Computability and Logic

Fifth Edition

GEORGE S. BOOLOS

JOHN P. BURGESS

Princeton University

RICHARD C. JEFFREY

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK
Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521877527

© George S. Boolos, John P. Burgess, Richard C. Jeffrey 1974, 1980, 1990, 2002, 2007

This publication is in copyright. Subject to statutory exception and to the provision of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published in print format 2007

ISBN-13 978-0-511-36668-0 eBook (EBL)
ISBN-10 0-511-36668-X eBook (EBL)

ISBN-13 978-0-521-87752-7 hardback
ISBN-10 0-521-87752-0 hardback

ISBN-13 978-0-521-70146-4 paperback
ISBN-10 0-521-70146-5 paperback

Cambridge University Press has no responsibility for the persistence or accuracy of urls for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

8

Equivalent Definitions of Computability

In the preceding several chapters we have introduced the intuitive notion of effective computability, and studied three rigorously defined technical notions of computability: Turing computability, abacus computability, and recursive computability, noting along the way that any function that is computable in any of these technical senses is computable in the intuitive sense. We have also proved that all recursive functions are abacus computable and that all abacus-computable functions are Turing computable. In this chapter we close the circle by showing that all Turing-computable functions are recursive, so that all three notions of computability are equivalent. It immediately follows that Turing's thesis, claiming that all effectively computable functions are Turing computable, is equivalent to Church's thesis, claiming that all effectively computable functions are recursive. The equivalence of these two theses, originally advanced independently of each other, does not amount to a rigorous proof of either, but is surely important evidence in favor of both. The proof of the recursiveness of Turing-computable functions occupies section 8.1. Some consequences of the proof of equivalence of the three notions of computability are pointed out in section 8.2, the most important being the existence of a universal Turing machine, *a Turing machine capable of simulating the behavior of any other Turing machine desired. The optional section 8.3 rounds out the theory of computability by collecting basic facts about* recursively enumerable *sets, sets of natural numbers that can be enumerated by a recursive function. Perhaps the most basic fact about them is that they coincide with the semirecursive sets introduced in the preceding chapter, and hence, if Church's (or equivalently, Turing's) thesis is correct, coincide with the (positively) effectively semidecidable sets.*

8.1 Coding Turing Computations

At the end of Chapter 5 we proved that all abacus-computable functions are Turing computable, and that all recursive functions are abacus computable. (To be quite accurate, the proofs given for Theorem 5.8 did not consider the three processes in their most general form. For instance, we considered only the composition of a two-place function f with two three-place functions g_1 and g_2. But the methods of proof used were perfectly general, and do suffice to show that any recursive function can be computed by some Turing machine.) Now we wish to close the circle by proving, conversely, that every function that can be computed by a Turing machine is recursive.

We will concentrate on the case of a one-place Turing-computable function, though our argument readily generalizes. Let us suppose, then, that f is a one-place function

computed by a Turing machine M. Let x be an arbitrary natural number. At the beginning of its computation of $f(x)$, M's tape will be completely blank except for a block of $x + 1$ strokes, representing the argument or input x. At the outset M is scanning the leftmost stroke in the block. When it halts, it is scanning the leftmost stroke in a block of $f(x) + 1$ strokes on an otherwise completely blank tape, representing the value or output $f(x)$. And throughout the computation there are finitely many strokes to the left of the scanned square, finitely many strokes to the right, and at most one stroke in the scanned square. Thus at any time during the computation, if there is a stroke to the left of the scanned square, there is a leftmost stroke to the left, and similarly for the right. We wish to use numbers to code a description of the contents of the tape. A particularly elegant way to do so is through the *Wang coding*. We use *binary notation* to represent the contents of the tape and the scanned square by means of a *pair of natural numbers*, in the following manner:

If we think of the blanks as zeros and the strokes as ones, then the infinite portion of the tape to the left of the scanned square can be thought of as containing a binary numeral (for example, 1011, or 1, or 0) prefixed by an infinite sequence of superfluous 0s. We call this numeral *the left numeral*, and the number it denotes in binary notation *the left number*. The rest of the tape, consisting of the scanned square and the portion to its right, can be thought of as containing a binary numeral *written backwards*, to which an infinite sequence of superfluous 0s is attached. We call this numeral, which appears backwards on the tape, *the right numeral*, and the number it denotes *the right number*. Thus the scanned square contains the digit in the unit's place of the right numeral. We take the right numeral to be written backwards to insure that changes on the tape will always take place in the vicinity of the unit's place of both numerals. If the tape is completely blank, then the left numeral = the right numeral = 0, and the left number = the right number = 0.

8.1 Example (The Wang coding). Suppose the tape looks as in Figure 8-1. Then the left numeral is 11101, the right numeral is 10111, the left number is 29, and the right number is 23. M now moves left, then the new left numeral is 1110, and the new left number is 14, while the new right numeral is 101111, and the new right number is 47.

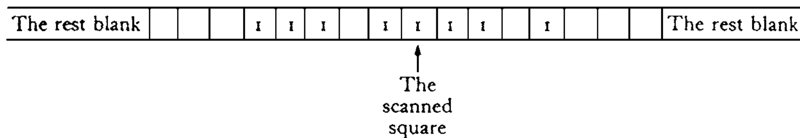

Figure 8-1. A Turing machine tape to be coded.

What are the left and right numbers when M begins the computation? The tape is then completely blank to the left of the scanned square, and so the left numeral is 0 and the left number is 0. The right numeral is $11 \dots 1$, a block of $x + 1$ digits 1. A sequence of m strokes represents in binary notation

$$2^{m-1} + \cdots + 2^2 + 2 + 1 = 2^m - 1.$$

Thus the right number at the start of M's computation of $f(x)$ will be

$$\text{strt}(x) = 2^{(x+1)} \dot{-} 1.$$

Note that strt is a primitive recursive function.

How do the left and right numbers change when M performs one step in the computation? That depends, of course, on what symbol is being scanned, as well as on what act is performed. How can we determine the symbol scanned? It will be a blank, or 0, if the binary representation of the right number ends in a 0, as is the case when the number is even, and a stroke, or 1, if the binary representation of the right number ends in a 1, as is the case when the number is odd. Thus in either case it will be the remainder on dividing the right number by two, or in other words, if the right number is r, then the symbol scanned will be

$$\text{scan}(r) = \text{rem}(r, 2).$$

Note that scan is a primitive recursive function.

Suppose the act is to *erase*, or put a 0 on, the scanned square. If there was already a 0 present, that is, if $\text{scan}(r) = 0$, there will be no change in the left or right number. If there was a 1 present, that is, if $\text{scan}(r) = 1$, the left number will be unchanged, but the right number will be decreased by 1. Thus if the original left and right numbers were p and r respectively, then the new left and new right numbers will be given by

$$\text{newleft}_0(p, r) = p$$
$$\text{newrght}_0(p, r) = r \dot{-} \text{scan}(r).$$

If instead the act is to *print*, or put a 1 on, the scanned square, there will again be no change in the left number, and there will be no change in the right number either if there was a 1 present. But if there was a 0 present, then the right number will be increased by 1. Thus the new left and new right number will be given by

$$\text{newleft}_1(p, r) = p$$
$$\text{newrght}_1(p, r) = r + 1 \dot{-} \text{scan}(r).$$

Note that all the functions here are primitive recursive.

What happens when M moves left or right? Let p and r be the old (pre-move) left and right numbers, and let $p*$ and $r*$ be the new (post-move) left and right numbers. We want to see how $p*$ and $r*$ depend upon p, r, and the direction of the move. We consider the case where the machine moves left.

If p is odd, the old numeral ends in a one. If $r = 0$, then the new right numeral is 1, and $r* = 1 = 2r + 1$. And if $r > 0$, then the new right numeral is obtained from the old by appending a 1 to it at its one's-place end (thus lengthening the numeral); again $r* = 2r + 1$. As for $p*$, if $p = 1$, then the old left numeral is just 1, the new left numeral is 0, and $p* = 0 = (p \dot{-} 1)/2 = \text{quo}(p, 2)$. And if p is any odd number greater than 1, then the new left numeral is obtained from the old by deleting the 1 in its one's place (thus shortening the numeral), and again $p* = (p \dot{-} 1)/2 = \text{quo}(p, 2)$. [In Example 8.1, for instance, we had $p = 29$, $p* = (29 - 1)/2 = 14$,

$r = 23$, $r^* = 2 \cdot 23 + 1 = 47$.] Thus we have established the first of the following two claims:

If M moves left	and p is odd	then $p^* = \text{quo}(p, 2)$	and $r^* = 2r + 1$
If M moves left	and p is even	then $p^* = \text{quo}(p, 2)$	and $r^* = 2r$.

The second claim is established in exactly the same way, and the two claims may be subsumed under the single statement that when M moves left, the new left and right numbers are given by

$$\text{newleft}_2(p, r) = \text{quo}(p, 2)$$
$$\text{newrght}_2(p, r) = 2r + \text{rem}(p, 2).$$

A similar analysis shows that if M moves right, then the new left and right numbers are given by

$$\text{newleft}_3(p, r) = 2p + \text{rem}(r, 2)$$
$$\text{newrght}_3(p, r) = \text{quo}(r, 2).$$

Again all the functions involved are primitive recursive. If we call printing 0, printing 1, moving left, and moving right acts numbers 0, 1, 2, and 3, then the new left number when the old left and right numbers are p and r and the act number is a will be given by

$$\text{newleft}(p, r, a) = \begin{cases} p & \text{if } a = 0 \text{ or } a = 1 \\ \text{quo}(p, 2) & \text{if } a = 2 \\ 2p + \text{rem}(r, 2) & \text{if } a = 3. \end{cases}$$

This again is a primitive recursive function, and there is a similar primitive recursive function newrght(p, r, a) giving the new right number in terms of the old left and right numbers and the act number.

And what are the left and right numbers when M halts? If M halts in *standard position* (or configuration), then the left number must be 0, and the right number must be $r = 2^{f(x)+1} \dot{-} 1$, which is the number denoted in binary notation by a string of $f(x) + 1$ digits 1. Then $f(x)$ will be given by

$$\text{valu}(r) = \text{lg}(r, 2).$$

Here lg is the primitive recursive function of Example 7.11, so valu is also primitive recursive. If we let nstd be the characteristic function of the relation

$$p \neq 0 \vee r \neq 2^{\text{lg}(r, 2)+1} \dot{-} 1$$

then the machine will be in standard position if and only if nstd$(p, r) = 0$. Again, since the relation indicated is primitive recursive, so is the function nstd.

So much, for the moment, for the topic of coding the contents of a Turing tape. Let us turn to the coding of Turing machines and their operations. We discussed the coding of Turing machines in section 4.1, but there we were working with positive integers and here we are working with natural numbers, so a couple of changes will be in order. One of these has already been indicated: we now number the acts 0 through 3 (rather than 1 through 4). The other is equally simple: let us now use

0 for the halted state. A Turing machine will then be coded by a finite sequence whose length is a multiple of four, namely $4k$, where k is the number of states of the machine (not counting the halted state), and with the even-numbered entries (starting with the initial entry, which we count as entry number 0) being numbers ≤ 3 to represent possible acts, while the odd-numbered entries are numbers $\leq k$, representing possible states. Or rather, a machine will be coded by a number coding such a finite sequence.

The instruction as to what *act* to perform when in state q and scanning symbol i will be given by entry number $4(q \dot{-} 1) + 2i$, and the instruction as to what *state* to go into will be given by entry number $4(q \dot{-} 1) + 2i + 1$. For example, the 0th entry tells what act to perform if in the initial state 1 and scanning a blank 0, and the 1st entry what state then to go into; while the 2nd entry tells what act to perform if in initial state 1 and scanning a stroke 1, and the 3rd entry what state then to go into. If the machine with code number m is in state q and the right number is r, so that the symbol being scanned is, as we have seen, given by scan(r), then the next action to be performed and new state to go into will be given by

$$\begin{aligned} \mathrm{actn}(m, q, r) &= \mathrm{ent}(m, 4(q \dot{-} 1) + 2 \cdot \mathrm{scan}(r)) \\ \mathrm{newstat}(m, q, r) &= \mathrm{ent}(m, (4(q \dot{-} 1) + 2 \cdot \mathrm{scan}(r)) + 1). \end{aligned}$$

These are primitive recursive functions.

We have discussed representing the tape contents at a given stage of computation by two numbers p and r. To represent the *configuration* at a given stage of computation, we need also to mention the state q the machine is in. The configuration is then represented by a triple (p, q, r), or by a single number coding such a triple. For definiteness let us use the coding

$$\mathrm{trpl}(p, q, r) = 2^p 3^q 5^r.$$

Then given a code c for the configuration of the machine, we can recover the left, state, and right numbers by

$$\mathrm{left}(c) = \mathrm{lo}(c, 2) \qquad \mathrm{stat}(c) = \mathrm{lo}(c, 3) \qquad \mathrm{rght}(c) = \mathrm{lo}(c, 5)$$

where lo is the primitive recursive function of Example 7.11. Again all the functions here are primitive recursive.

Our next main goal will be to define a primitive recursive function conf(m, x, t) that will give the code for the configuration after t stages of computation when the machine with code number m is started with input x, that is, is started in its initial state 1 on the leftmost of a block of $x + 1$ strokes on an otherwise blank tape. It should be clear already what the code for the configuration will be at the beginning, that is, after 0 stages of computation. It will be given by

$$\mathrm{inpt}(m, x) = \mathrm{trpl}(0, 1, \mathrm{strt}(x)).$$

What we need to analyse is how to get from a code for the configuration at time t to the configuration at time $t' = t + 1$.

Given the code number m for a machine and the code number c for the configuration at time t, to obtain the code number $c*$ for the configuration at time $t + 1$, we may

proceed as follows. First, apply left, stat, and rght to c to obtain the left number, state number, and right number p, q, and r. Then apply actn and newstat to m and r to obtain the number a of the action to be performed, and the number q^* of the state then to enter. Then apply newleft and newrght to p, r, and a to obtain the new left and right numbers p^* and r^*. Finally, apply trpl to p^*, q^*, and r^* to obtain the desired c^*, which is thus given by

$$c^* = \text{newconf}(m, c)$$

where newconf is a composition of the functions left, stat, rght, actn, newstat, newleft, newrght, and trpl, and is therefore a primitive recursive function.

The function conf(m, x, t), giving the code for the configuration after t stages of computation, can then be defined by primitive recursion as follows:

$$\begin{aligned}\text{conf}\,(m, x, 0) &= \text{inpt}(m, x)\\ \text{conf}\,(m, x, t') &= \text{newconf}\,(m, \text{conf}(m, x, t)).\end{aligned}$$

It follows that conf is itself a primitive recursive function.

The machine will be halted when stat(conf$(m, x, t)) = 0$, and will then be halted in standard position if and only if nstd(conf$(m, x, t)) = 0$. Thus the machine will be halted in standard position if and only if stdh$(m, x, t) = 0$, where

$$\text{stdh}(m, x, t) = \text{stat}(\text{conf}(m, x, t)) + \text{nstd}(\text{conf}(m, x, t)).$$

If the machine halts in standard configuration at time t, then the output of the machine will be given by

$$\text{otpt}(m, x, t) = \text{valu}(\text{rght}(\text{conf}(m, x, t))).$$

Note that stdh and otpt are both primitive recursive functions.

The time (if any) when the machine halts in standard configuration will be given by

$$\text{halt}(m, x) = \begin{cases}\text{the least } t \text{ such that stdh}(m, x, t) = 0 & \text{if such a } t \text{ exists}\\ \text{undefined} & \text{otherwise.}\end{cases}$$

This function, being obtained by minimization from a primitive recursive function, is a recursive partial or total function.

Putting everything together, let $F(m, x) = \text{otpt}(m, x, \text{halt}(m, x))$, a recursive function. Then $F(m, x)$ will be the value of the function computed by the Turing machine with code number m for argument x, if that function is defined for that argument, and will be undefined otherwise. If f is a Turing-computable function, then for some m—namely, for the code number of any Turing machine computing f—we have $f(x) = F(m, x)$ for all x. Since F is recursive, it follows that f is recursive. We have proved:

8.2 Theorem. A function is recursive if and only if it is Turing computable.

The circle is closed.

8.2 Universal Turing Machines

The connection we have established between Turing computability and recursiveness enables us to establish properties of each notion that it would have been more difficult to establish working with that notion in isolation. We begin with one example of this phenomenon pertaining to Turing machines, and one to recursive functions.

8.3 Theorem. The same class of functions are Turing computable whether one defines Turing machines to have a tape infinite in both directions or infinite in only one direction, and whether one requires Turing machines to operate with only one symbol in addition to the blank, or allows them to operate with any finite number.

Proof: Suppose we have a Turing machine M of the kind we have been working with, with a two-way infinite tape. In this chapter we have seen that the total or partial function f computed by M is recursive. In earlier chapters we have seen how a recursive function f can be computed by an abacus machine and hence by a Turing machine simulating an abacus machine. But the Turing machines simulating abacus machines are rather special: according to the problems at the end of Chapter 5, any abacus-computable function can be computed by a Turing machine *that never moves left of the square on which it is started*. Thus we have now shown that for any Turing machine there is another Turing machine computing the same function that uses only the right half of its tape. In other words, if we had begun with a more restrictive notion of Turing machine, where the tape is infinite in one direction only, we would have obtained the same class of Turing-computable functions as with our official, more liberal definition.

Inversely, suppose we allowed Turing machines to operate not only with the blank 0 and the stroke 1, but also with another symbol 2. Then in the proof of the preceding sections we would need to work with *ternary* rather than binary numerals, to code Turing machines by sequences of length a multiple of *six* rather than of four, and make similar minor changes. But with such changes, the proof would still go through, and show that any function computable by a Turing machine of this liberalized kind is still recursive—and therefore was computable by a Turing machine of the original kind already. The result generalizes to more than two symbols in an obvious way: for n symbols counting the blank, we need n-ary numerals and sequences of length a multiple of $2n$.

Similar, somewhat more complicated arguments show that allowing a Turing machine to work on a two-dimensional grid rather than a one-dimensional tape would not enlarge the class of functions that are computable. Likewise the class of functions computable would not be changed if we allowed the use of blank, 0, and 1, and redefined computations so that inputs and outputs are to be given in binary rather than stroke notation. That class is, as is said, *stable under perturbations of definition*, one mark of a *natural* class of objects.

8.4 Theorem (Kleene normal form theorem). Every recursive total or partial function can be obtained from the basic functions (zero, successor, identity) by composition, primitive recursion, and minimization, *using this last process no more than once*.

Proof: Suppose we have a recursive function f. We have seen in earlier chapters that f is computable by an abacus machine and hence by some Turing machine M. We have seen in this chapter that if m is the code number of M, then $f(x) = F(m, x)$ for all x, from which it follows that f can be obtained by composition from the constant function const_m, the identity function id, and the function F [namely, $f(x) = F(\text{const}_m(x), \text{id}(x))$, and therefore $f = \text{Cn}[F, \text{const}_m, \text{id}]$.] Now const_m and id are primitive recursive, and so obtainable from basic functions by composition and primitive recursion, without use of minimization. As for F, reviewing its definition, we see that minimization was used just once (namely, in defining $\text{halt}(m, x)$). Thus *any* recursive function f can be obtained using minimization only once.

An $(n + 1)$-place recursive function F with the property that for every n-place recursive function f there is an m such that

$$f(x_1, \ldots, x_n) = F(m, x_1, \ldots, x_n)$$

is called a *universal* function. We have proved the existence of a two-place universal function, and remarked at the outset that our arguments would apply also to functions with more places. A significant property of our two-place universal function, shared by the analogous many-place universal functions, is that its graph is a semirecursive relation. For $F(m, x) = y$ if and only if the machine with code number m, given input x, eventually halts in standard position, giving output y, which is to say, if and only if

$$\exists t(\text{stdh}(m, x, t) = 0 \,\&\, \text{otpt}(m, x, t) = y).$$

Since what follows the existential quantifier here is a primitive recursive relation, the graph relation $F(m, x) = y$ is obtainable by existential quantification from a primitive recursive relation, and therefore is semirecursive, as asserted. Thus we have the following.

8.5 Theorem. For every k there exists a universal k-place recursive function (whose graph relation is semirecursive).

This theorem has several substantial corollaries in the theory of recursive functions, but as these will not be essential in our later work, we have relegated them to an optional final section—in effect, an appendix—to this chapter. In the closing paragraphs of the present section, we wish to point out the implications of Theorem 8.5 for the theory of Turing machines. Of course, in the definition of universal function and the statement of the foregoing theorem we could have said ‘Turing-computable function’ in place of ‘recursive function’, since we now know these come to the same thing.

A Turing machine for computing a universal function is called a *universal Turing machine*. If U is such a machine (for, say, $k = 1$), then for any Turing machine M we like, the value computed by M for a given argument x will also be computed by U given a code m for M as a further argument in addition to x. Historically, as we have already mentioned, the theory of Turing computability (including the proof of the existence of a universal Turing machine) was established before (indeed, a decade or more before) the age of general-purpose, programmable computers, and

in fact formed a significant part of the theoretical background for the development of such computers. We can now say more specifically that the theorem that there exists a universal Turing machine, together with Turing's thesis that all effectively computable functions are Turing computable, heralded the arrival of the computer age by giving the first theoretical assurance that in principle *a general-purpose computer could be designed that could be made to mimic* any *special-purpose computer desired, simply by giving it coded instructions as to what machine it is to mimic as an additional input along with the arguments of the function we want computed.*

8.3* Recursively Enumerable Sets

An immediate consequence of Theorem 8.5 is the following converse to Proposition 7.17.

8.6 Corollary (Second graph principle). The graph relation of a recursive function is semirecursive.

Proof: If f is a recursive (total or partial) function, then there is an m such that $f(x) = F(m, x)$, where F is the universal function of the preceding section. For the graph relation of f we have

$$f(x) = y \leftrightarrow F(m, x) = y.$$

Hence, the graph relation of f is a section, in the sense of Problem 7.1, of the graph relation of F, which is semirecursive, and is therefore itself semirecursive.

At the beginning of this book we defined a set to be enumerable if it is the range of a total or partial function on the positive integers; and clearly we could have said 'natural numbers' in place of 'positive integers'. We now define a set of natural numbers to be *recursively enumerable* if it is the range of a total or partial *recursive* function on natural numbers. It turns out that we could say 'domain' here instead of 'range' without changing the class of sets involved, and that this class is one we have already met with under another name: the semirecursive sets. In the literature the name 'recursively enumerable' or 'r.e.' is more often used than 'semirecursive', though the two come to the same thing.

8.7 Corollary. Let A be a set of natural numbers. Then the following conditions are equivalent:

(a) A is the range of some recursive total or partial function.
(b) A is the domain of some recursive total or partial function.
(c) A is semirecursive.

Proof: First suppose A is semirecursive. Then the relation

$$Rxy \leftrightarrow Ax \,\&\, x = y$$

is semirecursive, since A is semirecursive, the identity relation is semirecursive, and semirecursive relations are closed under conjunction. But the relation R is the graph

relation of the restriction of the identity function to A, that is, of the function

$$\mathrm{id}_A(x) = \begin{cases} x & \text{if } Ax \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Since the graph relation is semirecursive, the function is recursive by Proposition 7.17. And A is both the range and the domain of id_A. Hence A is both the range of a recursive partial function and the domain of such a function.

Now suppose f is a recursive partial or total function. Then by Corollary 8.6 the graph relation $f(x) = y$ is semirecursive. Since semirecursive relations are closed under existential quantification, the following sets are also semirecursive:

$$Ry \leftrightarrow \exists x(f(x) = y)$$
$$Dx \leftrightarrow \exists y(f(x) = y).$$

But these sets are precisely the range and the domain of f. Thus the range and domain of any recursive function are semirecursive.

We have said quite a bit about recursively enumerable (or equivalently, semirecursive) sets without giving any examples of such sets. Of course, in a sense we *have* given many examples, since every recursive set is recursively enumerable. But are there any other examples? We are at last in a position to prove that there are.

8.8 Corollary. There exists a recursively enumerable set that is not recursive.

Proof: Let F be the universal function of Theorem 8.5, and let A be the set of x such that $F(x, x) = 0$. Since the graph relation of F is semirecursive, this set is also semirecursive (or equivalently, recursively enumerable). If it were recursive, its complement would also be recursive, which is to say, the characteristic function c of its complement would be a recursive function. But then, since F is a universal function, there would be an m such that $c(x) = F(m, x)$ for all x, and in particular, $c(m) = F(m, m)$. But since c is the characteristic function of the complement of A, we have $c(m) = 0$ if and only if m is *not* in A, which, by the definition of A, means if and only if $F(m, m)$ is *not* $= 0$ (is either undefined, or defined and > 0). This is a contradiction, showing that A cannot be recursive.

When we come to apply computability theory to logic, we are going to find that there are many more natural examples than this of recursively enumerable sets that are not recursive.

Problems

8.1 We proved Theorem 8.2 for one-place functions. For two-place (or many-place) functions, the only difference in the proof would occur right at the beginning, in defining the function strt. What is the right number at the beginning of a computation with arguments x_1 and x_2?

8.2 Suppose we liberalized our definition of Turing machine to allow the machine to operate on a two-dimensional grid, like graph paper, with vertical up and down

actions as well as horizontal left and right actions. Describe some reasonable way of coding a configuration of such a machine.

The remaining problems pertain to the optional section 8.3.

8.3 The (*positive*) *semicharacteristic function* of a set A is the function c such that $c(a) = 1$ if a is in A, and $c(a)$ is undefined otherwise. Show that a set A is recursively enumerable if and only if its semicharacteristic function is recursive.

8.4 A two-place relation S is called *recursively enumerable* if there are two recursive total or partial functions f and g with the same domain such that for all x and y we have $Sxy \leftrightarrow \exists t(f(t) = x \,\&\, g(t) = y)$. Show that S is recursively enumerable if and only if the set of all $J(x, y)$ such that Sxy is recursively enumerable, where J is the usual primitive recursive pairing function.

8.5 Show that any recursively enumerable set A can be defined in the form $Ay \leftrightarrow \exists w\, Ryw$ for some *primitive* recursive relation R.

8.6 Show that any nonempty recursively enumerable set A is the range of some *primitive* recursive function.

8.7 Show that any infinite recursively enumerable set A is the range of some *one-to-one* recursive total function.

8.8 A one-place total function f on the natural numbers is *monotone* if and only if whenever $x < y$ we have $f(x) < f(y)$. Show that if A is the range of a monotone recursive function, then A is recursive.

8.9 A pair of recursively enumerable sets A and B are called *recursively inseparable* if they are disjoint, but there is no recursive set C that contains A and is disjoint from B. Show that a recursively inseparable pair of recursively enumerable sets exists.

8.10 Give an example of a recursive partial function f such that f cannot be extended to a recursive total function, or in other words, such that there is no recursive total function g such that $g(x) = f(x)$ for all x in the domain of f.

8.11 Let R be a recursive relation, and A the recursively enumerable set given by $Ax \leftrightarrow \exists w\ Rxw$. Show that if A is not recursive, then for any recursive total function f there is an x in A such that the least 'witness' that x is in A (that is, the least w such that Rxw) is greater than $f(x)$.

8.12 Show that if f is a recursive *total* function, then there is a sequence of functions $f_1, \ldots, f_n$ with last item $f_n = f$, such that each either is a basic function (zero, successor, identity) or is obtainable from earlier functions in the sequence by composition, primitive recursion, or minimization, *and all functions in the sequence are total.*