_x \text{ is infinite}\}$.

(b) Show that $K \leq_m \{x \mid W_x \text{ is finite}\}$.

14. Show that $\text{Tot} \leq_m \{y \mid W_y \text{ is infinite}\}$.

15. Show that the binary relation Q defined by the condition

$$\langle x, y \rangle \in Q \iff \llbracket x \rrbracket \text{ and } \llbracket y \rrbracket \text{ are the same function}$$

is not computable. (This says, roughly, that the problem of determining, given two programs, if they compute the same function, is undecidable.)

16. Let f be some fixed computable partial function. Let I_f be its set of indices:

$$I_f = \{x \mid \llbracket x \rrbracket = f\}$$

(a) Show that I_f is never a computable set. Remark: Suppose you are teaching a programming class, and you assign to your students the problem of writing a program for f . Then the set of correct answers to this problem is an undecidable set!

(b) Further assume that f is total. Show that $\text{Tot} \leq I_f$.

17. Show that the binary relation R defined by the condition

$$\langle x, y \rangle \in R \iff W_x = W_y$$

is not computable.

18. (a) Prove a uniformized version of Exercise 2(a). That is, show that there is a primitive recursive function g such that whenever $\llbracket e \rrbracket$ is non-decreasing with infinite range, then $g(e)$ is an index for the characteristic function of $\text{ran } \llbracket e \rrbracket$.

(b) Show that Exercise 2(b) does not uniformize, even if we add the assumption that the range is finite. That is, show that there cannot be a computable partial function g such that whenever $\llbracket e \rrbracket$ is non-decreasing with finite range, then $g(e) \downarrow$ and $g(e)$ is an index for the characteristic function of $\text{ran } \llbracket e \rrbracket$. Suggestion: Look at the characteristic function of $\{\langle t, x \rangle \mid T(x, x, t)\}$ as a function of t .

19. Show that there is no computable partial function h such that whenever the set W_y is computable, then $h(y) \downarrow$ and $h(y)$ is an index for the characteristic function of W_y . This shows that knowing an acceptance procedure for a set,

plus knowing that the set is actually decidable, does not in general lead us to a decision procedure for the set. Suggestion: Look at the function

$$f(u, x) = \begin{cases} \mu t T(x, x, t) & \text{if } u = 0 \\ \uparrow & \text{if } u > 0. \end{cases}$$

20. (a) Using the parameter theorem, show that there is a primitive recursive function δ such that the equation

$$\llbracket \delta(y) \rrbracket(x) = \llbracket y \rrbracket^2(x, y)$$

for all x and y . (Here equality means that either both sides are undefined, or both are defined and are the same.)

(b) Now assume that g is a 2-place partial computable function. Then the function

$$\langle x, y \rangle \mapsto g(\delta(y), x)$$

is a computable partial function; let q be an index of it, and let $e = \delta(q)$. Show that for this e ,

$$\llbracket e \rrbracket(x) = g(e, x)$$

for all x , thereby proving the recursion theorem.

21. Show that there is a program so narcissistic that it only outputs its own Gödel number. That is, show that for some number e , the equation

$$\llbracket e \rrbracket(x) = e$$

holds for all x .

22. Let $S = \{x \mid \llbracket x \rrbracket(3) = 24\}$. Is S a computable set? Is S r.e.? Is \bar{S} r.e.?

23. Show that the set

$$\{x \mid \llbracket x \rrbracket(t) \uparrow \text{ for all } t \leq 900\}$$

is not r.e.

24. Show that there is no computable partial function f such that whenever W_x is non-empty, then $f(x)$ is defined and is the *least* member of W_x .