Recursive Function Theory

John David Stone

November 20, 2019

Contents

Chapter 1

Introduction

One can think of a computer, as it executes a program, as a physical realization of a mathematical function—a mapping from the program's inputs to its output. One way to pursue the question of what the capabilities of computers are, therefore, is to try to delimit the set of functions that they can realize. Can *every* mathematical function be realized by some suitably programmed computer, or are there some that are so complicated or so irregular that they cannot be computed? If every mathematical function can be computed, is there some way to automate the process of writing a program to compute a given function? If there are mathematical functions that cannot be computed, is there some systematic way to distinguish those that can from those that cannot, or at least to prove that certain functions *cannot* be realized by suitably programmed computers?

To simplify these questions slightly, we shall start by considering only functions that take natural numbers as inputs and produce natural numbers as outputs. (We'll use the symbol '$\mathbb{N}$' for the set $\{0, 1, 2, \ldots\}$ of non-negative integers and the term 'natural numbers' for its members.) Of course, suitably programmed computers are capable of computing functions on inputs of various types, such as signed integers, Booleans, characters, strings, lists, vectors, and so on. However, our theorems will be no less general if we initially restrict ourselves to functions that operate on natural numbers. As we'll see, it is straightforward to encode inputs of other types as natural numbers and decode the natural-number outputs into such other types, using techniques that are themselves readily computable.

Many programming languages also purport to offer real numbers as a data type, but this is a misnomer. The "real" values that figure in computer programs are approximations—approximations of good quality, in most cases, but approximations nevertheless. In most applications, this inexactness is harmless. It is generally impossible to measure continuous quantities perfectly to begin with, and modelling these inexact inputs with slightly inexact representations of real numbers usually makes little or no difference in the utility of the outputs. From the mathematician's point of view, however, the approximations on which computers operate constitute only a tiny subset of the real numbers. It would

be possible, without loss of information, to encode each of those approximations as a different natural number, treating them as data structures similar to those mentioned above.

Some programming languages, such as Scheme, perform computations that take *procedures* as inputs and produce procedures as outputs. Here, too, we would lose no generality if we could only figure out a way to encode mathematical functions (the very ones that we are studying, in fact) as natural numbers and to decode natural numbers back into mathematical functions. In this case, however, it is far from obvious that such an encoding is possible. This is a question that we'll have to study carefully in a later section.

Exercises

1–1 A deck of playing cards comprises one card of each of thirteen ranks (ace, king, queen, jack, ten, nine, eight, seven, six, five, four, trey, and deuce) in each of four suits (spades, hearts, diamonds, and clubs)—fifty-two different cards altogether. Suggest a way of encoding each of these cards as a unique natural number in the range from 0 to 51 (inclusive).

1–2 Given a natural number in the range from 0 to 51, how would you figure out which card it encodes, in the encoding system you proposed in your solution to the preceding exercise?

1–3 When all of the cards in a deck are assembled in a stack, we can think of them as being arranged in a linear order, from the top of the stack to the bottom. Suggest a way of encoding any such arrangement of a deck of playing cards as a unique natural number, and a way of decoding a given natural number to determine which arrangement it encodes (if any).

1–4 There is a mathematical function that maps every rational number to 1 and every irrational number to 0. Could a suitably programmed computer realize this function?

Chapter 2

Functions

Since so many different kinds of data can be encoded as natural numbers, we'll initially consider only functions that take natural numbers as inputs and produce natural numbers as outputs. We'll consider functions with any number of inputs: *singulary* functions that take one input, *binary* ones that take two, *ternary* ones that take three, and so on, as well as *nullary* ones that take no inputs. Giving the valence (the number of inputs) of a function is therefore sufficient to specify its domain: The domain of a function of valence n is $\mathbb{N}^n$. Also, the range of the functions that we'll be considering is always the same: it's $\mathbb{N}$.

Usually, when a mathematician wants to specify a function, she writes a kind of defining equation for it, like this:

$$f(x, y) = x^2 - 2xy + y^2$$

The left-hand side of such an equation indicates the function's valence (in this case, 2) and gives a name to each of the function's inputs, and the right-hand side is a recipe for constructing the output from those inputs.

Because of the open-ended nature of mathematical notation, however, the boundaries of the set of functions that can be defined by means of such equations are somewhat vague. Moreover, it's not immediately apparent that every mathematical expression that could appear on the right-hand side of such an equation really corresponds to a computational procedure. What if that expression includes some bizarre integral that doesn't have a closed form, or asks us to find the limit of some series that has not been proven to converge? In order to use mathematical functions to model computation, we shall need a more disciplined notation, one that guarantees that the computational process denoted by any expression consists of simple, effective steps.

Such a notation would be similar to a small, tightly structured, high-level programming language. The core of such a language consists of a small collection of primitives and a finite collection of mechanisms for combining them so as to define computational procedures of increasing complexity and subtlety. Ideally, the mechanisms of combination should apply not only to the primitives, but also, recursively, to the products of previous acts of construction. In that way,

we can leverage the power of a finite core language so as to generate an infinite range of expressions.

Exercises

2–1 Give an example of a ternary function. Write an equation that specifies (on its right-hand side) how the function's output depends on its inputs.

2–2 Give an example of a nullary function. Write an equation that indicates its valence, 0, and specifies its output.

2–3 Would the programming language that you know best be a suitable notation for defining mathematical functions? Justify your answer.

Chapter 3

Primitive Recursive Functions

In developing a notation well suited to the formal definition of mathematical functions, we begin by giving names to certain *primitive functions* that are so simple that one might consider them trivial:

- `zero`, a nullary function that produces the natural number 0 as its output.
- `successor`: a singulary function that produces as output the successor of its input. For instance, given the natural number 5 as input, `successor` outputs 6.
- For any natural numbers m and n such that $m < n$, $\mathtt{pr}_n^m$: a function of valence n that outputs, without change, the input that it receives in (zero-based) position m. For instance, given the inputs 12, 19, 16, and 5, $\mathtt{pr}_4^2$ outputs 16.

We can describe these functions equationally, in conventional mathematical notation:

$$\begin{aligned} \mathtt{zero}() &= 0, \\ \mathtt{successor}(x) &= x + 1, \\ \mathtt{pr}_n^m(x_0, \ldots, x_{n-1}) &= x_m. \end{aligned}$$

In this context, however, such equations are not definitions, but mere expository notes, helping readers to understand exactly which functions we're talking about. Rather, these are the functions that we're going to be using to define everything else (including, for instance, the addition operation). We have to be able to identify and understand the primitive functions without formal definitions; otherwise, our definitions will be circular.

We posit that all of these functions can be computed. Since natural numbers can be arbitrarily large, this assumption already goes somewhat beyond what

is physically possible. Finding the successor of a natural number containing $2^{1000000}$ bits, for instance, is not a straightforward operation on a real-world computer. We know *how* to do the computation, but we don't have enough storage, enough electrical power, or enough time actually to carry it out. Since these limitations on the physical representation of natural numbers are not relevant to the *theoretical* possibility or impossibility of computation, however, we shall ignore them.

Using these primitive functions, we shall define a number of others, using two modes of combination:

- Composition: If we are given a function f of valence m and m functions $g_0, \ldots, g_{m-1}$ that all have the same valence n, we can define a new function h of valence n by composition. In conventional mathematical notation, the equation specifying the function h obtained in this way would look like this:

$$h(x_0, \ldots, x_{n-1}) = f(g_0(x_0, \ldots, x_{n-1}), \ldots, g_{m-1}(x_0, \ldots, x_{n-1})).$$

  In other words, given n natural numbers as inputs, one computes h's output by providing all n of the given values as inputs to each of the functions $g_0, \ldots, g_{m-1}$, collecting the m outputs, providing them as inputs to f, and taking the resulting output from f to be the output from h. We shall write such compositions in a more concise notation, thus:

$$\texttt{[}\circ_n^m \;\; f \;\; g_0 \;\; \ldots \;\; g_{m-1}\texttt{]}$$

  The symbol '$\circ_n^m$' expresses the operation of composition. The superscript '^{m}' indicates the valence of f (and the number of g-functions from which it receives its inputs). The subscript '$_n$' indicates the valence of each of the g-functions (and the valence of the composite function h).

- Recursion: Given a function f of valence n and a function g of valence $n+2$, we can define a new function h of valence $n+1$ by recursion over natural numbers. In conventional mathematical notation, the specification of such a function is set out in a pair of recursion equations:

$$\begin{aligned} h(x_0, \ldots, x_{n-1}, 0) &= f(x_0, \ldots, x_{n-1}), \\ h(x_0, \ldots, x_{n-1}, t+1) &= g(t, h(x_0, \ldots, x_{n-1}, t), x_0, \ldots, x_{n-1}) \\ &\qquad \text{for every } t \in \mathbb{N}. \end{aligned}$$

  The definition of h is recursive with respect to the last input. The output for the base case, in which the last input to h is 0, is computed by applying f to all of the other inputs; the output for the inductive step, where the last input is some positive integer $t+1$, is computed by applying g to the predecessor t of the last input, the value $h(x_0, \ldots, x_{n-1}, t)$ of h at the previous step, and the remaining inputs $x_0, \ldots, x_{n-1}$.

Once again, we'll write such specifications in a much more concise way:

$$[\Upsilon_n \;\; f \;\; g]$$

The symbol 'Υ_n' expresses the operation of recursion, with the subscript '$_n$' indicating the valence of the base-case function f (and requiring that the valence of g is $n+2$, and the valence of the recursively defined function h is $n+1$).

A *primitive recursive* function is one that can be built up from these primitive functions using only composition and recursion. The set of primitive recursive functions is surprisingly extensive and diverse. Let's dig in and build some of the more interesting ones.

Addition

Theorem 3.1 *Addition is a primitive recursive function.*

Proof: Conventionally, addition is defined by the recursion equations

$$\begin{aligned} \mathtt{add}(x, 0) &= x, \\ \mathtt{add}(x, t+1) &= \mathtt{successor}(\mathtt{add}(x, t)). \end{aligned}$$

In terms of our primitive functions, these equations could be written thus:

$$\begin{aligned} \mathtt{add}(x, 0) &= \mathtt{pr}_1^0(x), \\ \mathtt{add}(x, t+1) &= \mathtt{successor}(\mathtt{pr}_3^1(t, \mathtt{add}(x, t), x)). \end{aligned}$$

Hence we can use composition and recursion to define `add` in terms of the primitives:

$$\mathtt{add} \equiv [\Upsilon_1 \;\; \mathtt{pr}_1^0 \;\; [\circ_3^1 \;\; \mathtt{successor} \;\; \mathtt{pr}_3^1]].$$

Since `successor` and $\mathtt{pr}_3^1$ are primitive functions, and are therefore primitive recursive, $[\circ_3^1 \;\; \mathtt{successor} \;\; \mathtt{pr}_3^1]$ is also primitive recursive. Since $\mathtt{pr}_1^0$ is primitive (and therefore primitive recursive) and $[\circ_3^1 \;\; \mathtt{successor} \;\; \mathtt{pr}_3^1]$ is primitive recursive, `add` is also primitive recursive. ■

Doubling

Theorem 3.2 *Doubling is a primitive recursive function.*

Proof: To double a number, add it to itself:

$$\begin{aligned} \mathtt{double}(x) &= \mathtt{add}(x, x) \\ &= \mathtt{add}(\mathtt{pr}_1^0(x), \mathtt{pr}_1^0(x)). \end{aligned}$$

Thus we can define the singulary function `double` by composition:

$$\texttt{double} \equiv [\circ_1^2\ \texttt{add}\ \texttt{pr}_1^0\ \texttt{pr}_1^0].$$

The theorem follows by Theorem 3.1 and the definition of 'primitive recursive'. ∎

Multiplication

Theorem 3.3 *Multiplication is a primitive recursive function.*

Proof: The recursion equations for the binary function `multiply` are:

$$\begin{aligned} \texttt{multiply}(x, 0) &= 0, \\ \texttt{multiply}(x, t+1) &= x \cdot t + x, \end{aligned}$$

or, in terms of our primitive functions,

$$\begin{aligned} \texttt{multiply}(x, 0) &= \texttt{zero}() \\ &= [\circ_1^0\ \texttt{zero}](x), \\ \texttt{multiply}(x, t+1) &= \texttt{add}(\texttt{multiply}(x, t), x), \\ &= \texttt{add}(\texttt{pr}_3^1(t, \texttt{multiply}(x, t), x), \\ &\qquad \texttt{pr}_3^2(t, \texttt{multiply}(x, t), x)), \end{aligned}$$

so that

$$\texttt{multiply} \equiv [\Upsilon_1\ [\circ_1^0\ \texttt{zero}]\ [\circ_3^2\ \texttt{add}\ \texttt{pr}_3^1\ \texttt{pr}_3^2]].$$

∎

The notation '$[\circ_1^0\ \texttt{zero}]$' may seem a little puzzling. The idea is that, in the base case of the recursion that defines `multiply`, we want a singulary function that yields the output 0 no matter what input it is given (reflecting the fact that multiplying any natural number by 0 yields 0 as the product). The function `zero` itself doesn't quite do the job, since it is nullary rather than singulary. But we can "compose" it with zero singulary functions to obtain a singulary function, which in conventional mathematical notation would be specified by the composition equation

$$[\circ_1^0\ \texttt{zero}](x) = \texttt{zero}().$$

This is the special case of the general composition equation for $m = 0$ and $n = 1$, with f being `zero`.

Let's adopt the concise notation '$\texttt{zero}_n$' for '$[\circ_n^0\ \texttt{zero}]$', the function of valence n that ignores its inputs and always outputs 0. The we can define `multiply` thus:

$$\texttt{multiply} \equiv [\Upsilon_1\ \texttt{zero}_1\ [\circ_3^2\ \texttt{add}\ \texttt{pr}_3^1\ \texttt{pr}_3^2]].$$

More generally, whenever f is a function of valence 0, we'll write f_n for a function of valence n defined by

$$f_n \equiv [\circ_n^0 \ f].$$

Exponentiation

Theorem 3.4 *Exponentiation is a primitive recursive function.*

Proof: The recursion equations for the exponentiation function, which we'll call `raise-to-power`, are quite similar to those for `multiply`:

$$\begin{aligned} \texttt{raise-to-power}(x, 0) &= 1, \\ \texttt{raise-to-power}(x, t+1) &= \texttt{multiply}(\texttt{raise-to-power}(x, t), x). \end{aligned}$$

For the first equation, we need a singulary function that always outputs 1. It seems natural to define `one` by taking the successor of the output of `zero`, thus:

$$\texttt{one} \equiv [\circ_0^1 \ \texttt{successor zero}],$$

but this again makes `one` a nullary function, just as `zero` is. Hence the function we really need is $\texttt{one}_1$ (that is, $[\circ_1^0$ `one`$]$, the composition of `one` with zero singulary functions).

(The function $\texttt{one}_1$ could also be expressed as $[\circ_1^1$ `successor` $\texttt{zero}_1]$. If you write out the composition equations and simplify, you'll find that in either case you wind up with a singulary function that outputs 1 no matter what its input is.)

Thus the definition of `raise-to-power` is

$$\texttt{raise-to-power} \equiv [\Upsilon_1 \ \texttt{one}_1 \ [\circ_3^2 \ \texttt{multiply} \ \texttt{pr}_3^1 \ \texttt{pr}_3^2]].$$

■

Predecessor and Subtraction

The inverse of `successor` would be a function that takes every positive integer into its predecessor. If we're willing to accept the arbitrary but not too implausible convention that an attempt to take the predecessor of 0 yields 0, we can define a suitable predecessor function using recursion.

Theorem 3.5 *The predecessor function is primitive recursive.*

Proof: The recursion equations are

$$\begin{aligned} \texttt{predecessor}(0) &= 0, \\ \texttt{predecessor}(t+1) &= t \\ &= \texttt{pr}_2^0(t, \texttt{predecessor}(t)), \end{aligned}$$

or, in our notation,

$$\texttt{predecessor} \equiv [\Upsilon_0\ \texttt{zero}\ \texttt{pr}_2^0].$$

■

With the help of the predecessor function, we can define a subtraction function. Strictly speaking, ordinary subtraction is not closed over the natural numbers; since we want a function that has $\mathbb{N}$ as its range, we'll continue the policy of having 0 play an additional role as a kind of error result, so that "subtracting" a greater number from a lesser one yields 0.

Theorem 3.6 *Subtraction is a primitive recursive function.*

Proof: Here are the recursion equations for this slightly offbeat subtraction:

$$\begin{aligned}
\texttt{subtract}(x, 0) &= x \\
&= \texttt{pr}_1^0(x), \\
\texttt{subtract}(x, t+1) &= \texttt{predecessor}(\texttt{subtract}(x, t)) \\
&= \texttt{predecessor}(\texttt{pr}_3^1(t, \texttt{subtract}(x, t), x)).
\end{aligned}$$

So our definition is

$$\texttt{subtract} \equiv [\Upsilon_1\ \texttt{pr}_1^0\ [\circ_3^1\ \texttt{predecessor}\ \texttt{pr}_3^1]].$$

■

In mathematics, this function is sometimes called "monus" and represented by the infix operator '$\dot{-}$'.

It turns out that the policy of having the results of the `subtract` function "bottom out" at 0 has some technical advantages. For instance, it gives us an easy way to define a function that computes the *disparity* between two natural numbers, that is, the result of subtracting the lesser number from the greater one (or 0, if the inputs are equal).

Theorem 3.7 *Disparity is a primitive recursive function.*

Proof: Since the result will be 0 when we do the subtraction the "wrong" way, we can simply do it both ways and add the results:

$$\begin{aligned}
\texttt{disparity}(x_0, x_1) &= (x_0 \dot{-} x_1) + (x_1 \dot{-} x_0) \\
&= \texttt{add}(\texttt{subtract}(x_0, x_1), \texttt{subtract}(x_1, x_0)) \\
&= \texttt{add}(\texttt{subtract}(x_0, x_1), \\
&\qquad \texttt{subtract}(\texttt{pr}_2^1(x_0, x_1), \texttt{pr}_2^0(x_0, x_1))),
\end{aligned}$$

so the definition is

$$\texttt{disparity} \equiv [\circ_2^2\ \texttt{add}\ \texttt{subtract}\ [\circ_2^2\ \texttt{subtract}\ \texttt{pr}_2^1\ \texttt{pr}_2^0]].$$

■

Exercises

3–1 Name the projection function that outputs the next-to-last of its seven inputs.

3–2 Define the `square` function that maps every natural number to its square, using the formal notation introduced in this section.

3–3 Prove that the `square` function is primitive recursive.

3–4 Define the `factorial` function on natural numbers, using the formal notation introduced in this section. Prove that the `factorial` function is primitive recursive.

3–5 Prove that the function f specified in conventional mathematical notation by the equation

$$f(x, y) = x^2 + 2xy + y^2$$

is primitive recursive.

Chapter 4

Modelling Boolean Values

The universe of natural numbers is large and rich enough that we can use it to represent values of other data types in systematic ways.

Let's begin with Booleans. There are actually at least two reasonably natural conventions for representing Boolean values: We could choose two specific natural numbers, most plausibly 0 and 1, to represent the two values of the Boolean data type. Alternatively, we could follow the example set by C and single out 0 as the sole representation of falsity, while allowing all other natural numbers to be treated as equally valid representations of truth. This would have the nice feature that disjunction could be simulated by addition and conjunction by multiplication. Also, *any* function could then be used as a predicate.

We'll split the difference between these two approaches, following the design principle known as Postel's law: Be liberal in what you accept as input and conservative in what you provide as output. In this case, the principle implies that we should allow any positive integer to count as a "truish" value in Boolean contexts; but the Boolean functions that we define will output only the values 0 and 1, and we'll reserve the term 'predicate' for functions that meet this restriction.

One simple example of a primitive recursive function that is a predicate is the singulary function `zero?`, which outputs 1 if its input is 0 and 0 otherwise.

Theorem 4.1 *The predicate* `zero?` *is a primitive recursive function.*

Proof: Its recursion equations are

$$\begin{aligned} \texttt{zero?}(0) &= 1, \\ \texttt{zero?}(t+1) &= 0, \end{aligned}$$

which yields the definition

$$\texttt{zero?} \equiv [\Upsilon_0\ \texttt{one}\ \texttt{zero}_2].$$

■

The function $\texttt{zero}_2$, of course, is the *binary* function that one obtains by composing `zero` with zero binary functions. That's the way the valences have to be in order to make the recursion rule work—to define a singulary function by recursion, the function that performs the inductive step must be binary, even if it ignores both of its inputs (which in this case would be t and $\texttt{zero?}(t)$).

Given any natural number as input, `zero?` outputs the canonical representative of the the opposite Boolean value, so it can also be thought of as an implementation of the Boolean operator `not`.

Theorem 4.2 *Boolean negation is a primitive recursive function.*

Proof:

$$\texttt{not} \equiv \texttt{zero?}$$

■

A natural number is positive just in case it is not zero. This gives us an easy way to define the predicate `positive?`.

Theorem 4.3 *The predicate* `positive?` *is a primitive recursive function.*

Proof:

$$\texttt{positive?} \equiv [\circ_1^1\ \texttt{not zero?}]$$

■

We'll sometimes use the name '`truish?`' for the same function.

Here is a slightly more complicated example:

Theorem 4.4 *The predicate* `even?` *is a primitive recursive function.*

Proof: The recursion equations for `even?` are

$$\begin{aligned} \texttt{even?}(0) &= 1, \\ \texttt{even?}(t+1) &= \texttt{not}(\texttt{even?}(t)), \end{aligned}$$

which yield the definition

$$\texttt{even?} \equiv [\curlyvee_0\ \texttt{one}\ [\circ_2^1\ \texttt{not}\ \texttt{pr}_2^1]].$$

■

The product of two numbers is truish if and only if both of them are truish, so multiplication could serve as the `and` function; but, in accordance with the convention announced above, we'll force the result to be either 0 or 1.

Theorem 4.5 *Conjunction is a primitive recursive function.*

Proof:

$$\texttt{and} \equiv [\circ_2^1\ \texttt{truish?}\ \texttt{multiply}].$$

■

The disjunction of two Boolean values is the negation of the conjunction of their negations.

Theorem 4.6 *Disjunction is a primitive recursive function.*

Proof:

$$\begin{aligned} \texttt{or}(x_0, x_1) &= \texttt{not}(\texttt{and}(\texttt{not}(x_0), \texttt{not}(x_1))) \\ &= \texttt{not}(\texttt{and}(\texttt{not}(\texttt{pr}_2^0(x_0, x_1)), \\ &\qquad \texttt{not}(\texttt{pr}_2^1(x_0, x_1)))) \end{aligned}$$

so that

$$\texttt{or} \equiv [\circ_2^1\ \texttt{not}\ [\circ_2^2\ \texttt{and}\ [\circ_2^1\ \texttt{not}\ \texttt{pr}_2^0]\ [\circ_2^1\ \texttt{not}\ \texttt{pr}_2^1]]].$$

■

Alternatively, we could define `or` as $[\circ_2^1\ \texttt{truish?}\ \texttt{add}]$. The same function is obtained using either construction.

Conditionals

In constructing primitive recursive functions, we shall often want to define them *by cases*, applying some test to the given inputs to determine which of two functions to apply to them. Specifically, if p, f, and g are primitive recursive functions with the same valence n, then we should be able to define a new function h of valence n that matches f on any inputs that satisfy p (that is, any inputs for which p outputs truish values) and matches g on inputs that don't satisfy p.

It turns out, however, that adding this construction as a third mechanism for defining primitive recursive functions is superfluous, because we can simulate it arithmetically.

Theorem 4.7 *For any primitive recursive predicate p and any primitive recursive functions f and g having the same valence as p, the function h defined by the case-construction*

$$h(x_0, \ldots, x_{n-1}) = \begin{cases} f(x_0, \ldots, x_{n-1}) & \text{if } p(x_0, \ldots, x_{n-1}) > 0, \\ g(x_0, \ldots, x_{n-1}) & \text{otherwise.} \end{cases}$$

is primitive recursive.

Proof: The idea is to multiply the value that f yields by the value that p yields, and the value that g yields by the opposite Boolean value. One or the other of the opposite Boolean values will be 0, and hence one of these products is also 0; the other Boolean value will be 1, so that the other product is simply the value of the non-Boolean multiplicand. Hence, if we add the two products

together, we'll be adding 0 to the value output by the selected function—f if p yields 1, g if it yields 0. In other words:

$$\begin{aligned} h(x_0, \ldots, x_{n-1}) &= f(x_0, \ldots, x_{n-1}) \cdot \texttt{truish?}(p(x_0, \ldots, x_{n-1})) + \\ & \quad g(x_0, \ldots, x_{n-1}) \cdot \texttt{not}(p(x_0, \ldots, x_{n-1})). \end{aligned}$$

So we can define h thus:

$$\begin{aligned} h \equiv\ & [\circ_n^2\ \texttt{add} \\ & \quad [\circ_n^2\ \texttt{multiply}\ f\ [\circ_n^1\ \texttt{truish?}\ p]] \\ & \quad [\circ_n^2\ \texttt{multiply}\ g\ [\circ_n^1\ \texttt{not}\ p]]]. \end{aligned}$$

■

We'll use expressions of the form '$[¿_n\ p\ f\ g]$' for functions defined by cases in this way. Note carefully, however, what Theorem 4.7 tells us about this addition to our notation: It is convenient, but formally superfluous. If necessary, we could rewrite any expression that uses '$¿_n$' so as to use only composition and recursion operations. The new convention allows us to specify primitive recursive functions more concisely, but it does not extend or enrich the underlying concept of primitive recursivity.

Exercises

4–1 The *exclusive-or* function takes two Boolean inputs and outputs the true Boolean value if one and only one of the inputs is true. It outputs the false Boolean value if both inputs are true or both false. Model this function using natural numbers and show that the `exclusive-or` function of your model is primitive recursive.

4–2 The *duplex* function takes three Boolean inputs and outputs the second input if the first input is true. It outputs its third input if the first input is false. Model this function using natural numbers and show that the `duplex` function of your model is primitive recursive.

4–3 Define a function `halve` that divides its input by 2, discarding the remainder and outputting the quotient. (So, for instance, $\texttt{halve}(7) = 3$.) Hint: Use a direct recursion in which the step function is a conditional.

4–4 Define a function `collatz-step` that, given a natural number n as input, outputs $\texttt{halve}(n)$ if n is even, or $3n + 1$ if n is odd. Prove that `collatz-step` is primitive recursive.

4–5 Define a binary, primitive recursive function `positive-count` that outputs 0 if both of its inputs are 0, outputs 1 if one of its inputs is 0 and the other is positive, and outputs 2 if both of its inputs are positive.

Chapter 5

More Arithmetic Functions

Comparison Predicates

Theorem 5.1 *Equality is a primitive recursive function.*

Proof: We can test whether two numbers are equal by checking whether their disparity is 0:

$$\texttt{equal?} \equiv [\circ_2^1\ \texttt{zero?}\ \texttt{disparity}].$$

■

Theorem 5.2 *The predicate* `less-or-equal?` *is a primitive recursive function.*

Proof: We can test whether one number is less than or equal to another by subtracting (using "monus" subtraction) and checking whether the result is 0:

$$\texttt{less-or-equal?} \equiv [\circ_2^1\ \texttt{zero?}\ \texttt{subtract}].$$

■

Theorem 5.3 *The predicate* `greater-or-equal?` *is a primitive recursive function.*

Proof: To reverse the direction of the comparison, we can use projection functions to swap the operands:

$$\begin{aligned} \texttt{greater-or-equal?}(x_0, x_1) &= \texttt{less-or-equal?}(x_1, x_0) \\ &= \texttt{less-or-equal?}(\texttt{pr}_2^1(x_0, x_1), \texttt{pr}_2^0(x_0, x_1)), \end{aligned}$$

so that the definition is

$$\texttt{greater-or-equal?} \equiv [\circ_2^2\ \texttt{less-or-equal?}\ \texttt{pr}_2^1\ \texttt{pr}_2^0].$$

■

The remaining inequality predicates are the negations of the preceding ones:

$$\begin{aligned} \texttt{less?} &\equiv [\circ_2^1\ \texttt{not greater-or-equal?}], \\ \texttt{greater?} &\equiv [\circ_2^1\ \texttt{not less-or-equal?}], \\ \texttt{unequal?} &\equiv [\circ_2^1\ \texttt{not equal?}]. \end{aligned}$$

Division

Dividing one member of $\mathbb{N}$ by another yields two results: a quotient and a remainder. Since applying a function yields a single result, we will consider division as comprising two functions, one yielding the quotient and the other the remainder.

Theorem 5.4 *The function* `remainder` *is primitive recursive.*

Proof: It turns out to be simpler to write recursion equations for the remainder if we initially write the divisor on the left and the dividend in the recursion slot, on the right. I'll call this version of the remainder function `redniamer`, since its inputs are (from the conventional point of view) backwards:

$$\begin{aligned} \texttt{redniamer}(x_0, 0) &= 0, \\ \texttt{redniamer}(x_0, t+1) &= \begin{cases} 0 & \text{if } \texttt{redniamer}(x_0, t) + 1 = x_0 \\ \texttt{redniamer}(x_0, t) + 1 & \text{otherwise,} \end{cases} \end{aligned}$$

or, making the projections more explicit,

$$\begin{aligned} \texttt{redniamer}(x_0, 0) &= \texttt{zero}(), \\ \texttt{redniamer}(x_0, t+1) &= \begin{cases} \texttt{zero}() \\ \quad \text{if } \texttt{equal?}(\texttt{pr}_3^2(t, \texttt{redniamer}(x_0, t), x_0), \\ \quad\quad \texttt{successor}(\texttt{pr}_3^1(t, \texttt{redniamer}(x_0, t), x_0))) \\ \texttt{successor}(\texttt{pr}_3^1(t, \texttt{redniamer}(x_0, t), x_0)) \\ \quad \text{otherwise.} \end{cases} \end{aligned}$$

Thus

$$\begin{aligned} \texttt{redniamer} \equiv [\Upsilon_1\ \texttt{zero}_1\ [\text{¿}_3\ [\circ_3^2\ \texttt{equal?}\ \texttt{pr}_3^2\ [\circ_3^1\ \texttt{successor}\ \texttt{pr}_3^1]] \\ \texttt{zero}_3 \\ [\circ_3^1\ \texttt{successor}\ \texttt{pr}_3^1]]]. \end{aligned}$$

To put the inputs in the more usual order (dividend as the first input, divisor as the second), we compose `redniamer` with projection functions:

$$\texttt{remainder} \equiv [\circ_2^2\ \texttt{redniamer}\ \texttt{pr}_2^1\ \texttt{pr}_2^0].$$

■

Under this definition, incidentally, division by 0 is permitted, but it leaves the entire dividend as the remainder. (The test that "resets" the remainder-counter to 0 never succeeds when the divisor is 0.)

Theorem 5.5 *Divisibility is a primitive recursive function.*

Proof: The primitive recursive predicate `divides?` tests whether its first input evenly divides its second:

$$\texttt{divides?}(x_0, x_1) \quad = \quad \texttt{zero?}(\texttt{redniamer}(x_0, x_1)).$$

or, using our notation,

$$\texttt{divides?} \equiv [\circ_2^2\ \texttt{zero?}\ \texttt{redniamer}].$$

■

Theorem 5.6 *The function* `quotient` *is primitive recursive.*

Proof: We can write recursion equations similar to those for `redniamer`, again beginning with a version in which the divisor is the first input and the dividend the second:

$$\begin{aligned} \texttt{tneitouq}(x_0, 0) &= 0, \\ \texttt{tneitouq}(x_0, t+1) &= \begin{cases} \texttt{successor}(\texttt{tneitouq}(x_0, t)) \\ \qquad \text{if } \texttt{divides?}(x_0, \texttt{successor}(t)), \\ \texttt{tneitouq}(x_0, t) \qquad \text{otherwise,} \end{cases} \end{aligned}$$

which yields the definition

$$\begin{aligned} \texttt{tneitouq} \equiv [\Upsilon_1\ \texttt{zero}_1\ [\text{¿}_3\ &[\circ_3^2\ \texttt{divides?}\ \texttt{pr}_3^2\ [\circ_3^1\ \texttt{successor}\ \texttt{pr}_3^0]] \\ &[\circ_3^1\ \texttt{successor}\ \texttt{pr}_3^1] \\ &\texttt{pr}_3^1]]. \end{aligned}$$

The `quotient` function, once again, can be derived from `tneitouq` by swapping the inputs:

$$\texttt{quotient} \equiv [\circ_2^2\ \texttt{tneitouq}\ \texttt{pr}_2^1\ \texttt{pr}_2^0].$$

■

Exercises

5–1 Define a ternary, primitive recursive predicate `between?` that determines whether its second input lies between its first and third inputs—in other words, whether the second input is greater than one of the remaining inputs and less than the other. Equality doesn't count—`between?` should output 0 if any two of its inputs are equal.

5–2 Define a ternary, primitive recursive predicate `common-divisor?` that determines whether its first input evenly divides both the second and third inputs.

5–3 Like `remainder`, `quotient` outputs a value even when its second input is 0—again, "division" by 0 is permitted. What value does `quotient` output in such cases? In other words, what is `quotient`$(n, 0)$?

Chapter 6

Accumulators

Bounded Quantifiers

For any function p of valence $n+1$, there is a *cumulative* predicate $\hat{p}$ of the same valence that is satisfied by some $(n+1)$-tuple $(x_0, \ldots, x_{n-1}, x_n)$ of natural numbers if, and only if, p outputs a truish value for *all* the $(n+1)$-tuples of the form $(x_0, \ldots x_{n-1}, x)$ such that $x \leq x_n$. The cumulative predicate is like an extended conjunction of applications of p to $(n+1)$-tuples with smaller last elements. The operation that converts p into $\hat{p}$ is called *bounded quantification*.

Theorem 6.1 *For any primitive recursive function p, the predicate $\hat{p}$ defined by*

$$\hat{p}(x_0, \ldots, x_{n-1}, x_n) = \begin{cases} 1 & \textit{if } p(x_0, \ldots, x_{n-1}, x) > 0 \textit{ for all } x \leq x_n \\ 0 & \textit{otherwise} \end{cases}$$

is a primitive recursive function.

Proof: For any primitive recursive predicate p, we can define $\hat{p}$ by the recursion equations

$$\begin{aligned} \hat{p}(x_0, \ldots, x_{n-1}, 0) &= \texttt{truish?}(p(x_0, \ldots, x_{n-1}, 0)), \\ \hat{p}(x_0, \ldots, x_{n-1}, t+1) &= \texttt{and}(\hat{p}(x_0, \ldots, x_{n-1}, t), p(x_0, \ldots, x_{n-1}, t+1)), \end{aligned}$$

or, in terms of previously defined functions,

$$\begin{aligned}\hat{p}(x_0,\ldots,x_{n-1},0) \quad = \quad & \texttt{truish?}(p(\texttt{pr}_n^0(x_0,\ldots,x_{n-1}),\\ & \qquad \texttt{pr}_n^1(x_0,\ldots,x_{n-1}),\\ & \qquad\quad \ldots,\\ & \qquad \texttt{pr}_n^{n-1}(x_0,\ldots,x_{n-1}),\\ & \qquad \texttt{zero}())),\end{aligned}$$

$$\begin{aligned}&\hat{p}(x_0,\ldots,x_{n-1},t+1) \quad =\\ &\quad \texttt{and}(\\ &\qquad \texttt{pr}_{n+2}^1(t,\hat{p}(x_0,\ldots,x_{n-1},t),x_0,\ldots,x_{n-1}),\\ &\qquad p(\texttt{pr}_{n+2}^2(t,\hat{p}(x_0,\ldots,x_{n-1},t),x_0,\ldots,x_{n-1}),\\ &\qquad\quad \ldots,\\ &\qquad\quad \texttt{pr}_{n+2}^{n+1}(t,\hat{p}(x_0,\ldots,x_{n-1},t),x_0,\ldots,x_{n-1}),\\ &\qquad\quad \texttt{successor}(\\ &\qquad\qquad \texttt{pr}_{n+2}^0(t,\hat{p}(x_0,\ldots,x_{n-1},t),x_0,\ldots,x_{n-1}))) \qquad .\end{aligned}$$

Hence

$$\begin{aligned}\hat{p} \equiv\ & [\Upsilon_n\ [\circ_n^1\ \texttt{truish?}\ [\circ_n^{n+1}\ p\ \texttt{pr}_n^0\ \ldots\ \texttt{pr}_n^{n-1}\ \texttt{zero}_n]]\\ & \quad [\circ_{n+2}^2\ \texttt{and}\\ & \qquad\quad \texttt{pr}_{n+2}^1\\ & \qquad\quad [\circ_{n+2}^{n+1}\ p\ \texttt{pr}_{n+2}^2\ \cdots\ \texttt{pr}_{n+2}^{n+1}\ [\circ_{n+2}^1\ \texttt{successor}\ \texttt{pr}_{n+2}^0]]]].\end{aligned}$$

■

We use an expression of the form '$[\bar{\forall}_n\ p]$' for the bounded quantification of p. For instance,

$$[\bar{\forall}_0\ \texttt{even?}](117) = 0,$$

since it is not the case that every natural number up to and including 117 is even; but

$$[\bar{\forall}_1\ \texttt{divides?}](1,40) = 1,$$

because 1 does divide every natural number up to and including 40.

Note that the subscript indicates the number of *fixed* parameters and so is one less than the valence of the predicate that is being defined.

Bounded Summation

The *bounded sum* Σf of a singulary function f is a function that takes one input, the upper bound x, and outputs the sum of the values of f for all of the inputs from 0 up to and including x:

$$\Sigma f(x) = f(0) + \cdots + f(x) = \sum_{t=0}^{x} f(t).$$

We can extend this idea to functions of any positive valence: The bounded sum Σf of a function f of valence $n+1$ is defined by

$$\begin{aligned}\Sigma f(x_0,\ldots,x_n) &= f(x_0,\ldots,x_{n-1},0)+\cdots+f(x_0,\ldots,x_{n-1},x_n)\\ &= \sum_{t=0}^{x_n} f(x_0,\ldots,x_{n-1},t).\end{aligned}$$

Theorem 6.2 *The bounded sum Σf of any primitive recursive function f is also a primitive recursive function.*

Proof: We can use the same idea that we used for bounded quantification, but with addition rather than conjunction. The recursion equations are:

$$\begin{aligned}\Sigma f(x_0,\ldots,x_{n-1},0) &= f(x_0,\ldots,x_{n-1},0),\\ \Sigma f(x_0,\ldots,x_{n-1},t+1) &= \Sigma f(x_0,\ldots,x_{n-1},t)+f(x_0,\ldots,x_{n-1},t+1),\end{aligned}$$

or, in terms of previously defined functions,

$$\begin{aligned}\Sigma f(x_0,\ldots,x_{n-1},0) &= f(\mathtt{pr}^0_n(x_0,\ldots,x_{n-1}),\\ &\quad\ \mathtt{pr}^1_n(x_0,\ldots,x_{n-1}),\\ &\quad\ \ldots,\\ &\quad\ \mathtt{pr}^{n-1}_n(x_0,\ldots,x_{n-1}),\\ &\quad\ \mathtt{zero}()),\end{aligned}$$

$$\begin{aligned}&\Sigma f(x_0,\ldots,x_{n-1},t+1) =\\ &\ \mathtt{add}(\\ &\quad \mathtt{pr}^1_{n+2}(t,\Sigma f(x_0,\ldots,x_{n-1},t),x_0,\ldots,x_{n-1}),\\ &\quad f(\mathtt{pr}^2_{n+2}(t,\Sigma f(x_0,\ldots,x_{n-1},t),x_0,\ldots,x_{n-1}),\\ &\qquad \ldots,\\ &\qquad \mathtt{pr}^{n+1}_{n+2}(t,\Sigma f(x_0,\ldots,x_{n-1},t),x_0,\ldots,x_{n-1}),\\ &\qquad \mathtt{successor}(\mathtt{pr}^0_{n+2}(t,\Sigma f(x_0,\ldots,x_{n-1},t),x_0,\ldots,x_{n-1}))).\end{aligned}$$

Hence

$$\begin{aligned}\Sigma f \equiv\ &[\Upsilon_n\ [\circ^{n+1}_n\ f\ \mathtt{pr}^0_n\ \ldots\ \mathtt{pr}^{n-1}_n\ \mathtt{zero}_n]\\ &\quad [\circ^2_{n+2}\ \mathtt{add}\\ &\qquad\quad \mathtt{pr}^1_{n+2}\\ &\qquad\quad [\circ^{n+1}_{n+2}\ f\ \mathtt{pr}^2_{n+2}\ \ldots\ \mathtt{pr}^{n+1}_{n+2}\ [\circ^1_{n+2}\ \mathtt{successor}\ \mathtt{pr}^0_{n+2}]]].\end{aligned}$$

■

We shall write '$[\bar{\Sigma}_n\ \ f]$' for the bounded sum of f. The subscript n is the number of fixed inputs to f (and so the valence of f is $n+1$).

Bounded Minimization

The *minimum* of a singulary function p under bound x, where x is a natural number, is the least natural number i such that $p(i)$ is truish, provided that

there is at least one such number less than or equal to x. If there is no such number, then the minimum is arbitrarily declared to be 0.

Once again, this idea can be generalized to functions of any positive valence: The *bounded minimization* $\bar{\mu}p$ of a function p of valence $n+1$ is a function of valence $n+1$ that, when given the inputs $x_0, \ldots, x_{n-1}, x_n$, yields the least natural number i such that $i \leq x_n$ and $p(x_0, \ldots, x_{n-1}, i)$ is truish, or 0 if there is no such natural number.

Theorem 6.3 *The bounded minimization $\bar{\mu}p$ of any primitive recursive predicate p is a primitive recursive function.*

Proof: We can define $\bar{\mu}p$ thus:

$$\bar{\mu}p(x_0, \ldots, x_n) = \begin{cases} 0 \quad \text{if } \texttt{[}\bar{\forall}_n \ \texttt{[}\circ^1_{n+1} \ \texttt{not} \ p\texttt{]]}(x_0, \ldots, x_n) > 0, \\ \texttt{[}\bar{\Sigma}_n \ \texttt{[}\bar{\forall}_n \ \texttt{[}\circ^1_{n+1} \ \texttt{not} \ p\texttt{]]]}(x_0, \ldots, x_n) \\ \qquad \text{otherwise,} \end{cases}$$

or, more concisely,

$$\bar{\mu}p \equiv \texttt{[¿}_{n+1} \ \texttt{[}\bar{\forall}_n \ \texttt{[}\circ^1_{n+1} \ \texttt{not} \ p\texttt{]]} \ \texttt{zero}_{n+1} \ \texttt{[}\bar{\Sigma}_n \ \texttt{[}\bar{\forall}_n \ \texttt{[}\circ^1_{n+1} \ \texttt{not} \ p\texttt{]]]]}.$$

■

The idea is that $\texttt{[}\bar{\forall}_n \ \texttt{[}\circ^1_{n+1} \ \texttt{not} \ p\texttt{]]}$ tests for the "error" case in which no integer i that is less than or equal to the specified bound satisfies p, $\texttt{zero}_{n+1}$ ensures that the result is 0 in the error case, and in all other cases the bounded summation tallies up the natural numbers that are encountered before the first one that satisfies p is reached (since those are the inputs for which $\texttt{[}\bar{\forall}_n \ \texttt{[}\circ^1_{n+1}$ `not` $p\texttt{]]}$ outputs 1). Since the least natural number that satisfies p is equal to the number of natural numbers that preceded it, this tally is exactly the result we want.

We shall use expressions of the form '$\texttt{[}\bar{\mu}_n \ p\texttt{]}$' for bounded minimization functions derived from predicates p in this way. Once again, the subscript indicates the number of fixed inputs (excluding the bound), so that the valence of $\texttt{[}\bar{\mu}_n \ p\texttt{]}$ is $n+1$.

Exercises

6–1 Using `bounded-sum`, define a singulary function `termial` that outputs the sum of all of the natural numbers less than or equal to its input. Prove that `termial` is primitive recursive.

6–2 The Euler ϕ-function counts the exact divisors of a given positive integer. For instance, 12 has six divisors (1, 2, 3, 4, 6, and 12), so $\phi(12) = 6$. Define a singulary, primitive recursive function `euler-phi` that outputs 0 (arbitrarily) when its input is 0 and $\phi(n)$ when its input is any positive integer n.

6–3 Extend our formal notation to include a bounded exponential quantifier, analogous to the bounded universal quantifier introduced in this section. Prove that, for any primitive recursive function p of positive valence $n + 1$, the predicate $\exists p$ is also primitive recursive, where $\exists p(x_0, \ldots, x_{n-1}, x_n)$ is 1 if $p(x_0, \ldots, x_{n-1}, x) > 0$ for some $x \leq x_n$ and 0 otherwise. (Hint: Existential quantification is definable in terms of universal quantification and negation.)

6–4 A natural number is *prime* if it is greater than or equal to 2 and evenly divisible only by itself and 1. Define a singulary, primitive recursive predicate `prime?` that determines whether its input is prime.

6–5 Describe the function $[\bar{\mu}_1$ `less?`$]$, explaining how its output is related to its inputs.

Chapter 7

Data Structures

Rather surprisingly, we can build data structures even inside the apparently flat set $\mathbb{N}$, using encoding and decoding functions to build them and to extract their components as needed.

Pairs

For instance, consider the binary encoding function `cons`, defined by

$$\texttt{cons}(x_0, x_1) = 2^{x_0} \cdot (2x_1 + 1).$$

This function maps every pair (x_0, x_1) of natural numbers into a different positive integer: Odd numbers encode pairs in which the first term is 0, doubles of odd numbers (2, 6, 10, 14, and so on) encode pairs in which the first term is 1, quadruples of odd numbers encode pairs in which the first term is 2, and so on.

Theorem 7.1 *The function* `cons` *is primitive recursive.*

Proof:

$$\begin{aligned}
\texttt{two} &\equiv [\circ_0^1\ \texttt{successor one}],\\
\texttt{cons} &\equiv [\circ_2^2\ \texttt{multiply}\\
&\qquad [\circ_2^2\ \texttt{raise-to-power two}_2\ \texttt{pr}_2^0]\\
&\qquad [\circ_2^1\ \texttt{successor}\ [\circ_2^1\ \texttt{double pr}_2^1]]].
\end{aligned}$$

■

We can also decode a pair to get its components, using two functions—`car` to recover the first component, `cdr` to recover the second one.

Theorem 7.2 *The functions* `car` *and* `cdr` *are primitive recursive.*

The car is the least natural number y such that 2^{y+1} fails to divide the pair. Since this natural number is obviously less than the one that encodes the

entire pair, we can use bounded minimization to recover it. First, let's define the predicate to which minimization will be applied—a binary predicate `d` such that $\mathtt{d}(x, y)$ is true just in case 2^{y+1} does not divide x:

$$\begin{aligned}\mathtt{d} \equiv [\circ_2^1\ \mathtt{not}\ [\circ_2^2\ \mathtt{divides?}\\ [\circ_2^2\ \mathtt{raise\text{-}to\text{-}power}\ \mathtt{two}_2\ [\circ_2^1\ \mathtt{successor}\ \mathtt{pr}_2^1]]\\ \mathtt{pr}_2^0]]\end{aligned}$$

The selector function `car`, then, is the bounded minimum of `d`, with the encoded pair being supplied as each input—as dividend in the first input position, and as upper bound for the predecessor of the divisor's exponent in the second input position:

$$\mathtt{car} \equiv [\circ_1^2\ [\bar{\mu}_1\ \mathtt{d}]\ \mathtt{pr}_1^0\ \mathtt{pr}_1^0].$$

To recover the second term of a given pair, we can raise two to the power of the car, divide the number that encodes the pair by the resulting power to get the odd factor, subtract one from that factor, and divide the result by two:

$$\begin{aligned}\mathtt{cdr} \equiv [\circ_1^2\ \mathtt{quotient}\\ [\circ_1^1\ \mathtt{predecessor}\\ [\circ_1^2\ \mathtt{quotient}\ \mathtt{pr}_1^0\ [\circ_1^2\ \mathtt{raise\text{-}to\text{-}power}\ \mathtt{two}_1\ \mathtt{car}]]]\\ \mathtt{two}_1].\end{aligned}$$

■

Since $\mathtt{cons}(x_0, x_1) \neq 0$ for any $x_0, x_1 \in \mathbb{N}$, the effect of applying `car` and `cdr` to 0 is somewhat arbitrary. If you thread through the definitions, it turns out that $\mathtt{car}(0) = 0$ and $\mathtt{cdr}(0) = 0$; nevertheless, $\mathtt{cons}(0, 0)$ is 1, not 0. (Consider the 0 that `car` and `cdr` return in this case as an error indicator.)

Lists

We could have explored other encodings in which ordered pairs of natural numbers are mapped to natural numbers with no leftovers, and in some ways that would be a more elegant system. However, there is a rationale for setting 0 aside as a non-pair, which experienced functional programmers have undoubtedly already anticipated: We need a spare encoding for the empty list. To formalize this convention, we adopt the name `nil` for a nullary function that yields the empty list (encoded as the natural number 0), and `null?` for a singulary function that determines whether its input is the empty list. Naturally, both of these functions are primitive recursive:

$$\begin{aligned}\mathtt{nil} &\equiv \mathtt{zero},\\ \mathtt{null?} &\equiv \mathtt{zero?}.\end{aligned}$$

Now we can provide an encoding that maps *every* finite sequence of natural numbers, of whatever length, to a different natural number, and a decoding that

maps *every* natural number to a different finite sequence of natural numbers:

$$\begin{aligned}
0 = \texttt{nil}() &\mapsto () \\
1 = \texttt{cons}(0,0) = \texttt{cons}(0,\texttt{nil}()) &\mapsto (0) \\
2 = \texttt{cons}(1,0) = \texttt{cons}(1,\texttt{nil}()) &\mapsto (1) \\
3 = \texttt{cons}(0,1) = \texttt{cons}(0,\texttt{cons}(0,\texttt{nil}())) &\mapsto (0,0) \\
4 = \texttt{cons}(2,0) = \texttt{cons}(2,\texttt{nil}()) &\mapsto (2) \\
5 = \texttt{cons}(0,2) = \texttt{cons}(0,\texttt{cons}(1,\texttt{nil}())) &\mapsto (0,1) \\
6 = \texttt{cons}(1,1) = \texttt{cons}(1,\texttt{cons}(0,\texttt{nil}())) &\mapsto (1,0) \\
7 = \texttt{cons}(0,3) = \texttt{cons}(0,\texttt{cons}(0,\texttt{cons}(0,\texttt{nil}()))) &\mapsto (0,0,0) \\
8 = \texttt{cons}(3,0) = \texttt{cons}(3,\texttt{nil}()) &\mapsto (3) \\
9 = \texttt{cons}(0,4) = \texttt{cons}(0,\texttt{cons}(2,\texttt{nil}())) &\mapsto (0,2) \\
10 = \texttt{cons}(1,2) = \texttt{cons}(1,\texttt{cons}(1,\texttt{nil}())) &\mapsto (1,1) \\
11 = \texttt{cons}(0,5) = \texttt{cons}(0,\texttt{cons}(0,\texttt{cons}(1,\texttt{nil}()))) &\mapsto (0,0,1) \\
12 = \texttt{cons}(2,1) = \texttt{cons}(2,\texttt{cons}(0,\texttt{nil}())) &\mapsto (2,0), \text{ etc.}
\end{aligned}$$

The function `nth-cdr` applies the `cdr` function a specified number of times to a given list (encoded as a natural number) and outputs the result.

Theorem 7.3 *The function* `nth-cdr` *is primitive recursive.*

Proof: Its recursion equations are

$$\begin{aligned}
\texttt{nth-cdr}(x,0) &= x \\
&= \texttt{pr}_1^0(x), \\
\texttt{nth-cdr}(x,t+1) &= \texttt{cdr}(\texttt{nth-cdr}(x,t)) \\
&= \texttt{cdr}(\texttt{pr}_3^1(t,\texttt{nth-cdr}(x,t),x)),
\end{aligned}$$

so we can define it thus:

$$\texttt{nth-cdr} \equiv [\Upsilon_1\ \texttt{pr}_1^0\ [\circ_3^1\ \texttt{cdr}\ \texttt{pr}_3^1]].$$

■

If we "run off the end" of the list by providing an index (second input) greater than or equal to the length of the list, `nth-cdr` simply yields the usual error value, 0.

In fact, we can *define* the length of the list encoded by x as the least t such that $\texttt{nth-cdr}(x,t)$ is 0.

Theorem 7.4 *The* `length` *function is primitive recursive.*

Proof: The approach is to use bounded minimization. The number x that encodes the list can itself be the upper bound for the search, since the length of a list is always less than or equal to its encoding.

$$\texttt{length} \equiv [\circ_1^2\ [\bar{\mu}_1\ [\circ_2^1\ \texttt{null?}\ \texttt{nth-cdr}]]\ \texttt{pr}_1^0\ \texttt{pr}_1^0].$$

■

The indexing selector for lists is `list-ref`.

Theorem 7.5 *The function* `list-ref` *is primitive recursive.*

Proof: Just take the `car` of the `nth-cdr`:

$$\texttt{list-ref} \equiv [\circ_2^1\ \texttt{car nth-cdr}].$$

■

Iteration

The pattern by which we derived `nth-cdr` from `cdr` comes up often enough that it will be helpful to have a name for it. For any singulary function f, we can define a binary iterate $\hat{f}$ by means of the recursion equations

$$\begin{aligned} \hat{f}(x,0) &= x, \\ \hat{f}(x,t+1) &= f(\hat{f}(x,t)), \end{aligned}$$

which lead to the definition

$$\hat{f} \equiv [\Upsilon_1\ \texttt{pr}_1^0\ [\circ_3^1\ f\ \texttt{pr}_3^1]]$$

We shall use the notation '$[\circledcirc_0\ f]$' to express the iterate of f defined in this way. Thus the `nth-cdr` function is $[\circledcirc_0$ `cdr`$]$, and `add` is $[\circledcirc_0$ `successor`$]$.

In fact, we can generalize the pattern still further, to convert any function f of valence $n+1$ into an iterate $\hat{f}$ of valence $n+2$, which we'll express as '$[\circledcirc_n f]$':

$$\begin{aligned} \hat{f}(x_0,\ldots,x_n,0) &= x_n, \\ \hat{f}(x_0,\ldots,x_n,t+1) &= f(x_0,\ldots,x_{n-1},\hat{f}(x_0,\ldots,x_n,t)), \end{aligned}$$

Theorem 7.6 *The iterate* $\hat{f}$ *of any primitive recursive function* f *is primitive recursive.*

Proof:

$$[\circledcirc_n\ f] \equiv [\Upsilon_{n+1}\ \texttt{pr}_{n+1}^n\ [\circ_{n+3}^{n+1}\ f\ \texttt{pr}_{n+3}^2\ \ldots\ \texttt{pr}_{n+3}^{n+1}\ \texttt{pr}_{n+3}^1]].$$

■

Strings

As another example of encoding, we can extend the notion of primitive recursive functions to functions that take strings on some finite alphabet as inputs and output such strings as values. The idea is to associate each symbol in the alphabet with a positive integer that acts as its serial number and then to treat strings as numerals in an appropriate "positional" system of numeration. The encoding of a string will then be the number that it denotes in such a system.

Suppose that the number of symbols in the alphabet is n. We'll let the serial numbers for the symbols, then, be $1, 2, 3, \ldots, n$. We can define the encoding for a given string recursively:

- The encoding for the null string, ε, is 0.
- For any string x and any symbol a from the alphabet, the encoding of xa is the sum of a's serial number and n times the encoding of x.

So, for instance, if the alphabet is $\{\mathtt{a}, \mathtt{b}\}$, let's give `a` the serial number 1 and `b` the serial number 2. Then, since in this case $n = 2$, the encoding for `abbab` would be

$$2 \cdot (2 \cdot (2 \cdot (2 \cdot 0 + 1) + 2) + 2) + 1) + 2,$$

or 44.

This technique has the convenient feature that ascending numerical order among the encodings of strings corresponds exactly to a fairly natural order among the strings: shorter strings precede longer ones, and strings of equal length are ordered lexicographically.

Since the particular encoding used for strings depends on the size n of the alphabet, our string-related functions take n as an additional input throughout. By convention, we shall put this size-of-alphabet input first.

Truncation

The `truncate` function strips away the rightmost symbol of any non-empty string. More precisely: given the encoding $\bar{s}$ of a non-empty string s and the number of symbols in the alphabet used, `truncate` outputs the encoded result of removing the rightmost symbol. (Thus the inputs to `truncate`, as well as the value it outputs, are natural numbers.)

Theorem 7.7 *Truncation is a primitive recursive function.*

Proof:

$$\mathtt{truncate}(n, \bar{s}) = \mathtt{quotient}(\mathtt{predecessor}(\bar{s}), n),$$

so that

$$\mathtt{truncate} \equiv [\circ_2^2\ \mathtt{quotient}\ [\circ_2^1\ \mathtt{predecessor}\ \mathtt{pr}_2^1]\ \mathtt{pr}_2^0].$$

■

String Length

Theorem 7.8 *The function* `string-length` *is primitive recursive.*

Proof: We can compute the length of a string as the number of times it can be truncated before becoming null.

$$\texttt{string-length} \equiv [\circ_2^3\ [\bar{\mu}_2\ [\circ_3^1\ \texttt{zero?}\ [\circledcirc_1\ \texttt{truncate}]]]\ \texttt{pr}_2^0\ \texttt{pr}_2^1\ \texttt{pr}_2^1].$$

■

Note that `string-length` takes two inputs: the alphabet size and the encoded string. The encoding for the string itself serves as the upper bound on the number of truncation steps that will be required (which is why the second input is projected out twice, to serve as both the second and third inputs to the bounded minimization). This definition therefore presupposes a property of the encoding: The length of any string must be less than or equal to its encoding.

String Indexing

We will need a `string-ref` function that extracts the encoding for a symbol at a given (zero-based) position in a string from the encoding for the string itself.

Theorem 7.9 *The function* `string-ref` *is primitive recursive.*

Proof: We truncate the string until the specified position is at the right end, then use `remainder` (or, actually, `redniamer`) to inspect the rightmost symbol (correcting a zero remainder to n). First, let's define the function that extracts the rightmost symbol of any non-null string:

$$\texttt{last-symbol}(n, \bar{s}) = \begin{cases} n & \text{if } \texttt{divides?}(n, \bar{s}), \\ \texttt{redniamer}(n, \bar{s}) & \text{otherwise,} \end{cases}$$

so that

$$\texttt{last-symbol} \equiv [\textit{¿}_2\ \texttt{divides?}\ \texttt{pr}_2^0\ \texttt{redniamer}]$$

Here, then, is the string-indexing function `string-ref`. The first input is the alphabet size, the second is the string, and the third is the zero-based position in the string.

$$\begin{aligned} &\texttt{string-ref}(n, \bar{s}, p) \\ &\quad = \ \texttt{last-symbol}(n, \\ &\qquad [\circledcirc_1\ \texttt{truncate}](n, \bar{s}, \\ &\qquad\quad \texttt{subtract}(\texttt{string-length}(n, \bar{s}), \texttt{successor}(p)))), \end{aligned}$$

so that

$$\begin{aligned}
\texttt{string-ref} \equiv [\circ_3^2\ & \texttt{last-symbol} \\
& \texttt{pr}_3^0 \\
& [\circ_3^3\ [\circledcirc_1\ \texttt{truncate}] \\
& \quad \texttt{pr}_3^0 \\
& \quad \texttt{pr}_3^1 \\
& \quad [\circ_3^2\ \texttt{subtract} \\
& \quad\quad [\circ_3^2\ \texttt{string-length}\ \texttt{pr}_3^0\ \texttt{pr}_3^1] \\
& \quad\quad [\circ_3^1\ \texttt{successor}\ \texttt{pr}_3^2]]]].
\end{aligned}$$

■

Concatenation

Theorem 7.10 *String concatenation is a primitive recursive function.*

Proof: To concatenate two strings, we can "scale up" the first one by a power of the size of the alphabet and simply add the second one to the result. The exponent on the scale factor is the length of the second string.

$$\begin{aligned}
\texttt{concatenate}(n, \bar{s}_0, \bar{s}_1) \quad &= \quad \bar{s}_0 \cdot n^m + \bar{s}_1, \\
&\qquad \text{where } m = \texttt{string-length}(n, \bar{s}_1).
\end{aligned}$$

Hence

$$\begin{aligned}
\texttt{concatenate} \equiv [\circ_3^2\ & \texttt{add} \\
& [\circ_3^2\ \texttt{multiply} \\
& \quad \texttt{pr}_3^1 \\
& \quad [\circ_3^2\ \texttt{raise-to-power} \\
& \quad\quad \texttt{pr}_3^0 \\
& \quad\quad [\circ_3^2\ \texttt{string-length}\ \texttt{pr}_3^0\ \texttt{pr}_3^2]]] \\
& \texttt{pr}_3^2].
\end{aligned}$$

■

Since the encoding for a single symbol is equal to the encoding for the string containing just that symbol, either or both of the last two inputs to `concatenate` can be understood in either way.

Selecting Substrings

Instead of a single symbol, we can select any substring of a given string by providing the position at which the substring begins and the position just before which it ends.

Theorem 7.11 *The function* `substring` *is primitive recursive.*

Proof: We use recursion equations to define the `ss` function, which also extracts substrings, but has a slightly different interface. Its first input is the size of the alphabet, as usual; its second is the encoded string, $\bar{s}$; its third is the zero-based position at which the substring begins; but its fourth input is the *length* of the substring to be extracted.

Selecting any zero-length substring yields ε (in its encoded form, 0), and selecting a substring of length $t+1$ yields the result of concatenating the corresponding substring of length t with the symbol at the next following position. If there is no such symbol, we concatenate nothing and simply output the shorter substring. So:

$$\begin{aligned}
\texttt{ss}(n, \bar{s}, p, 0) &= 0, \\
\texttt{ss}(n, \bar{s}, p, t+1) &= \begin{cases} \texttt{concatenate}(n, \texttt{ss}(n, \bar{s}, p, t), \\ \qquad\qquad \texttt{string-ref}(n, \bar{s}, p+t)) \\ \qquad \text{if } p + t < \texttt{string-length}(n, \bar{s}), \\ \texttt{ss}(n, \bar{s}, p, t) \quad \text{otherwise.} \end{cases}
\end{aligned}$$

The definition of `ss` as a primitive recursive function, then, is

$$\begin{array}{l}
\texttt{ss} \equiv [\Upsilon_3\ \texttt{zero}_3 \\
\quad [\textit{¿}_5\ [\circ_5^2\ \texttt{less-than} \\
\qquad [\circ_5^2\ \texttt{add}\ \texttt{pr}_5^4\ \texttt{pr}_5^0] \\
\qquad [\circ_5^2\ \texttt{string-length}\ \texttt{pr}_5^2\ \texttt{pr}_5^3]] \\
\qquad [\circ_5^3\ \texttt{concatenate} \\
\qquad\quad \texttt{pr}_5^2 \\
\qquad\quad \texttt{pr}_5^1 \\
\qquad\quad [\circ_5^3\ \texttt{string-ref}\ \texttt{pr}_5^2\ \texttt{pr}_5^3\ [\circ_5^2\ \texttt{add}\ \texttt{pr}_5^4\ \texttt{pr}_5^0]]] \\
\qquad \texttt{pr}_5^1]]
\end{array}$$

The more Scheme-like `substring` function, taking the position just before which the substring ends as its fourth input instead of the length of the substring, can be defined in terms of `ss`:

$$\texttt{substring}(n, \bar{s}, p, e) = \texttt{ss}(n, \bar{s}, p, e \dot{-} p),$$

so that

$$\texttt{substring} \equiv [\circ_4^4\ \texttt{ss}\ \texttt{pr}_4^0\ \texttt{pr}_4^1\ \texttt{pr}_4^2\ [\circ_4^2\ \texttt{subtract}\ \texttt{pr}_4^3\ \texttt{pr}_4^2]].$$

■

String Update

The `string-update` function inserts a new symbol into a given string at a given position, replacing the symbol that previously occupied that position.

Theorem 7.12 *The function* `string-update` *is primitive recursive.*

Proof: We concatenate the part of the string that precedes the given position, the new symbol, and the part of the string that follows the given position.

$$\begin{aligned}\texttt{string-update}(n, \bar{s}, p, \bar{a}) \;=\; & \texttt{concatenate}(n,\\ & \quad \texttt{substring}(n, \bar{s}, 0, p),\\ & \quad \texttt{concatenate}(n, \bar{a},\\ & \quad\quad \texttt{substring}(n, \bar{s}, p+1,\\ & \quad\quad\quad \texttt{string-length}(n, \bar{s})))),\end{aligned}$$

so that

$$\begin{aligned}\texttt{string-update} \equiv\ & [\circ_4^3\ \texttt{concatenate}\\ & \quad \texttt{pr}_4^0\\ & \quad [\circ_4^4\ \texttt{substring}\ \texttt{pr}_4^0\ \texttt{pr}_4^1\ \texttt{zero}_4\ \texttt{pr}_4^2]\\ & \quad [\circ_4^3\ \texttt{concatenate}\\ & \quad\quad \texttt{pr}_4^0\\ & \quad\quad \texttt{pr}_4^3\\ & \quad\quad [\circ_4^4\ \texttt{substring}\\ & \quad\quad\quad \texttt{pr}_4^0\\ & \quad\quad\quad \texttt{pr}_4^1\\ & \quad\quad\quad [\circ_4^1\ \texttt{successor}\ \texttt{pr}_4^2]\\ & \quad\quad\quad [\circ_4^2\ \texttt{string-length}\ \texttt{pr}_4^0\ \texttt{pr}_4^1]]]]\end{aligned}$$

■

Since `substring` outputs an empty string if the specified starting position is greater than or equal to the specified ending position, the `string-update` function can also be used to add a symbol at the end of a string. One need only specify an update position that is greater than or equal to the length of s.

Exercises

7–1 Define a singulary, primitive recursive function `exch` that takes the encoding for a pair (x_0, x_1) as its input and outputs the encoding for a similar pair (x_1, x_0) with car and cdr reversed. (Since 0 is not the encoding for any pair, you may arrange for `exch`(0) to be any value that happens to be convenient.)

7–2 In your favorite programming language, write and test procedures that implement the encoding and decoding methods for lists of natural numbers.

7–3 What happens if the second input to `list-ref` is greater than or equal to the length of the list that the first input encodes?

7–4 Describe the function $[\circledcirc_0\ \texttt{double}]$, explaining how its output is related to its input.

7–5 The `reverse-list` function inputs the encoding for a list and outputs the encoding for a list with the same elements, arranged in the opposite order. Prove that `reverse-list` is primitive recursive.

7–6 Using iteration, define a ternary, primitive recursive function `repl` that takes the size of an alphabet, the encoding for a string on that alphabet, and a natural number used as a repetition factor as its inputs and outputs the encoding for a string consisting of that many repetitions of the given string. (For instance, since the encoding for the string `abbac` on the alphabet $\{\texttt{a}, \texttt{b}, \texttt{c}\}$ is 159, $\texttt{repl}(3, 159, 4)$ is 2290903800, which encodes `abbacabbacabbacabbac`.)

Chapter 8

Course-of-Values Recursion

Sometimes it is most convenient to define a function f by means of a recursion equation in which the value of $f(x_0, \ldots, x_{n-1}, t+1)$ depends not only on the immediately preceding value $f(x_0, \ldots, x_{n-1}, t)$, but on any or all of the preceding values $f(x_0, \ldots, x_{n-1}, t), f(x_0, \ldots, x_{n-1}, t-1), \ldots, f(x_0, \ldots, x_{n-1}, 0)$. A definition of this kind is called a *course-of-values recursion*.

For instance, the *Catalan sequence* is a sequence $C_0, C_1, \ldots$ of natural numbers specified by the equations

$$\begin{aligned} C_0 &= 1, \\ C_{n+1} &= \sum_{k=0}^{n} C_k C_{n-k}. \end{aligned}$$

Let's compute the first few values of a function `catalan` that inputs a natural number and outputs the corresponding term of the sequence, just to get a feel for how the preceding values are used:

$$\begin{aligned} \texttt{catalan}(0) &= 1, \\ \texttt{catalan}(1) &= \texttt{catalan}(0) \cdot \texttt{catalan}(0) \\ &= 1 \cdot 1 \\ &= 1, \\ \texttt{catalan}(2) &= \texttt{catalan}(0) \cdot \texttt{catalan}(1) + \\ &\qquad \texttt{catalan}(1) \cdot \texttt{catalan}(0) \\ &= 1 \cdot 1 + 1 \cdot 1 \\ &= 1 + 1 \\ &= 2, \\ \texttt{catalan}(3) &= \texttt{catalan}(0) \cdot \texttt{catalan}(2) + \\ &\qquad \texttt{catalan}(1) \cdot \texttt{catalan}(1) + \\ &\qquad \texttt{catalan}(2) \cdot \texttt{catalan}(0) \end{aligned}$$

$$\begin{aligned}
&= 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 \\
&= 2 + 1 + 2 \\
&= 5, \\
\texttt{catalan}(4) &= \texttt{catalan}(0) \cdot \texttt{catalan}(3) + \\
&\qquad \texttt{catalan}(1) \cdot \texttt{catalan}(2) + \\
&\qquad \texttt{catalan}(2) \cdot \texttt{catalan}(1) + \\
&\qquad \texttt{catalan}(3) \cdot \texttt{catalan}(0) \\
&= 1 \cdot 5 + 1 \cdot 2 + 2 \cdot 1 + 5 \cdot 1 \\
&= 5 + 2 + 2 + 5 \\
&= 14, \\
\texttt{catalan}(5) &= \texttt{catalan}(0) \cdot \texttt{catalan}(4) + \\
&\qquad \texttt{catalan}(1) \cdot \texttt{catalan}(3) + \\
&\qquad \texttt{catalan}(2) \cdot \texttt{catalan}(2) + \\
&\qquad \texttt{catalan}(3) \cdot \texttt{catalan}(1) + \\
&\qquad \texttt{catalan}(4) \cdot \texttt{catalan}(0) \\
&= 1 \cdot 14 + 1 \cdot 5 + 2 \cdot 2 + 5 \cdot 1 + 14 \cdot 1 \\
&= 14 + 5 + 4 + 5 + 14 \\
&= 42.
\end{aligned}$$

Such course-of-values recursions don't follow the function-building pattern that Υ_n encapsulates, since in that pattern only the immediately preceding value of the function being defined can be recovered from the inputs to the function that helps define it. Are functions like `catalan` nevertheless primitive recursive? If so, how can we use composition and recursion to construct them?

Suppose that we're trying to define a function f of positive valence $n+1$ by course-of-values recursion. What would we need in order to be in a position to define f directly, without relying on the Υ_n operation? We'd have to receive all the preceding values of f as *inputs* somehow, so that the computation of the new value could use them. In fact, just one additional input would be enough, if that input were a list of all of the preceding values of f. We could use `list-ref` to select any values that we needed from that list.

If we knew that f itself was primitive recursive, we could define a function $\tilde{f}$ that takes the same inputs as f, but outputs the encoding for a list of *all* the "preceding values" of f (arranged by the value of the last input to f, in descending order). The following recursion equations specify $\tilde{f}$ in terms of f:

$$\begin{aligned}
\tilde{f}(x_0, \ldots, x_{n-1}, 0) &= \texttt{nil}(), \\
\tilde{f}(x_0, \ldots, x_{n-1}, t+1) &= \texttt{cons}(f(x_0, \ldots, x_{n-1}, t), \tilde{f}(x_0, \ldots, x_{n-1}, t)).
\end{aligned}$$

If we could somehow show independently that $\tilde{f}$ is primitive recursive, we could use it to define f. After all, the whole problem is that f needs access to some or all of its own preceding values, which is exactly what $\tilde{f}$ provides. We

can imagine the helper function f_h, just like f except that it has one additional input, the list of preceding values, as $\tilde{f}$ constructs it:

$$f(x_0, \ldots, x_{n-1}, t) = f_h(x_0, \ldots, x_{n-1}, t, \tilde{f}(x_0, \ldots, x_{n-1}, t)).$$

This equation would be a recipe for definition f as a composition of f_h and $\tilde{f}$ if only we knew that $\tilde{f}$ was primitive recursive.

The course-of-values recursion equations for a function like `catalan` can tell us how to define f_h as a primitive recursive function, but that doesn't resolve the basic circularity. Even if f_h is primitive recursive, we still seem to need f in the definition of $\tilde{f}$ and $\tilde{f}$ in the definition of f.

Just to illustrate our predicament, let's see what the definitions of f_h (which we'll call `catalan-helper`), $\tilde{f}$ (`catalan-list`), and f itself (`catalan`) would look like in our notation.

First, let's get clear on what the `catalan-helper` function is supposed to do. When its inputs are n and the encoding of the list containing C_{n-1}, C_{n-2}, $\ldots$, C_1, and C_0, `catalan-helper` should output C_n. Apart from the special case in which $n = 0$, for which the output 1 is specified by the first of the course-of-values recursion equations, we'll compute the output as a bounded sum of products of values extracted from the list.

To extract C_k from the list (where $k < n$), we apply `list-ref` to the encoding for the list and the correct zero-based position, which works out to be position $n - 1 - k$. (If this seems backwards, recall that the preceding values in the sequence are listed in descending order, so that C_{n-1} is in position 0, at the beginning of the list, and C_0 is at the end, in position $n - 1$.)

To compute each product $C_k C_{n-k}$, we'll need n as well as the encoding for the list and the summation variable k. The function `catalan-core` receives these three quantities as inputs and outputs the desired product:

```
catalan-core ≡ [∘₃² multiply
                [∘₃² list-ref pr₃¹
                              [∘₃² subtract [∘₃¹ predecessor pr₃⁰]
                                            pr₃²]]
                [∘₃² list-ref pr₃¹ [∘₃¹ predecessor pr₃²]]]
```

Then `catalan-helper` computes a bounded sum of values of `catalan-core`, providing three inputs to the summation function: n, the encoding of the list of previous values, and (as the inclusive upper bound for the summation) the predecessor of n:

```
catalan-helper ≡ [¿₂ [∘₂¹ zero? pr₂⁰]
                     one₂
                     [∘₂³ [Σ̄₂ catalan-core] pr₂⁰
                                            pr₂¹
                                            [∘₂¹ predecessor pr₂⁰]]]
```

The `catalan-list` function is a list construction, using `catalan` to generate each element:

$$\texttt{catalan-list} \text{ “}\equiv\text{”} [\Upsilon_0\ \texttt{nil}\ [\circ_2^2\ \texttt{cons}\ [\circ_2^1\ \texttt{catalan}\ \texttt{pr}_2^0]\ \texttt{pr}_2^1]]$$

and `catalan` itself is a composition of `catalan-helper` and `catalan-list`:

$$\texttt{catalan} \text{ “}\equiv\text{”} [\circ_1^2\ \texttt{catalan-helper}\ \texttt{pr}_1^0\ \texttt{catalan-list}]$$

But these last two aren't really definitions, because they would be circular: `catalan-list` depends on `catalan`, and `catalan` depends on `catalan-list`. (I've placed the $\equiv$ symbol in sarcastic quotation marks in these non-definitions as a reminder that it would be pointless to rely on them.) If either of these functions is primitive recursive, so is the other, but so far we haven't proven that *either* of them is primitive recursive.

Note, however, that the definitions of `catalan-core` and `catalan-helper` do not share this circularity and are, in fact, completely sound. Those constructions prove that `catalan-core` and, therefore, `catalan-helper` *are* primitive recursive functions.

The key to resolving the mutual dependence of `catalan` and `catalan-list` is to use pairs to define them *jointly*. Instead of regarding them as two functions, one that outputs natural numbers and another that outputs list encodings, let's merge them into a single function, `catalan-pair`, that outputs encodings for pairs. On input n, `catalan-pair` will output the encoding for a pair that has the natural number that we would like `catalan` to output as its car and the list encoding that we would like `catalan-list` to output as its cdr.

As it turns out, it's surprisingly easy to define `catalan-pair` as a direct simple recursion. We can start from the recursion equations:

$$\begin{aligned} \texttt{catalan-pair}(0) &= \texttt{cons}(\texttt{catalan-helper}(0, \texttt{nil}), \texttt{nil}), \\ \texttt{catalan-pair}(t+1) &= \texttt{cons}(\texttt{catalan-helper}(t+1, \texttt{catalan-pair}(t)), \\ &\qquad \texttt{catalan-pair}(t)). \end{aligned}$$

The second of these equations relies on a sort of pun based on the dual role of `cons`, as a function that couples two values to form a pair and as a function that prepends a value to a list to form a list. We can think of each output from `catalan-pair` either as a pair of two values, `catalan`(n) and `catalan-list`(n), or as a list of values comprising `catalan`(n), `catalan`$(n-1)$, ..., `catalan`(0). The `catalan-pair` function in effect prepends a new element to this list at each step.

In our notation, then, we can define `catalan-pair` thus:

$$\begin{aligned} \texttt{catalan-pair} \equiv [\Upsilon_0\ [\circ_0^2\ \texttt{cons}\ [\circ_0^2\ \texttt{catalan-helper zero nil}]\ \texttt{nil}] \\ [\circ_2^2\ \texttt{cons}\ [\circ_2^2\ \texttt{catalan-helper}\ [\circ_2^1\ \texttt{successor}\ \texttt{pr}_2^0] \\ \texttt{pr}_2^1] \\ \texttt{pr}_2^1]] \end{aligned}$$

The definitions of `catalan` and (incidentally at this point) `catalan-list` follow at once:

$$\begin{aligned} \texttt{catalan} &\equiv [\circ_1^1\ \texttt{car catalan-pair}] \\ \texttt{catalan-list} &\equiv [\circ_1^1\ \texttt{cdr catalan-pair}] \end{aligned}$$

Now we can line the definitions up in sequence—`catalan-core`, `catalan-helper`, `catalan-pair`, `catalan`—so that each relies only on the definitions that precede it. We have therefore proven, by construction,

Theorem 8.1 *The function* `catalan` *is primitive recursive.*

More generally, we begin with a helper function f_h that expresses the way in which f depends on the previous values that $\tilde{f}$ accumulates. Then we use f_h and ordinary recursion to define a function $\ddot{f}$ that computes f and $\tilde{f}$ in parallel, in the following sense: Given any inputs, $\ddot{f}$ returns a pair in which the car is the result of applying f to those inputs and the cdr is the result of applying $\tilde{f}$ to them. In other words, the goal is to define $\ddot{f}$ by direct recursion in such a way that

$$\ddot{f}(x_0, \ldots, x_n) = \texttt{cons}(f(x_0, \ldots, x_n), \tilde{f}(x_0, \ldots, x_n)).$$

We will now show that, whenever we have a helper function f_h that relates f and $\tilde{f}$ in the way described above, we can define $\ddot{f}$ in terms of f_h, and $\ddot{f}$ will be primitive recursive whenever f_h is.

First, let's set up the general recursion equations for $\ddot{f}$, making the role of f_h clear.

$$\begin{aligned}
\ddot{f}(x_0, \ldots, x_{n-1}, 0) &= \texttt{cons}(f(x_0, \ldots, x_{n-1}, 0), \tilde{f}(x_0, \ldots, x_{n-1}, 0)) \\
&= \texttt{cons}(f_h(x_0, \ldots, x_{n-1}, 0, 0), 0) \\
&= \texttt{cons}(f_h(x_0, \ldots, x_{n-1}, 0, \texttt{nil}_n(x_0, \ldots, x_{n-1})), \\
&\quad \texttt{nil}_n(x_0, \ldots, x_{n-1})), \\
\ddot{f}(x_0, \ldots, x_{n-1}, t+1) &= \texttt{cons}(f(x_0, \ldots, x_{n-1}, t+1), \\
&\qquad \tilde{f}(x_0, \ldots, x_{n-1}, t+1)) \\
&= \texttt{cons}(f_h(x_0, \ldots, x_{n-1}, t+1, \\
&\qquad \texttt{cons}(f(x_0, \ldots, x_{n-1}, t), \tilde{f}(x_0, \ldots, x_{n-1}, t))), \\
&\quad \texttt{cons}(f(x_0, \ldots, x_{n-1}, t), \tilde{f}(x_0, \ldots, x_{n-1}, t))) \\
&= \texttt{cons}(f_h(x_0, \ldots, x_{n-1}, t+1, \ddot{f}(x_0, \ldots, x_{n-1}, t)), \\
&\quad \ddot{f}(x_0, \ldots, x_{n-1}, t)).
\end{aligned}$$

So, in our notation, we can define $\ddot{f}$ by direct recursion, using f_h to generate each new value of the underlying function f as needed:

$$\begin{array}{l}\ddot{f} \equiv [\Upsilon_n\ [\circ^2_n\ \texttt{cons} \\ \qquad [\circ^{n+2}_n\ f_h\ \texttt{pr}^0_n\ \ldots\ \texttt{pr}^{n-1}_n\ \texttt{zero}_n\ \texttt{nil}_n] \\ \qquad \texttt{nil}_n] \\ \quad [\circ^2_{n+2}\ \texttt{cons} \\ \qquad [\circ^{n+2}_{n+2}\ f_h \\ \qquad\quad \texttt{pr}^2_{n+2} \\ \qquad\quad \vdots \\ \qquad\quad \texttt{pr}^{n+1}_{n+2} \\ \qquad\quad [\circ^1_{n+2}\ \texttt{successor}\ \texttt{pr}^0_{n+2}] \\ \qquad\quad \texttt{pr}^1_{n+2}] \\ \qquad \texttt{pr}^1_{n+2}]].\end{array}$$

We can then define the desired function f from $\ddot{f}$:

$$f \equiv [\circ^1_{n+1}\ \texttt{car}\ \ddot{f}].$$

When a function f of valence $n+1$ is derived in this way from another function f_h (by way of $\ddot{f}$), we'll write it as '$[\tilde{\Upsilon}_n\ f_h]$'.

Theorem 8.2 *For any primitive recursive function f_h, $[\tilde{\Upsilon}_n\ f_h]$ is a primitive recursive function.*

Proof: We simply combine the constructions already given:

$$\begin{array}{l}[\tilde{\Upsilon}_n\ f_h] \equiv [\circ^1_{n+1}\ \texttt{car}\ [\Upsilon_n\ [\circ^2_n\ \texttt{cons} \\ \qquad [\circ^{n+2}_n\ f_h\ \texttt{pr}^0_n\ \ldots\ \texttt{pr}^{n-1}_n\ \texttt{zero}_n\ \texttt{nil}_n] \\ \qquad \texttt{nil}_n] \\ \quad [\circ^2_{n+2}\ \texttt{cons} \\ \qquad [\circ^{n+2}_{n+2}\ f_h \\ \qquad\quad \texttt{pr}^2_{n+2} \\ \qquad\quad\ \vdots \\ \qquad\quad \texttt{pr}^{n+1}_{n+2} \\ \qquad\quad [\circ^1_{n+2}\ \texttt{successor}\ \texttt{pr}^0_{n+2}] \\ \qquad\quad \texttt{pr}^1_{n+2}] \\ \qquad \texttt{pr}^1_{n+2}]]].\end{array}$$

■

As another example of how to use course-of-values recursion, let us define a function `decrement-last` that takes as its input the encoding of a list of natural numbers and outputs the encoding for a list that is precisely similar except that the last element has been reduced by 1. (If the last element in the input list is 0, the output will be the same as the input; if the input list is empty, the function will return the encoding for a list containing 0 as its only element.)

Theorem 8.3 *The function* `decrement-last` *is primitive recursive.*

Proof: The first step is to define a function to play the role of f_h; we'll call it `decrement-last-helper`. Recall that the job of such a helper function is to compute the value that the function we ultimately want to define (in this case, `decrement-last`), given the input to that function (in this case, the encoding for a list) and a list of all of the outputs that that function would compute from smaller inputs.

If x_0 is the encoding for a non-empty list and x_1 is a list of "previous outputs" of `decrement-last`, (`decrement-last`$(x_0 - 1), \ldots,$ `decrement-last`(0)), then we can recover `decrement-last`(`cdr`(x_0)) from x_1 by selecting the element that is followed by `cdr`(x_0) other elements. The zero-based index for that element is `length`$(x_1) - 1 -$ `cdr`(x_0).

$$\texttt{decrement-last-helper}(x_0, x_1) = \begin{cases} \texttt{cons}(\texttt{predecessor}(\texttt{car}(x_0)), \texttt{nil}()) & \text{if } \texttt{null?}(\texttt{cdr}(x_0)), \\ \texttt{cons}(\texttt{car}(x_0), \texttt{list-ref}(x_1, \texttt{length}(x_1) - 1 - \texttt{cdr}(x_0))) & \\ \quad \text{otherwise}, & \end{cases}$$

or, in our notation,

```
decrement-last-helper ≡
    [¿2 [∘21 null? [∘21 cdr pr20]]
        [∘22 cons [∘21 predecessor [∘21 car pr20]] nil2]
        [∘22 cons
            [∘21 car pr20]
            [∘22 list-ref
                pr21
                [∘22 subtract [∘21 predecessor [∘21 length pr21]]
                              [∘21 cdr pr20]]]]]
```

The `decrement-last` function itself is then defined from its helper:

$$\texttt{decrement-last} \equiv [\tilde{\Upsilon}_0 \ \texttt{decrement-last-helper}]$$

■

Since we'll often need to select an element from a list of previous values of a function that is being defined by course-of-values recursion, it will be helpful to separate out the `list-ref-from-end` function, which recovers an element of a list, given its zero-based position from the *end* of the list:

Theorem 8.4 *The function* `list-ref-from-end` *is primitive recursive.*

Proof:

```
list-ref-from-end ≡ [∘₂² list-ref
                        pr₂⁰
                        [∘₂² subtract
                            [∘₂¹ predecessor [∘₂¹ length pr₂⁰]]
                            pr₂¹]]
```

■

Using this function clarifies the structure of `decrement-last-helper`:

```
decrement-last-helper ≡
    [¿₂ [∘₂¹ null? [∘₂¹ cdr pr₂⁰]]
        [∘₂² cons [∘₂¹ predecessor [∘₂¹ car pr₂⁰]] nil₂]
        [∘₂² cons
            [∘₂¹ car pr₂⁰]
            [∘₂² list-ref-from-end pr₂¹ [∘₂¹ cdr pr₂⁰]]]]
```

Exercises

8–1 Compute C_6 by hand.

8–2 Use course-of-values recursion to define the `fibonacci` function, which inputs a natural number denoting a (zero-based) position in the Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, \ldots$ and outputs the natural number that occupies that position. The Fibonacci sequence is specified the recursion equations

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_{t+2} &= F_t + F_{t+1} \qquad \text{for every natural number } t. \end{aligned}$$

8–3 The *Hofstadter G* sequence is defined by the recursion equations

$$\begin{aligned} G_0 &= 0, \\ G_{t+1} &= t + 1 - G_{G_t} \qquad \text{for every natural number } t. \end{aligned}$$

Use course-of-values recursion to define the `hofstadter-g` function, which inputs a natural number n and outputs G_n.

Chapter 9

Additional List Functions

Course-of-values recursion makes it straightforward to develop other commonly used list functions. For example, we can define end-of-list analogues of `car`, `cdr`, and `cons`:

Theorem 9.1 *The functions* `last`*,* `all-but-last`*, and* `cons-at-end` *are all primitive recursive.*

Proof:

$$\begin{aligned}
\texttt{last} \equiv\ & [\tilde{\Upsilon}_0\ [\textit{¿}_2\ [\circ_2^1\ \texttt{null?}\ [\circ_2^1\ \texttt{cdr}\ \texttt{pr}_2^0]] \\
& \quad [\circ_2^1\ \texttt{car}\ \texttt{pr}_2^0] \\
& \quad [\circ_2^2\ \texttt{list-ref-from-end}\ \texttt{pr}_2^1\ [\circ_2^1\ \texttt{cdr}\ \texttt{pr}_2^0]]]]]
\end{aligned}$$

$$\begin{aligned}
\texttt{all-but-last} \equiv\ & [\tilde{\Upsilon}_0\ [\textit{¿}_2\ [\circ_2^1\ \texttt{null?}\ [\circ_2^1\ \texttt{cdr}\ \texttt{pr}_2^0]] \\
& \quad \texttt{nil}_2 \\
& \quad [\circ_2^2\ \texttt{cons} \\
& \qquad [\circ_2^1\ \texttt{car}\ \texttt{pr}_2^0] \\
& \qquad [\circ_2^2\ \texttt{list-ref-from-end} \\
& \qquad\quad \texttt{pr}_2^1 \\
& \qquad\quad [\circ_2^1\ \texttt{cdr}\ \texttt{pr}_2^0]]]]]
\end{aligned}$$

$$\begin{aligned}
\texttt{cons-at-end} \equiv\ & [\tilde{\Upsilon}_1\ [\textit{¿}_3\ [\circ_3^1\ \texttt{null?}\ \texttt{pr}_3^1] \\
& \quad [\circ_3^2\ \texttt{cons}\ \texttt{pr}_3^0\ \texttt{nil}_3] \\
& \quad [\circ_3^2\ \texttt{cons} \\
& \qquad [\circ_3^1\ \texttt{car}\ \texttt{pr}_3^1] \\
& \qquad [\circ_3^2\ \texttt{list-ref-from-end} \\
& \qquad\quad \texttt{pr}_3^2 \\
& \qquad\quad [\circ_3^1\ \texttt{cdr}\ \texttt{pr}_3^1]]]]]
\end{aligned}$$

■

From the last example, it is straightforward to abstract the general pattern of direct list recursion. We'll denote this pattern by $[\vec{\Upsilon}_n\ f\ g]$, where n is the number of "fixed" inputs that precede the last one (which is interpreted as the encoding for a list), f is a function that applies to the fixed inputs to generate the output in the base case (where the list is empty), and g is a function that applies to the fixed inputs, the car of the list (when it is not empty), and the recursive result for the cdr of the list to generate the output in any of the non-base cases.

Theorem 9.2 *For any primitive recursive functions f of valence n and g of valence $n+2$, $[\vec{\Upsilon}_n\ f\ g]$ is primitive recursive.*

Proof:

$$
\begin{aligned}
[\vec{\Upsilon}_n\ f\ g] \equiv [\tilde{\Upsilon}_n\ [\text{¿}_{n+2}\ & [\circ^1_{n+2}\ \texttt{null?}\ \texttt{pr}^n_{n+2}] \\
& [\circ^n_{n+2}\ f\ \texttt{pr}^0_{n+2}\ \ldots\ \texttt{pr}^{n-1}_{n+2}] \\
& [\circ^{n+2}_{n+2}\ g \\
& \qquad \texttt{pr}^0_{n+2} \\
& \qquad\quad \vdots \\
& \qquad \texttt{pr}^{n-1}_{n+2} \\
& \qquad [\circ^1_{n+2}\ \texttt{car}\ \texttt{pr}^n_{n+2}] \\
& \qquad [\circ^2_{n+2}\ \texttt{list-ref-from-end} \\
& \qquad\quad \texttt{pr}^{n+1}_{n+2} \\
& \qquad\quad [\circ^1_{n+2}\ \texttt{cdr}\ \texttt{pr}^n_{n+2}]]]]]
\end{aligned}
$$

■

For instance, $[\vec{\Upsilon}_0\ \texttt{zero}\ \texttt{add}]$ is a function that determines the sum of the elements of a given list.

Here are two additional patterns that often come in handy. Given a function f of valence $n+1$, we can define a similar function $[\vec{\circ}_n\ f]$ (read "map f") that takes the encoding for a list L as its last input and outputs a list of values that f outputs as it receives the elements of L (with the other inputs held constant).

Theorem 9.3 *For every primitive recursive function f of valence $n+1$, $[\vec{\circ}_n\ f]$ is a primitive recursive function.*

Proof:

$$[\vec{\circ}_n\ f] \equiv [\vec{\Upsilon}_n\ \texttt{nil}_n\ [\circ^2_{n+2}\ \texttt{cons}\ [\circ^{n+1}_{n+2}\ f\ \texttt{pr}^0_{n+2}\ \ldots\ \texttt{pr}^n_{n+2}]\ \texttt{pr}^{n+1}_{n+2}]]$$

■

Given a function p of valence $n+1$, $\texttt{[}\vec{\wedge}_n\ p\texttt{]}$ (read "all p") is a predicate that takes the encoding for a list L as its last input and determines whether p outputs a truish value for every element of L (with the other inputs held constant), outputting 1 if so and 0 if not.

Theorem 9.4 *For every primitive recursive predicate p of valence $n+1$, $\texttt{[}\vec{\wedge}_n\ p\texttt{]}$ is a primitive recursive function.*

Proof:

$$\texttt{[}\vec{\wedge}_n\ p\texttt{]} \equiv \texttt{[}\vec{\Upsilon}_n\ \texttt{one}_n\ \texttt{[}\circ^2_{n+2}\ \texttt{and [}\circ^{n+1}_{n+2}\ p\ \texttt{pr}^0_{n+2}\ \texttt{...}\ \texttt{pr}^n_{n+2}\texttt{]}\ \texttt{pr}^{n+1}_{n+2}\texttt{]]}$$

■

Exercises

9–1 Define the `list-product` function, which inputs the encoding for a list and outputs the product of its elements (or 1, if the list is empty).

9–2 Prove that the function `list-max`, which inputs the encoding for a list and outputs the greatest element in that list (or 0, if the list is empty), is primitive recursive. (Hint: First define `max`, which takes two inputs and outputs the greater one.)

9–3 Define a list-generation operator $\vec{\star}_n$ such that, for any function f of valence $n+1$ and any natural numbers $x_0, \ldots, x_n$, $\texttt{[}\vec{\star}_n\ f\texttt{]}(x_0, \ldots, x_{n-1}, x_n)$ is the encoding of the list

$$(f(x_0, \ldots, x_{n-1}, 0), f(x_0, \ldots, x_{n-1}, 1), \ldots, f(x_0, \ldots, x_{n-1}, x_n - 1)).$$

Show that, if f is primitive recursive, so is $\texttt{[}\vec{\star}_n\ f\texttt{]}$.

Chapter 10

Partial Functions

We now extend the concept of a function to include the kind of mathematical entity that maps every tuple in its domain to *at most one* member of its range. Such an entity is called a *partial function*, and is said to be *undefined at* any input that it does not map to anything. We shall write '$f(x_0, \ldots, x_{n-1})\uparrow$' to indicate that the partial function f is undefined at $(x_0, \ldots, x_{n-1})$. Similarly, we shall write '$f(x_0, \ldots, x_{n-1})\downarrow$' to indicate that f is defined at $(x_0, \ldots, x_{n-1})$, without indicating what value f maps that input to.

Primitive recursive functions are all *total* functions, defined for all inputs. The set of all total functions is a proper subset of the set of all partial functions.

Composition and Recursion of Partial Functions

To extend the operations of composition and recursion to partial functions, we must provide for the possibility of encountering undefinedness at some point during the application of the higher-order operation. We shall follow the natural convention that if we encounter undefinedness at any point, the function that we are constructing is undefined for the given inputs.

Specifically, in the composition equation

$$h(x_0, \ldots, x_{n-1}) = f(g_0(x_0, \ldots, x_{n-1}), \ldots, g_{m-1}(x_0, \ldots, x_{n-1})),$$

we might find that one or more of the "inner" functions $g_0, \ldots, g_{m-1}$ is undefined at the input $(x_0, \ldots, x_{n-1})$, and in any such case we shall stipulate that h is also undefined at that input. Moreover, even if all of the inner functions yield values $y_0 = g_0(x_0, \ldots, x_{n-1}), \ldots, y_{m-1} = g_{m-1}(x_0, \ldots, x_{n-1})$, we might find that f is undefined at the input $(y_0, \ldots y_{m-1})$. In that case, too, we stipulate that h is undefined at $(x_0, \ldots, x_{n-1})$.

Similarly, in the recursion equations

$$\begin{aligned} h(x_0, \ldots, x_{n-1}, 0) &= f(x_0, \ldots, x_{n-1}), \\ h(x_0, \ldots, x_{n-1}, t+1) &= g(t, h(x_0, \ldots, x_{n-1}, t), x_0, \ldots, x_{n-1}), \end{aligned}$$

we might discover that $f(x_0, \ldots, x_{n-1})\uparrow$; if so, then $h(x_0, \ldots, x_{n-1}, 0)\uparrow$. And likewise if, for some natural number t, we find either that $h(x_0, \ldots, x_{n-1}, t)\uparrow$ or that $h(x_0, \ldots, x_{n-1}, t) = y$ and $g(t, y, x_0, \ldots, x_{n-1})\uparrow$, then $h(x_0, \ldots, x_{n-1}, t+1)$ is also undefined.

Without changing their definitions, we will extend all of our other "higher-order" operations ($\dot{\iota}_n$, $\bar{\forall}_n$, $\bar{\Sigma}_n$, $\bar{\mu}_n$, $\circledcirc_n$, $\vec{\Upsilon}_n$, $\vec{\circ}_n$, and $\vec{\Lambda}_n$), so that they too become ways of defining partial functions in terms of other partial functions.

Unbounded Minimization

Of course, it would be idle to extend composition and recursion to partial functions if we stayed within the class of primitive recursive functions. To go beyond that class, we shall now adopt a third elementary mechanism for constructing new functions from our primitives: The *unbounded minimization* of an $(n+1)$-ary function p is an n-ary function μp. Given the n-tuple $(x_0, \ldots, x_{n-1})$, μp outputs the least natural number t such that $p(x_0, \ldots, x_{n-1}, t) > 0$.

Thus the general equation defining an unbounded minimization is

$$\mu p(x_0, \ldots, x_{n-1}) = \min_{t \in \mathbb{N}} \left(p(x_0, \ldots, x_{n-1}, t) > 0\right).$$

We shall write '$\texttt{[}\mu_n\ p\texttt{]}$' to denote μp in our constructions.

Since the range of t is unbounded, it is natural to ask what happens if there is no natural number t such that $p(x_0, \ldots, x_{n-1}, t) > 0$. In that case, $\mu p(x_0, \ldots, x_{n-1})\uparrow$. The unbounded minimization of a function may be undefined for certain inputs even if the function itself is total. For example, we could define "true" subtraction, the exact inverse of addition, like this:

$$\texttt{minus}(x_0, x_1) = \min_{t \in \mathbb{N}} \left(x_0 = x_1 + t\right),$$

or, in our official notation,

$$\texttt{minus} \equiv \texttt{[}\mu_2\ \texttt{[}\circ_3^2\ \texttt{equal?}\ \texttt{pr}_3^0\ \texttt{[}\circ_3^2\ \texttt{add}\ \texttt{pr}_3^1\ \texttt{pr}_3^2\texttt{]]]}.$$

Whenever $x_0 \geq x_1$, it turns out that $\texttt{minus}(x_0, x_1) = \texttt{subtract}(x_0, x_1)$; but when $x_0 < x_1$, $\texttt{subtract}(x_0, x_1) = 0$, while $\texttt{minus}(x_0, x_1)\uparrow$.

For every natural number n, there is an extreme partial function of valence n that is undefined throughout its domain. I don't know of a standard name for these functions, so I propose to use the name `mist` for the nullary version, defined by

$$\texttt{mist} \equiv \texttt{[}\mu_0\ \texttt{zero}_1\texttt{]},$$

and, in accordance with an earlier convention, '$\texttt{mist}_n$' as shorthand for '$\texttt{[}\circ_n^0$ `mist`]', which denotes the everywhere-undefined function of valence n.

Functions that can be constructed from our primitives (`zero`, `successor`, and the $\texttt{pr}_n^m$ functions) using composition, recursion, and unbounded minimization are called *partial recursive* functions. Naturally, all primitive recursive

functions are also partial recursive, since they can be defined from the primitives using composition and recursion. Moreover, the higher-order operations $¿_n$, $\bar{\forall}_n$, $\bar{\Sigma}_n$, $\bar{\mu}_n$, $\circledcirc_n$, $\vec{\Upsilon}_n$, $\vec{\circ}_n$, and $\vec{\wedge}_n$, operating on partial recursive functions, construct partial recursive functions. (The proofs are identical to the corresponding proofs for primitive recursive functions, even though the meanings of the constructions are now somewhat different.)

Exercises

10–1 The partial function `exactly-halve` is singulary. If its input is even, `exactly-halve` outputs the result of dividing the input by 2. If the input is odd, `exactly-halve` does not output anything; the function is undefined for odd inputs, since they cannot be exactly halved. Prove that `exactly-halve` is partial recursive.

10–2 One might be tempted to define the `exactly-halve` function in the previous exercise as

$$\texttt{[¿}_1\ \texttt{even?}\ \texttt{[}\circ_1^2\ \texttt{quotient}\ \texttt{pr}_1^0\ \texttt{two}_1\texttt{]}\ \texttt{mist}_1\texttt{]}$$

However, this construction expresses a partial function that is everywhere undefined (in other words, it's $\texttt{mist}_1$ under another name). Explain why.

10–3 Show that, for every primitive recursive function f, there is a partial recursive function f' such that $f'(x_0, \ldots, x_{n-1})\uparrow$ if $f(x_0, \ldots, x_{n-1}) = 0$, and $f'(x_0, \ldots, x_{n-1}) = f(x_0, \ldots, x_{n-1})$ otherwise.

Chapter 11

Encoding Partial Recursive Functions

Since we've seen data values of a variety of types encoded as natural numbers, it is natural to wonder whether functions themselves might be so encoded. Unfortunately, no such encoding exists for functions generally. Even if we consider only total, singulary functions, there are just too many of them for each one to be mapped to a different natural number! The proof (by contradiction) is entertaining:

Suppose that there were an encoding that mapped every total, singulary function f to a different natural number $\bar{f}$. Then there would also be a total, binary function e such that

$$e(x, \bar{f}) = f(x)$$

for any total, singulary function f and every natural number x. The function e would exist even if the encoding were not a surjection, that is, even if some natural numbers did not encode any function, because we could simply stipulate that e outputs 0 whenever its second input does not encode a function. The function e would be a *universal* function, capable of mimicking the behavior of *every* total, singulary function, and using its second input to determine which of those functions to mimic.

But if there were such a function e, then there would also have to be a total, singulary function e' such that

$$e'(x) = e(x, x) + 1.$$

In effect, e' takes any natural number, decodes it to get a function, applies that function to its own encoding, and adds 1 to the output of that function. This seems like an unusual thing to do, but it's clear that it would be possible if we had a way of encoding every function to begin with.

The contradiction appears when we consider that, since e' is a total, singulary function, it too will have an encoding $\bar{e'}$, and e will have to take this encoding as an input; indeed, since $\bar{e'}$ is a natural number, e will have to be able to accept it

in either input position, or in both. In the special case where both of the inputs are $\bar{e'}$, the first of the two equations above says that

$$e(\bar{e'}, \bar{e'}) = e'(\bar{e'}),$$

while the second says that

$$e'(\bar{e'}) = e(\bar{e'}, \bar{e'}) + 1.$$

It follows that $e(\bar{e'}, \bar{e'}) = e(\bar{e'}, \bar{e'}) + 1$, which is impossible.

Thus not all total, singulary functions can be encoded as natural numbers, nor (obviously) can all total functions of all valences or all partial functions, since these sets include the total, singulary functions as a subset.

However, it turns out that it is possible to encode and enumerate all *partial recursive* functions, by encoding and enumerating their construction recipes—the very programs that we have been writing to define them and to prove that they are partial recursive. Let's review the concept of a partial recursive function and look at what it would take to encode the programs that define them.

First, there are the primitive functions, beginning with the `zero` function. It seems fitting to use the natural number 0 as the encoding for `zero`.

$$\texttt{encode-zero-function} \equiv \texttt{zero}$$
$$\texttt{zero-function?} \equiv \texttt{zero?}$$

We'll need an encoding for the `successor` function. Again, there is a strikingly obvious choice, namely 1.

$$\texttt{encode-successor-function} \equiv \texttt{one}$$
$$\texttt{successor-function?} \equiv [\circ_1^2\ \texttt{equal?}\ \texttt{pr}_1^0\ \texttt{one}_1]$$

Next, we'll need encodings for all of the projection functions. There are infinitely many of these, so if we aren't careful, we could use up all of our natural numbers on the projection functions alone. To keep this from happening, let's select out an infinite subset of the remaining natural numbers and use just that subset to represent projection functions. For reasons that will shortly become obvious, the subset that I'll choose is $\{4k + 2 \mid k \in \mathbb{N}\}$—that is, the subset comprising 2, 6, 10, 14, 18, and so on. The k in this construction can be our encoding for the pair $\texttt{cons}(m, n)$, where n is the number of inputs to the projection function and m is the zero-based position of the input that the projection function passes along as its output. So, for instance, the encoding for $\texttt{pr}_2^0$ will be $4 \cdot \texttt{cons}(0, 2) + 2$, or 22, and the encoding for $\texttt{pr}_3^1$ will be $4 \cdot \texttt{cons}(1, 3) + 2$, which works out to be 58.

$$\texttt{encode-projection-function}(m, n) = 4 \cdot \texttt{cons}(m, n) + 2$$

and so

$$\begin{aligned}\texttt{encode-projection-function} \equiv [\circ_2^2\ &\texttt{add}\\ &[\circ_2^2\ \texttt{multiply}\ \texttt{four}_2\ \texttt{cons}]\\ &\texttt{two}_2]\end{aligned}$$

where

$$\begin{aligned} \texttt{three} &\equiv [\circ_0^1\ \texttt{successor two}] \\ \texttt{four} &\equiv [\circ_0^1\ \texttt{successor three}] \end{aligned}$$

Under this approach, the encoding function is not "onto"—there will be some natural numbers that do not encode any programs. For instance, 2 does not encode a program, since $2 = 4 \cdot 0 + 2$, and 0 does not encode a pair. Nor does 6 encode a program; $6 = 4 \cdot \texttt{cons}(0, 0) + 2$, but there is no such projection function as $\texttt{pr}_0^0$. (The upper index in a projection must be strictly less than the lower index.) This is not a problem as long as (a) the encoding function is one-to-one, and (b) we can use a primitive recursive function to distinguish the valid encodings from the invalid ones.

In particular, we can still define decoding functions that will recover the indices m and n from the encoding for $\texttt{pr}_n^m$:

$$\begin{aligned} &\texttt{projection-indices} \equiv [\circ_1^2\ \texttt{quotient} \\ &\qquad\qquad [\circ_1^2\ \texttt{subtract pr}_1^0\ \texttt{two}_1] \\ &\qquad\qquad \texttt{four}_1] \\ &\texttt{upper-projection-index} \equiv [\circ_1^1\ \texttt{car projection-indices}] \\ &\texttt{lower-projection-index} \equiv [\circ_1^1\ \texttt{cdr projection-indices}] \\ &\texttt{projection-function?} \equiv [\circ_1^2\ \texttt{and} \\ &\qquad\qquad [\circ_1^2\ \texttt{equal?} \\ &\qquad\qquad\quad [\circ_1^2\ \texttt{remainder pr}_1^0\ \texttt{four}_1] \\ &\qquad\qquad\quad \texttt{two}_1] \\ &\qquad\qquad [\circ_1^2\ \texttt{less?} \\ &\qquad\qquad\quad \texttt{upper-projection-index} \\ &\qquad\qquad\quad \texttt{lower-projection-index}]] \end{aligned}$$

The inputs to selector functions for the projection indices can be any natural numbers, regardless of whether or not they are valid encodings for projection functions. Since the selectors are primitive recursive functions, they output numerical results regardless. However, those results are of no interest to us, and we shall ensure that none of the functions that we define subsequently depends on them.

Now let us consider the three mechanisms for constructing partial recursive functions from the primitives: composition, recursion, and unbounded minimization. Each of these three mechanisms can be applied to infinitely many different operands, so again we'll need to take care not to use up our encodings for any one of them. Let's reserve $\{4k + 3 \mid k \in \mathbb{N}\}$ for compositions, $\{4k + 4 \mid k \in \mathbb{N}\}$ for recursions, and $\{4k + 5 \mid k \in \mathbb{N}\}$ for minimizations. (Now it is apparent why the multiplier is 4: we need four different infinite sets of encodings for projections, compositions, recursions, and minimizations, so we set up four different residue classes to represent them.)

When encoding a composition as $4k+3$, the k can be the encoding for a list in which the initial element is the upper index (the valence m of the "outer"

function f in the composition), the next element is the lower index (the valence n of the "inner" functions $g_0, \ldots, g_{m-1}$), and the remaining elements are the encodings of $f, g_0, \ldots, g_{m-1}$ themselves, which I'll call the *components* of the composition. So, for instance, the encoding for $[\circ_3^1$ `successor` $\mathtt{pr}_3^1]$ will be $4 \cdot \mathtt{cons}(1, \mathtt{cons}(3, \mathtt{cons}(1, \mathtt{cons}(58, \mathtt{nil}())))) + 3$, or 295147905179352826507.

In our encoder for functions defined by composition, which we'll call `encode-composition`, we'll assume that the components have already been assembled into a list, so that there will always be exactly three inputs—m, n, and the encoding c for the list of components:

$$\mathtt{encode\text{-}composition}(m, n, c) = 4 \cdot \mathtt{cons}(m, \mathtt{cons}(n, c)) + 3,$$

```
encode-composition ≡ [∘₃² add
                        [∘₃² multiply
                            four₃
                            [∘₃² cons pr₃⁰ [∘₃² cons pr₃¹ pr₃²]]]
                        three₃]
composition-parameters ≡ [∘₁² quotient
                            [∘₁² subtract pr₁⁰ three₁]
                            four₁]
upper-composition-index ≡ [∘₁¹ car composition-parameters]
lower-composition-index ≡ [∘₁¹ car [∘₁¹ cdr composition-parameters]]
components ≡ [∘₁¹ cdr [∘₁¹ cdr composition-parameters]]
composition-function? ≡ [∘₁² and
                            [∘₁² equal?
                                [∘₁² remainder pr₁⁰ four₁]
                                three₁]
                            [∘₁² greater-or-equal?
                                [∘₁¹ length composition-parameters]
                                three₁]]
```

Similarly, when encoding a recursion as $4k + 4$, k can be the encoding of a three-element list in which the initial element is the subscript indicating the number of fixed inputs to the recursion and the other two elements are the encodings for the base function f and the step function g. So, for instance, the encoding for our `predecessor` function, defined as $[\Upsilon_0$ `zero` $\mathtt{pr}_2^0]$, will be $4 \cdot \mathtt{cons}(0, \mathtt{cons}(0, \mathtt{cons}(14, \mathtt{nil}()))) + 4$, or 262160.

$$\mathtt{encode\text{-}recursion}(n, f, g) = 4 \cdot \mathtt{cons}(n, \mathtt{cons}(f, \mathtt{cons}(g, \mathtt{nil}()))) + 4,$$

```
encode-recursion ≡
    [∘₃² add
        [∘₃² multiply
            four₃
            [∘₃² cons pr₃⁰ [∘₃² cons pr₃¹ [∘₃² cons pr₃² [∘₃⁰ nil]]]]]
        four₃]
recursion-parameters ≡ [∘₁² quotient [∘₁² subtract pr₁⁰ four₁] four₁]
recursion-index ≡ [∘₁¹ car recursion-parameters]
base-operand ≡ [∘₁¹ car [∘₁¹ cdr recursion-parameters]]
step-operand ≡ [∘₁¹ car [∘₁¹ cdr [∘₁¹ cdr recursion-parameters]]]
recursion-function? ≡ [∘₁² and
                           [∘₁² and
                               [∘₁² divides? four₁ pr₁⁰]
                               [∘₁² greater-or-equal? pr₁⁰ four₁]]
                           [∘₁² equal?
                               [∘₁¹ length recursion-parameters]
                               three₁]]
```

Finally, when encoding a minimization as $4k + 5$, k can be the encoding of a pair in which the car is the valence of the function we are defining and the cdr is the encoding for the function to which the minimization operation is being applied. For instance, the encoding for `mist`, which is defined as $[\mu_0\ \texttt{zero}_1]$, will be $4 \cdot \texttt{cons}(0, 55) + 5$, or 449 (since 55 is the encoding for $\texttt{zero}_1$, that is, for $[\circ_1^0\ \texttt{zero}]$).

$$\texttt{encode-minimization}(n, p) = 4 \cdot \texttt{cons}(n, p) + 5,$$

so that

```
five ≡ [∘₀¹ successor four]
encode-minimization ≡ [∘₂² add [∘₂² multiply four₂ cons] five₂]
min-parameters ≡ [∘₁² quotient [∘₁² subtract pr₁⁰ five₁] four₁]
minimization-index ≡ [∘₁¹ car min-parameters]
minimization-predicate ≡ [∘₁¹ cdr min-parameters]
minimization-function? ≡ [∘₁² and
                              [∘₁² and
                                  [∘₁² equal?
                                      [∘₁² remainder pr₁⁰ four₁]
                                      one₁]
                                  [∘₁² greater-or-equal? pr₁⁰ five₁]]
                              [∘₁¹ positive? min-parameters]]
```

It is possible to determine the encoding for any of the partial recursive functions that we have defined, then, by performing elementary arithmetic operations that can be expressed as primitive recursive functions.

From the encoding for any function, we can recover its valence:

```
valence ≡ [¿1 zero-function?
              zero1
              [¿1 successor-function?
                  one1
                  [¿1 projection-function?
                      lower-projection-index
                      [¿1 composition-function?
                          lower-composition-index
                          [¿1 recursion-function?
                              [∘11 successor recursion-index]
                              [¿1 minimization-function?
                                  minimization-index
                                  zero1]]]]]]
```

The final $\texttt{zero}_1$ supplies the default error value when `valence` receives an input that does not encode a function.

Now that we have the `valence` function, we can tighten up the last three classification predicates, which are not yet doing the complete job of validating their inputs as correct function encodings. Specifically, we can now add three new conditions to `composition-function?`: (a) that the length of the list of components is one greater than the upper composition index, (b) that the valence of the first component is equal to the upper composition index, and (c) that the valences of all of the other components are equal to the lower composition index:

```
checked-composition-function? ≡
    [∘12 and
        composition-function?
        [∘12 and
            [∘12 equal?
                [∘11 length components]
                [∘11 successor upper-composition-index]]
            [∘12 and
                [∘12 equal?
                    [∘11 valence [∘11 car components]]
                    upper-composition-index]
                [∘12 [⃗∧1 [∘12 equal? pr02 [∘12 valence pr12]]]
                    lower-composition-index
                    [∘11 cdr components]]]]]
```

Similarly, we can add to `recursion-function?` the conditions that the valence of the base operand is equal to the recursion index, and that the valence

of the step operand is two greater than the recursion index:

```
checked-recursion-function? ≡
    [∘₁² and
       recursion-function?
       [∘₁² and
          [∘₁² equal?
             [∘₁¹ valence base-function]
             recursion-index]
          [∘₁² equal?
             [∘₁¹ valence step-function]
             [∘₁² add recursion-index two₁]]]]
```

And we can add to `minimization-function?` the condition that the valence of the minimization predicate is one greater than the minimization index.

```
checked-minimization-function? ≡
    [∘₁² and
       minimization-function?
       [∘₁² equal?
          [∘₁¹ valence minimization-predicate]
          [∘₁¹ successor minimization-index]]]
```

The singulary function called `function?`, then, does *all* of the necessary tests to confirm that its input correctly encodes a partial recursive function: It confirms that the encoding satisfies one of the classification predicates developed above, including the valence checks, and moreover, using course-of-values recursion, it confirms (what the previous classification predicates took for granted) that the components of a composition function, the base and step operands of a recursion function, and the minimization predicate of a minimization function are themselves partial recursive functions.

```
function? ≡
    [Υ̃_0 [∘_2^2 or
            [∘_2^1 zero-function? pr_2^0]
            [∘_2^2 or
                [∘_2^1 successor-function? pr_2^0]
                [∘_2^2 or
                    [∘_2^1 projection-function? pr_2^0]
                    [∘_2^2 or
                        [∘_2^2 and
                            [∘_2^1 checked-composition-function? pr_2^0]
                            [∘_2^2 [Λ⃗_1 list-ref-from-end]
                                pr_2^1
                                [∘_2^1 components pr_2^0]]]]
                        [∘_2^2 or
                            [∘_2^2 and
                                [∘_2^1 checked-recursion-function? pr_2^0]
                                [∘_2^2 and
                                    [∘_2^2 list-ref-from-end
                                        pr_2^1
                                        [∘_2^1 base-function pr_2^0]]
                                    [∘_2^2 list-ref-from-end
                                        pr_2^1
                                        [∘_2^1 step-function pr_2^0]]]]]
                            [∘_2^2 and
                                [∘_2^1 checked-minimization-function?
                                    pr_2^0]
                                [∘_2^2 list-ref-from-end
                                    pr_2^1
                                    [∘_2^1 minimization-predicate
                                        pr_2^0]]]]]]]]
```

Exercises

11–1 It is not quite accurate to say that the encoding system presented in this section gives a unique encoding for every partial recursive function, since the same partial recursive function can be computed in various ways, using different programs. Each program receives a unique encoding, but various programs that compute the same function have different encodings.

Since the relationship between the partial recursive functions and the natural numbers that encode them is not one-to-one, how do we know that every partial

recursive function has at least one encoding?

11–2 Compute the encoding for the `add` function. You may use a computer to help with the calculation.

11–3 Define a primitive recursive function `code-size` that inputs the encoding for a program that computes a partial recursive function and outputs a measure of the complexity of that program, counting 1 for each reference to a primitive function, 1 plus the sum of the code-sizes of all of the component functions in a composition, 1 plus the sum of the code-sizes of the base and step functions in a recursion, and 1 plus the code-size of the operand predicate in an unbounded minimization.

Chapter 12

Computations

Now that we have a method for encoding programs that express the partial recursive functions, we can also work up a system for encoding the computation that one would perform in order to work out the result of applying a partial recursive function to specific inputs—provided that there *is* such a result, i.e., that the function is defined for those inputs.

We'll represent a computation as a data structure with four components: the encoding of some program for the function that is being applied, the encoding for the list of inputs to which it is being applied, the output resulting from the computation, and the encoding for a list of *subcomputations*—other applications of functions to inputs whose values must be computed as part of the main computation. When the function that is being applied is a primitive (`zero`, `successor`, or a projection function), the list of subcomputations will be empty. Compositions, recursions, and minimizations, however, always have subcomputations.

The arrangement of the four components is arbitrary; let's just make them the four constituents of a pair of pairs. Here are the selectors for recovering these components from the natural number that encodes a computation:

$$\begin{aligned} \texttt{program} &\equiv [\circ_1^1\ \texttt{car car}] \\ \texttt{inputs} &\equiv [\circ_1^1\ \texttt{cdr car}] \\ \texttt{output} &\equiv [\circ_1^1\ \texttt{car cdr}] \\ \texttt{subcomputations} &\equiv [\circ_1^1\ \texttt{cdr cdr}] \end{aligned}$$

The `launch-checks?` predicate tests a whether a given input meets the starting preconditions for a computation: all four components must be present, the `program` component must be the encoding for a partial recursive function, and the length of the list of inputs must be equal to the valence of that function.

```
launch-checks? ≡ [∘₁² and
                  positive?
                  [∘₁² and
                      [∘₁¹ positive? car]
                      [∘₁² and
                          [∘₁¹ positive? cdr]
                          [∘₁² and
                              [∘₁¹ function? program]
                              [∘₁² equal?
                                  [∘₁¹ length inputs]
                                  [∘₁¹ valence program]]]]]]
```

The `zero-computation?` predicate tests whether its input is the encoding for a computation in which the function to be applied is `zero`. It determines whether (a) the program in the computation is the correct encoding for the `zero` function, (b) the output is 0, and (c) the list of subcomputations is null:

```
zero-computation? ≡ [∘₁² and
                     [∘₁¹ zero-function? program]
                     [∘₁² and
                         [∘₁¹ zero? output]
                         [∘₁¹ null? subcomputations]]]
```

The `successor-computation?` predicate tests whether its input is the encoding for a computation in which the function to be applied is `successor`. It checks that (a) the program in the computation is the correct encoding for the `successor` function, (b) the output really is the successor of the input, and (c) the list of subcomputations is null:

```
successor-computation? ≡
    [∘₁² and
        [∘₁¹ successor-function? program]
        [∘₁² and
            [∘₁² equal? [∘₁¹ successor [∘₁¹ car inputs]] output]
            [∘₁¹ null? subcomputations]]]
```

The `projection-computation?` predicate tests whether its input is the encoding for a computation in which the function to be applied is a projection function. It determines whether (a) the program encodes a projection function, (b) the output matches the input at the position indicated by the upper projection index, and (c) the list of subcomputations is null.

```
projection-computation? ≡
    [∘₁² and
        [∘₁² projection-function? program]
        [∘₁² and
            [∘₁² equal?
                [∘₁² list-ref
                    inputs
                    [∘₁¹ upper-projection-index program]]
                output]
            [∘₁¹ null? subcomputations]]]
```

The `composition-computation?` predicate tests whether its input is the encoding for a computation in which the function to be applied is a composition, say $[\circ_n^m\ f\ g_0\ \ldots\ g_{m-1}]$. When such a function is applied to inputs $x_0 \ldots x_{n-1}$, there will be a total of $m+1$ subcomputations, one for each of the components. In drawing up the list of subcomputations, we will adopt the convention that the subcomputations are arranged in the same (left-to-right) order as the component functions.

The `composition-computation?` predicate, therefore, checks to make sure that (a) the program in the computation is a composition function, (b) the length of the list of subcomputations is one greater than the upper composition index, (c) the programs of the subcomputations are the components of the program in the main computation, (d) the inputs for all of the subcomputations except the first are the same as the inputs in the main computation; (e) the inputs for the first subcomputation are the outputs from the remaining subcomputations; and (f) the output of the first subcomputation is the output for the main computation.

Eventually, it will also be necessary to ensure that each of the subcomputations is itself a valid subcomputation of one of the six types, but we'll defer that part of the testing until we assemble the overall `computation?` predicate, at which point we can do a single course-of-values recursion to perform that check on all kinds of subcomputations.

```
composition-computation? ≡
    [∘¹₁ and
        [∘¹₁ composition-function? program]
        [∘¹₁ and
            [∘¹₁ equal?
                [∘¹₁ length subcomputations]
                [∘¹₁ successor
                    [∘¹₁ upper-projection-index program]]]
            [∘¹₁ and
                [∘¹₁ equal?
                    [∘¹₁ components program]
                    [∘¹₁ [⃗∘₀ program] subcomputations]]
                [∘¹₁ and
                    [∘¹₁ [⃗∧₁ equal?]
                        inputs
                        [∘¹₁ [⃗∘₀ inputs]
                            [∘¹₁ cdr subcomputations]]]
                    [∘¹₁ and
                        [∘¹₁ equal?
                            [∘¹₁ inputs [∘¹₁ car subcomputations]]
                            [∘¹₁ [⃗∘₀ output]
                                [∘¹₁ cdr subcomputations]]]
                        [∘¹₁ equal?
                            output
                            [∘¹₁ output
                                [∘¹₁ car subcomputations]]]]]]]]
```

The `recursion-computation?` predicate tests whether its input is the encoding for a computation in which the function to be applied is a recursion, say $[\Upsilon_n\ f\ g]$. When the last input to such a function is 0, there will be only one subcomputation (the application of f to the other inputs); when the last input is positive, there will be two (the recursive application of $[\Upsilon_n\ f\ g]$, with the last input decremented, and the application of g that post-processes the output of the recursion). In the latter case, we will adopt the convention that the application of g is the car of the list of subcomputations and the recursive application is the car of its cdr.

Thus the `recursion-computation?` predicate, therefore, has some intricate work to do. It checks to make sure that (a) the program encodes a recursion function and (b) the output of the first subcomputation is the output of the main computation.

It then tests whether the last input is 0. If so, it confirms that (c) the length of the list of subcomputations is 1, (d) the program of the subcomputation is the

base operand f of the program of the main computation, and (e) the inputs for the subcomputation are all but the last of the inputs for the main computation.

On the other hand, if the last input to the main computation is positive, the predicate confirms that (f) the length of the list of subcomputations is 2, (g) the program of the second subcomputation (the recursive one) is the same as the program in the main computation, (h) the inputs for the second subcomputation are the same as the inputs for the main computation, except that the last input has been decremented by 1, (i) the program of the first subcomputation is the step operand g of the program in the main computation, and (j) the inputs for the first subcomputation are, first, the predecessor of the last input of the main computation, then the output from the second subcomputation (i.e., the recursive result), then the rest of the inputs of the main computation.

```
recursion-computation? ≡
    [∘₁² and
        [∘₁¹ recursion-function? program]
        [∘₁² and
            [∘₁² equal?
                output
                [∘₁¹ output [∘₁¹ car subcomputations]]]
            [¿₁ [∘₁¹ zero? [∘₁¹ last inputs]]
                [∘₁² and
                    [∘₁² equal? [∘₁¹ length subcomputations] one₁]
                    [∘₁² and
                        [∘₁² equal?
                            [∘₁¹ base-operand program]
                            [∘₁¹ program [∘₁¹ car subcomputations]]]
                        [∘₁² equal?
                            [∘₁¹ all-but-last inputs]
                            [∘₁¹ inputs [∘₁¹ car subcomputations]]]]]
                [∘₁² and
                    [∘₁² equal? [∘₁¹ length subcomputations] two₁]
                    [∘₁² and
                        [∘₁² equal?
                            program
                            [∘₁¹ program
                                [∘₁¹ car [∘₁¹ cdr subcomputations]]]]
```

(continued on next page)

```
[∘₁² and
    [∘₁² equal?
        [∘₁¹ decrement-last inputs]
        [∘₁¹ inputs
            [∘₁¹ car
                [∘₁¹ cdr subcomputations]]]]
    [∘₁² and
        [∘₁² equal?
            [∘₁¹ step-operand program]
            [∘₁¹ program
                [∘₁¹ car subcomputations]]]
        [∘₁² equal?
            [∘₁² cons
                [∘₁¹ predecessor [∘₁¹ last inputs]]
                [∘₁² cons
                    [∘₁¹ output
                        [∘₁¹ car
                            [∘₁¹ cdr
                                subcomputations]]]
                    [∘₁¹ all-but-last inputs]]]
            [∘₁¹ inputs
                [∘₁¹ car subcomputations]]]]]]]]]]
```

The `minimization-computation?` predicate tests whether its input is the encoding for a computation in which the function to be applied is a minimization, say $[\mu_n\ p]$. When such a function is applied successfully, yielding an output, all of the subcomputations are applications of the function p to a list of inputs that include the inputs $x_0, \ldots, x_{n-1}$ to the main computation as well as the satisfaction candidate t (at the end). We'll conventionally arrange these subcomputations in such a way that the last input in each case matches the subcomputation's zero-based position in the list. In other words, the initial subcomputation will be the application of p to $x_0, \ldots, x_{n-1}, 0$, the next will be the application of p to $x_0, \ldots, x_{n-1}, 1$, and so on.

The number of subcomputations is one greater than the output r of the minimization function, since in order to find r it is necessary to discover that the values $0, \ldots, r-1$ do not satisfy p and that r does satisfy p. Thus the output of all but the last of the subcomputations is 0 and the output of the last one is 1.

When a minimization function is undefined for some particular combination of inputs, there is no corresponding computation, since the list of subcomputations would have to be infinite, and we have no way to encode infinitely long lists.

The `minimization-computation?` predicate, therefore, checks to make sure

that (a) the program in the computation encodes a minimization function, (b) the length of the list of subcomputations is the successor of the output, (c) the program for each subcomputation is the minimization predicate of the program in the main computation, (d) the inputs for each subcomputation are the inputs for the main computation together with the appropriate satisfaction candidate, and (e) the output for each subcomputation except the last is 0 and the output for the last subcomputation is 1. (To implement (e), we actually match the subcomputation's output against the result of an equality test between the satisfaction candidate and the main computation's output.)

```
minimization-computation? ≡
    [∘₁² and
        [∘₁¹ minimization-function? program]
        [∘₁² and
            [∘₁² equal?
                [∘₁¹ length subcomputations]
                [∘₁¹ successor output]]
            [∘₁⁴ [∀̄₃ [∘₄² and
                        [∘₄² equal?
                            pr₄⁰
                            [∘₄¹ program
                                [∘₄² list-ref subcomputations pr₄³]]]
                        [∘₄² and
                            [∘₄² equal?
                                [∘₄² cons-at-end pr₄¹ pr₄³]
                                [∘₄¹ inputs
                                    [∘₄² list-ref
                                        subcomputations
                                        pr₄³]]]
                            [∘₄² equal?
                                [∘₄² equal? pr₄² pr₄³]
                                [∘₄¹ output
                                    [∘₄² list-ref
                                        subcomputations
                                        pr₄³]]]]]]
                [∘₁¹ minimization-predicate program]
                inputs
                output
                output]]]
```

To count as the encoding of a valid computation, a number must satisfy one of the preceding six predicates, as well as the `launch-checks?` predicate, and

moreover it must satisfy the condition that all of its subcomputations must also encode valid computations. We deferred the enforcement of that last condition until now in order to be able to use course-of-values recursion to apply it; since all of the subcomputations of a computation are (obviously) less than the computation itself, we can find the results of applying the predicate `computation?` recursively to those smaller numbers by looking them up in the list provided by the course-of-values mechanism.

```
computation? ≡
    [Ỹ0 [∘2^2 and
            [∘2^2 or
                [∘2^1 zero-computation? pr2^0]
                [∘2^2 or
                    [∘2^1 successor-computation? pr2^0]
                    [∘2^2 or
                        [∘2^1 projection-computation? pr2^0]
                        [∘2^2 or
                            [∘2^1 composition-computation? pr2^0]
                            [∘2^2 or
                                [∘2^1 recursion-computation? pr2^0]
                                [∘2^1 minimization-computation?
                                    pr2^0]]]]]]
            [∘2^2 and
                [∘2^1 launch-checks? pr2^0]
                [∘2^2 [∧⃗1 list-ref-from-end]
                    pr2^1
                    [∘2^1 subcomputations pr2^0]]]]]
```

Note that, since we have used only primitive recursive functions in the definitions of these predicates, `computation?` itself is a primitive recursive predicate and so yields an output, either 0 or 1, for every possible input.

Exercises

12–1 Compute the encoding of the simplest possible computation: applying `zero` to no inputs, outputting 0 after no subcomputations.

12–2 Define a singulary, primitive recursive function `gir` that, given the encoding for a computation as input, outputs the greatest number that resulted from any of its subcomputations. (The name '`gir`' is an acronym for "greatest intermediate result.") If the input to `gir` does not encode a computation, `gir` should output 0.

12–3 Define a singulary function `successor-applications` that inputs the encoding for a computation and returns a tally of the number of times the

`successor` function is applied to an input in the course of that computation. If the input does not encode a computation, `successor-applications` should output 0. Prove that `successor-applications` is primitive recursive.

Chapter 13

The Universality Theorem

With the help of the `computation?` predicate, we can proceed to define a single "universal" function that can, in effect, emulate any desired partial recursive function. We'll call this universal function `apply`. It takes two inputs: the encoding for the program of the function f to be emulated, and the encoding for the list of the inputs to which f is to be applied.

For instance, we saw in section 11 that the natural number 262160 encodes our `predecessor` function. To simulate the application of `predecessor` to the input 7, form the encoding of a list with 7 as its only element (which is $\texttt{cons}(7, \texttt{nil}())$, or 128). Then $\texttt{apply}(262160, 128) = \texttt{predecessor}(7) = 6$.

Theorem 13.1 *The universal function* `apply` *is partial recursive.*

Proof: The approach to defining `apply` is basically brute-force search. We use unbounded minimization to run through the natural numbers in ascending order, checking each one to see whether it encodes a computation whose program is the program for f and whose inputs are the specified inputs. When and if we find such a computation, we recover its output. That's all there is to it!

$$
\begin{array}{l}
\texttt{apply} \equiv [\circ_2^1\ \texttt{output} \\
\qquad [\mu_2\ [\circ_3^2\ \texttt{and} \\
\qquad\qquad [\circ_3^1\ \texttt{computation?}\ \texttt{pr}_3^2] \\
\qquad\qquad [\circ_3^2\ \texttt{and} \\
\qquad\qquad\quad [\circ_3^2\ \texttt{equal?}\ \texttt{pr}_3^0\ [\circ_3^1\ \texttt{program}\ \texttt{pr}_3^2]] \\
\qquad\qquad\quad [\circ_3^2\ \texttt{equal?}\ \texttt{pr}_3^1\ [\circ_3^1\ \texttt{inputs}\ \texttt{pr}_3^2]]]]]]
\end{array}
$$

■

This definition proves that `apply` is a partial recursive function, but, since the definition uses unbounded minimization, it may not be primitive recursive. In fact, since `apply` has to simulate partial recursive functions that are not total, such as `minus`, `apply` itself cannot be total; it will be undefined whenever

the function it is emulating is undefined, since in those cases no encoding for a suitable computation will ever be found.

It may seem artificial or inconvenient to have to assemble f's inputs into a list for the benefit of `apply`. This is necessary only because `apply`, which must itself have some fixed valence, allows f to have any valence. An alternative approach would be to have a separate universal function for each possible valence:

$$\begin{aligned} \texttt{apply-nullary}(e) &= \texttt{apply}(e, \texttt{nil}()) \\ \texttt{apply-singulary}(e, x) &= \texttt{apply}(e, \texttt{cons}(x, \texttt{nil}())) \\ \texttt{apply-binary}(e, x_0, x_1) &= \texttt{apply}(e, \texttt{cons}(x_0, \texttt{cons}(x_1, \texttt{nil}()))) \\ &\ldots \end{aligned}$$

or, in our notation,

$$\begin{aligned} \texttt{apply-nullary} &\equiv [\circ_1^2\ \texttt{apply}\ \texttt{pr}_1^0\ \texttt{nil}_1] \\ \texttt{apply-singulary} &\equiv [\circ_2^2\ \texttt{apply}\ \texttt{pr}_2^0\ [\circ_2^2\ \texttt{cons}\ \texttt{pr}_2^1\ \texttt{nil}_2]] \\ \texttt{apply-binary} &\equiv [\circ_3^2\ \texttt{apply}\ \texttt{pr}_3^0\ [\circ_3^2\ \texttt{cons}\ \texttt{pr}_3^1\ [\circ_3^2\ \texttt{cons}\ \texttt{pr}_3^2\ \texttt{nil}_3]]] \end{aligned}$$

and so on.

Exercises

13–1 Could one make the encoding e_{apply} for the `apply` function as the first input to the `apply` function itself? What kind of value would the second input to `apply` have to be in this situation? What kind of value would be the result be?

13–2 Can `apply` be undefined when its first input is the encoding of a total function? Justify your answer.

13–3 Define a partial recursive function `application-sum` that takes two inputs, the first of which is the encoding e for a singulary function f and the second a natural number n, and outputs the sum of the values obtained by applying f to every natural number less than or equal to n. We should have $\texttt{application-sum}(e, n)\uparrow$ whenever $f(k)\uparrow$ for any natural number k less than or equal to n.

Chapter 14

The Halting Predicate

It would be quite useful, as a kind of supplement to `apply`, to be able to work with a total predicate `defined?` that would take as inputs the encoding for a partial recursive function f and a list of inputs $x_0, \ldots, x_{n-1}$ to f and determine whether $f(x_0, \ldots, x_{n-1})\downarrow$. Given `apply`, it is tempting to try to define such a predicate as $[\circ_2^1$ `truish?` $[\circ_2^1$ `one`$_1$ `apply`$]]$. Unfortunately, this definition fails; the function that it describes correctly returns 1 whenever $f(x_0, \ldots, x_{n-1})\downarrow$, but when $f(x_0, \ldots, x_{n-1})\uparrow$ it is itself undefined instead of returning 0.

Indeed, it turns out that `defined?` is not a partial recursive function at all!

Theorem 14.1 *The predicate* `defined?` *is not partial recursive.*

Proof: The proof is by contradiction, using a diagonal argument. Suppose that `defined?`, in addition to being total and a predicate, were partial recursive. Then we would be able to use it to define another function, `self-blocker`, as follows:

$$\begin{aligned} \texttt{self-undefined?} &\equiv [\circ_1^1 \texttt{ not } [\circ_1^2 \texttt{ defined? } \texttt{pr}_1^0 \ [\circ_1^2 \texttt{ cons } \texttt{pr}_1^0 \texttt{ nil}_1]]] \\ \texttt{self-blocker} &\equiv [\mu_1 \ [\circ_2^1 \texttt{ self-undefined? } \texttt{pr}_2^0]] \end{aligned}$$

Given the encoding e of a singulary partial recursive function f, the predicate `self-undefined?` would compute and output $\texttt{not}(\texttt{defined?}(e, \texttt{cons}(e, \texttt{nil}())))$, which (by definition) would be 0 if $f(e)\downarrow$ and 1 if $f(e)\uparrow$. So, given e as its first input and any natural number t as its second input, the predicate to which the unbounded minimization is applied would ignore t completely, again outputting 0 if $f(e)\downarrow$ and 1 if $f(e)\uparrow$. So the function defined by unbounded minimization, namely `self-blocker`, would be undefined for any input e such that $f(e)\downarrow$ (because no choice of t would make $[\circ_2^1$ `self-undefined?` $\texttt{pr}_2^0](e, t)$ positive), and would output 0 for any input e such that $f(e)\uparrow$.

Now, if `defined?` were partial recursive, then `self-blocker` would also be partial recursive, and so there would be a natural number d that encoded it. Then, as we have seen, `self-blocker`(d) would be defined (and would be 0) if,

and only if, `self-blocker`$(d)\uparrow$. This is a contradiction. ■

Exercises

14–1 Is `self-blocker` (a) a partial recursive function that cannot be defined in terms of `defined?`; (b) a function that is not partial recursive; or (c) not a function at all? Justify your answer.

14–2 Let `total?` be a total singulary function that takes as input the encoding for a partial recursive function f and outputs 1 if f is total and 0 otherwise. Prove that `total?` is not partial recursive.

Chapter 15

Recursive and Recursively Enumerable Sets

A function is said to be *recursive* if, and only if, it is both partial recursive and total. This is not quite the same as being primitive recursive, since it is possible, even when f is total, that every definition of f includes at least one use of unbounded minimization. In the computations for such functions, the minimization operation always succeeds eventually, but there is no fixed upper bound on the number of satisfaction candidates that must be tried. (It is true that all primitive recursive functions are recursive, but the converse is not true.)

A set S of natural numbers is said to be *recursive* if, and only if, there is some singulary, recursive function p such that the inputs that satisfy p are exactly the members of S:

$$p(x) > 0 \iff x \in S.$$

Since recursive functions are total, this condition implies that $p(x) = 0 \iff x \notin S$. The predicate p that meets this condition (so that $p(x) = 1 \iff x \in S$) is sometimes called the *characteristic function* of S.

We have already encountered a few such predicates. For instance, `zero?` is the characteristic function of $\{0\}$, and `even?` is the characteristic function of $\{0, 2, 4, \ldots\}$, so both of these sets are recursive. It is not difficult to see that every finite set of natural numbers is recursive; for instance, the set $\{0, 2, 5\}$ has the primitive recursive characteristic function

```
[∘₁² or
    [∘₁² equal? pr₁⁰ zero₁]
    [∘₁² or
        [∘₁² equal? pr₁⁰ two₁]
        [∘₁² equal? pr₁⁰ five₁]]]
```

and this pattern can be extended to any finite set.

Theorem 15.1 *The set of encodings of valid computations is recursive.*

Proof: The `computation?` predicate defined at the end of § 12 is singulary and primitive recursive (and therefore recursive), and it is the characteristic function for the set of encodings of valid computations. ■

However, not all sets are recursive. Consider, for instance, the set K of encodings for pairs (i, j) such that i encodes a partial recursive function f, j encodes a list of inputs $x_0, \ldots, x_{n-1}$ for that function, and $f(x_0, \ldots, x_{n-1})\downarrow$.

Theorem 15.2 *K is not recursive.*

Proof: If K were recursive, then its characteristic function (let's call it `applicable-pair?`) would also be recursive, and hence partial recursive. But then the halting predicate `defined?` from § 14 would be partial recursive, since we would be able to define it thus:

$$\texttt{defined?} \equiv [\circ_2^1\ \texttt{applicable-pair? cons}]$$

Since `defined?` is not partial recursive, neither is `applicable-pair?`, and hence K is not a recursive set. ■

Theorem 15.3 *The union $S \cup T$ and the intersection $S \cap T$ of any recursive sets S and T are recursive.*

Proof: Let s and t be the characteristic functions for S and T, respectively. Then $[\circ_1^2\ \texttt{or}\ s\ t]$ is a characteristic function for $S \cup T$, and $[\circ_1^2\ \texttt{and}\ s\ t]$ is a characteristic function for $S \cap T$. ■

Theorem 15.4 *The complement $\tilde{S}$ of any recursive set S is recursive.*

Proof: Let s be the characteristic function for S. Then $[\circ_1^1\ \texttt{not}\ s]$ is a characteristic function for $\tilde{S}$. ■

A set S of natural numbers is said to be *recursively enumerable* if, and only if, there is some singulary partial recursive function f such that the inputs for which f is defined are exactly the members of S:

$$f(x)\downarrow \iff x \in S.$$

Let's call f the *acceptance function* for S.

Theorem 15.5 *Every recursive set is recursively enumerable.*

Proof: Any recursive set S has a partial recursive characteristic function s, so that the function $[\mu_1\ [\circ_2^1\ s\ \texttt{pr}_2^0]]$ will also be partial recursive. This function is an acceptance function for S. ■

Theorem 15.6 *K is recursively enumerable.*

Proof: The singulary partial recursive function $[\circ_1^2$ `apply car cdr`$]$ is an acceptance function for K. ■

Thus recursive sets form a proper subset of recursively enumerable sets.

Theorem 15.7 *The intersection $S \cap T$ of any recursively enumerable sets S and T is recursively enumerable.*

Proof: Let s and t be acceptance functions for S and T, respectively. Then $[\circ_1^2$ `add` s $t]$ is an acceptance function for $S \cap T$. ■

Theorem 15.8 *The union $S \cup T$ of any recursively enumerable sets S and T is recursively enumerable.*

The proof of this theorem, though instructive, is a little subtle. We need to return to the notion of computation and, in effect, carry through parallel searches for computations involving members of S and members of T.

Proof: Since S and T are recursively enumerable, they must have singulary partial recursive acceptance functions, say s and t respectively. Let `s-encoded` and `t-encoded` be nullary functions with the encoding for s and the encoding for t as their respective outputs. Then we can define a partial recursive predicate `stx?` that takes two inputs, a natural number x and the encoding c for a computation, and asks whether c is a valid computation that has either s or t as its program and x as its input:

```
stx? ≡ [∘₂² and
          [∘₂¹ computation? pr₂¹]
          [∘₂² and
              [∘₂² equal? [∘₂¹ inputs pr₂¹] [∘₂² cons pr₂⁰ nil₂]]
              [∘₂² or
                  [∘₂² equal? [∘₂¹ program pr₂¹] s-encoded₂]
                  [∘₂² equal? [∘₂¹ program pr₂¹] t-encoded₂]]]]
```

Whenever $s(x)\downarrow$, there is a computation that has the encoding of s as its program, $\texttt{cons}(x, \texttt{nil}())$ as its list of inputs, and $s(x)$ as its output; if c_s is the encoding for such a computation, then $\texttt{stx?}(x, c_s)$ is 1. Similarly, whenever $t(x)\downarrow$, there is a computation, encoded by, say c_t, such that $\texttt{stx?}(x, c_t)$ is 1. But if $s(x)\uparrow$ and $t(x)\uparrow$, then $\texttt{stx?}(x, c) = 0$ for every natural number c, because there is no computation that meets all the requisite conditions.

The function $[\mu_1$ `stx?`$]$, then, takes one input, x, and searches for the least natural number c such that $\texttt{stx?}(x, c) > 0$. So $[\mu_1$ `stx?`$](x)\downarrow$ if, and only if, $s(x)\downarrow$ or $t(x)\downarrow$ (or both). Therefore, $[\mu_1$ `stx?`$]$ is an acceptance function for $S \cup T$. So $S \cup T$ is recursively enumerable. ■

The situation with respect to complements of recursively enumerable sets is even more complicated. If a set S is recursive, then S is recursively enumerable (because all recursive sets are), and its complement $\tilde{S}$ is also recursively enumerable (because the complement of a recursive set is recursive, and all recursive sets are recursively enumerable). It turns out that the converse is also true:

Theorem 15.9 *If a set S and its complement $\tilde{S}$ are both recursively enumerable, then S is recursive.*

Proof: Suppose that both S and $\tilde{S}$ are recursively enumerable. Then they must have singulary partial recursive acceptance functions, say s and $\tilde{s}$. From these, we can define the following related functions:

$$\begin{aligned}
\texttt{yes}(x) &= \begin{cases} 1 & \text{if } s(x)\downarrow, \\ \uparrow & \text{otherwise,} \end{cases} \\
\texttt{no}(x) &= \begin{cases} 0 & \text{if } \tilde{s}(x)\downarrow, \\ \uparrow & \text{otherwise,} \end{cases} \\
\texttt{yes} &\equiv [\circ_1^1\ \texttt{one}_1\ s], \\
\texttt{no} &\equiv [\circ_1^1\ \texttt{zero}_1\ \tilde{s}].
\end{aligned}$$

Thus `yes` will also be an acceptance function for S and `no` an acceptance function for $\tilde{S}$. Let `yes-encoded` and `no-encoded` be nullary functions that output the encodings for `yes` and `no`, respectively.

Now, since S and $\tilde{S}$ are complements, every natural number x is a member of one or the other. Consequently, for every natural number x, either `yes`(x) or `no`(x) is defined, so that there is a valid computation c in which either `yes` or `no` is applied to x. So define a function `yes-or-no?` as follows:

```
yes-or-no? ≡ [∘₂² and
                [∘₂¹ computation? pr₂¹]
                [∘₂² and
                    [∘₂² equal? [∘₂¹ inputs pr₂¹] [∘₂² cons pr₂⁰ nil₂]]
                    [∘₂² or
                        [∘₂² equal? [∘₂¹ program pr₂¹] yes-encoded₂]
                        [∘₂² equal? [∘₂¹ program pr₂¹] no-encoded₂]]]]
```

No matter what x is, there will always be some encoding c for a computation such that `yes-or-no?`$(x, c) = 1$. The output of that computation will be 1 (from `yes`) if $x \in S$, 0 (from `no`) if $x \in \tilde{S}$. So $[\circ_1^1$ `output` $[\mu_1$ `yes-or-no?`$]]$ is a singulary, total, partial recursive characteristic function for S. So S is recursive, as required. ∎

From the preceding theorems, it follows that there are some sets that are not even recursively enumerable. In particular:

Theorem 15.10 *$\tilde{K}$ is not recursively enumerable.*

Proof: If $\tilde{K}$ were recursively enumerable, then, since K is recursively enumerable (Theorem 15.6), K would be recursive, by Theorem 15.9. But this would contradict Theorem 15.2. ■

There are several alternative ways to characterize recursively enumerable sets. For instance, every recursively enumerable set, except the empty set, is the set of outputs of some primitive recursive function:

Theorem 15.11 *If a set S is recursively enumerable and not empty, then there is a singulary primitive recursive function f such that $S = \{f(n) \mid n \in \mathbb{N}\}$.*

Proof. Since S is not empty, it has some least member s. Let `s-constant` be a nullary function that outputs s. Since S is recursively enumerable, there is an acceptance function g for it. Let `g-encoded` be a nullary function that outputs the encoding for g. Then define

```
g-inputs ≡ [¿1 [∘1^2 and
                   computation?
                   [∘1^2 equal? program g-encoded1]]
               [∘1^1 car inputs]
               s-constant1]
```

Given a natural number x, `g-inputs` checks whether it encodes a valid computation in which the program encodes g. If so, it outputs the input n to that program; since n encodes a valid computation, $g(n)\downarrow$, so $n \in S$. If x does not encode a valid computation, or encodes a computation for some function other than g, `g-inputs` outputs s, which is by definition a member of S. Thus every output of `g-inputs` is a member of S. Conversely, for every member n of S, $g(n)\downarrow$, so there must be a computation with a program that encodes g and a list of inputs whose only element is n; when `g-inputs` is given the encoding for this computation, it returns n. So $S = \{\texttt{g-inputs}(n) \mid n \in \mathbb{N}\}$, so that the required function f in the statement of the theorem is the singulary, primitive recursive function `g-inputs`. ■

The converse of the preceding theorem is also true, and indeed we can even weaken the condition, so that f need only be partial recursive, not primitive recursive:

Theorem 15.12 *For any partial recursive function f, the set $\{f(n) \mid n \in \mathbb{N}\}$ is recursively enumerable.*

Let `f-encoded` be a nullary function that outputs the encoding for f. Then define

```
f-output-match? ≡ [∘₂² and
                       [∘₂¹ computation? pr₂¹]
                       [∘₂² and
                           [∘₂² equal? [∘₂¹ program pr₂¹] f-encoded₂]
                           [∘₂² equal? pr₂⁰ [∘₂¹ output pr₂¹]]]]
```

The `f-output-match?` predicate takes two inputs, n and c, and determines whether c encodes a computation that has a program that encodes f and an output that is equal to n.

Now consider $[\mu_1$ `f-output-match?`$]$. Given a natural number n, this function searches for a natural number c such that `f-output-match?`$(n, c) > 0$—that is, for the encoding of a computation in which f outputs n. If $[\mu_1$ `f-output-match?`$]$ finds such a c, it outputs it; if no such computation exists, $[\mu_1$ `f-output-match?`$](n)\uparrow$. Thus $[\mu_1$ `f-output-match?`$]$ is an acceptance function for S. Hence S is recursively enumerable.

Exercises

15–1 Prove that the set $\{0, 1, 4, 9, 16, \ldots\}$ of squares of natural numbers is recursive.

15–2 Prove that the set of squares of members of the set K (as defined in the explanation preceding Theorem 15.2) is recursively enumerable.

15–3 Is the set of all natural numbers recursively enumerable? Is it recursive? Justify your answers.

Chapter 16

Turing Machines and Their Configurations

In automata theory, a different model is more commonly used to analyze functions and computations. A *Turing machine* is a simple, idealized computational device, consisting of

- a *store*, which is a memory of finite capacity, capable of holding any of a specified finite repertoire of values, called the *states* of the Turing machine;
- a *read-write head*, which can interoperate with an external storage device of unlimited capacity, reading one symbol at a time from that device or writing one symbol at a time to it; and
- a *control*, directing the activity of the store and the read-write head according to a fixed program.

The external storage device for a Turing machine is a *tape*, an infinite sequence of storage cells, in one-to-one correspondence with the natural numbers. Each cell is capable of storing one symbol from a finite *tape alphabet*, which is the set of symbols that the Turing machine is capable of reading or writing. The tape alphabet includes a blank symbol, and storage cells that are not otherwise initialized contain this symbol by default.

The designer of a Turing machine designates a proper subset of the tape alphabet, not including the blank, as the *input alphabet*. The input to a Turing machine is a finite string s of symbols from the input alphabet. To prepare a tape for a Turing machine. we store the first symbol of the input s in cell 0, the second in cell 1, and so on in sequence. Cells corresponding to natural numbers greater than or equal to $|s|$ are left blank. Since a Turing machine can write only one symbol at a time onto its tape, all but a finite number of cells remain blank at any stage in a computation.

When a tape is loaded into a Turing machine, the control initializes the store to a designated start state q_0 and positions its read-write head over cell 0. It then repeatedly executes the following steps:

- It reads the tape symbol currently under the read-write head.
- Using that tape symbol and the contents of the store, it deterministically computes three values: a new state, a new tape symbol, and one of the two arbitrary values L and R.
- It updates the store to contain the new state.
- It writes the new tape symbol into the cell currently under the read-write head.
- If the third computed value is R, the control moves the read-write head one cell to the right along the tape (to the cell that corresponds to the next greater natural number). If the third computed value is L, the control moves the read-write head one cell to the left if possible, to the cell that corresponds to the next lesser natural number; but if the read-write head is already over cell 0, the control does not move it at all, but leaves it on cell 0.

This cycle is repeated until the control finds that the store contains one of two designated *stopping states*, q_{accept} or q_{reject}. The control recognizes these as commands to halt the processing of the tape. If, at the time it halts, the store contains q_{accept}, the Turing machine *accepts* the input string that was on its tape initially. If, instead, the store contains q_{reject}, the Turing machine *rejects* that string.

It may happen that, on some or all possible input strings, a particular Turing machine continues to run through its processing cycle forever, without ever storing q_{accept} or q_{reject}, so that the Turing machine neither accepts nor rejects its input.

In effect, then, a Turing machine computes a partial function f that has one input, a string s of symbols from its input alphabet, and one output, a Boolean value. To compute $f(s)$, we load s onto a tape, insert the tape into the Turing machine, and inspect the state of the store if and when the machine halts.

For any Turing machine M, the *language* of M, $L(M)$, is the set of strings that M accepts. M is said to *recognize* this set. If M rejects every string that it does not accept (or, equivalently, if M eventually halts no matter what input string it is given), then M is said to *decide* $L(M)$.

A set of strings is *Turing-recognizable* if there is some Turing machine that recognizes it, and *decidable* if there is some Turing machine that decides it.

Modelling Turing Machines

Using the data structures developed in § 7, we can model Turing machines and their configurations within $\mathbb{N}$. To describe the workings of Turing machines using partial recursive functions, we shall first develop a system for encoding Turing machines as natural numbers.

Formally, we'll define (deterministic, single-tape) Turing machines as septuples of the form $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q is a finite set of states, Σ is a finite alphabet not containing the blank symbol, Γ is a finite alphabet containing the blank symbol and every symbol in Σ (as well as, possibly, others), δ is a transition function with domain $(Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma$ and range $Q \times \Gamma \times \{\text{L}, \text{R}\}$, $q_0 \in Q$, $q_{\text{accept}} \in Q$, $q_{\text{reject}} \in Q$, and $q_{\text{reject}} \neq q_{\text{accept}}$.

Even in this highly abstract and mathematized form, Turing machines are clearly quite complicated structures. Nevertheless, finding a correspondence that maps each Turing machine to a different natural number is just a matter of perseverance. We already have all of the tools we need.

Let's begin with the states. Each state can be given a serial number, starting with 0 and counting upwards without skipping. The last serial number used will be $n - 1$, where n is the number of states. Every Turing machine has at least two states, since $q_{\text{reject}} \neq q_{\text{accept}}$. Without loss of generality, we can stipulate that q_{reject} will always be given the largest serial number and q_{accept} the next largest one.

If the serial numbers are assigned in this way, we can represent the set of states in our encoding simply by the integer n. Any number greater than or equal to 2 is a valid encoding for a set of states.

It will be convenient, then, to define several functions relating to the states of a Turing machine: `TM-states`, which inputs the encoding for a Turing machine and returns the number of its states; `TM-accept-state` and `TM-reject-state`, which input the encoding for a Turing machine and return, respectively, the serial numbers of its accept and reject state; `TM-states?`, a predicate that determines whether a given natural number is a valid number of states for a Turing machine; and `TM-stopping-state?`, a predicate that inputs the encoding for a Turing machine and the serial number of one of its states and determines whether the state is either of the machine's stopping states (i.e., the accept and reject states).

```
TM-states ≡ [∘₁² list-ref pr₁⁰ zero₁]
TM-accept-state ≡ [∘₁² subtract TM-states two₁]
TM-reject-state ≡ [∘₁² subtract TM-states one₁]
TM-states? ≡ [∘₁² greater-or-equal? pr₁⁰ two₁]
TM-stopping-state? ≡ [∘₂² greater-or-equal?
                        pr₂¹
                        [∘₂² subtract [∘₂¹ TM-states pr₂⁰] two₂]]
```

Next, the alphabets. As in § 7, we can give each symbol in the tape alphabet a different positive integer as its serial number. Without loss of generality, we can use the serial number 1 for the blank and give the immediately following serial numbers to members of the input alphabet, saving the higher serial numbers for non-blank symbols that are members of the tape alphabet but not of the input alphabet. In our encodings of Turing machines, then, we can represent the input alphabet and tape alphabets simply as their sizes, constraining the tape alphabet to be at least 1 and at least as great as the input alphabet.

The `TM-input-alphabet` function inputs the encoding for a Turing machine and outputs the size of its input alphabet, and the `TM-tape-alphabet` function similarly recovers the size of a given Turing machine's tape alphabet. The `TM-input-alphabet?` predicate determines whether a given natural number is a valid size for an input alphabet; since the input alphabet can be of any size, even 0, this predicate is simply $\texttt{one}_1$. Lastly, the `TM-tape-alphabet?` predicate inputs two natural numbers, interprets the first as the size of a Turing machine's input alphabet, and determines whether the second is a valid size for that same Turing machine's tape alphabet.

$\texttt{TM-input-alphabet} \equiv [\circ_1^2\ \texttt{list-ref}\ \texttt{pr}_1^0\ \texttt{one}_1]$,
$\texttt{TM-tape-alphabet} \equiv [\circ_1^2\ \texttt{list-ref}\ \texttt{pr}_1^0\ \texttt{two}_1]$,
$\texttt{TM-input-alphabet?} \equiv \texttt{one}_1$,
$\texttt{TM-tape-alphabet?} \equiv [\circ_2^2\ \texttt{and}\ [\circ_2^1\ \texttt{positive?}\ \texttt{pr}_2^1]\ \texttt{less-or-equal?}]$

Now let's consider the transition function δ. The values of the transition function are triples of the form (q, a, d), where $q \in Q$, $a \in \Gamma$, and $d \in \{\mathrm{L}, \mathrm{R}\}$. We can encode such a triple as a three-element list in which the first element is $\bar{q}$, the second $\bar{a}$, and the third is 0 if d is L and 1 if d is R. (Here $\bar{q}$ is the encoding for the state q and $\bar{a}$ the encoding for the symbol a, considered as an element of the tape alphabet.)

The `encode-TM-action` function constructs and outputs such a triple from the encodings for its components, and the next three functions defined below extract the respective components from such a triple. Lastly, `TM-action?` predicate inputs three natural numbers, interprets the first two as the number of states in a Turing machine and the size of its tape alphabet, and determines whether the third could be used to represent an action of that same Turing machine.

$\texttt{encode-TM-action} \equiv [\circ_3^2\ \texttt{cons}\ \texttt{pr}_3^0\ [\circ_3^2\ \texttt{cons}\ \texttt{pr}_3^1\ [\circ_3^2\ \texttt{cons}\ \texttt{pr}_3^2\ \texttt{nil}_3]]]$
$\texttt{TM-action-target-state} \equiv \texttt{car}$
$\texttt{TM-action-symbol-to-print} \equiv [\circ_1^1\ \texttt{car}\ \texttt{cdr}]$
$\texttt{TM-action-direction} \equiv [\circ_1^1\ \texttt{car}\ [\circ_1^1\ \texttt{cdr}\ \texttt{cdr}]]$

```
TM-action? ≡ [∘₃² and
                [∘₃² equal? [∘₃¹ length pr₃²] three₃]
                [∘₃² and
                   [∘₃² less? [∘₂¹ car pr₃²] pr₃⁰]]
                   [∘₃² and
                      [∘₃² [∘₂² and
                             [∘₂¹ positive? pr₂⁰]
                             less-or-equal?]
                         [∘₃¹ car [∘₃¹ cdr pr₃²]]
                         pr₃¹]
                      [∘₃² less-or-equal?
                         [∘₃¹ car [∘₃¹ cdr [∘₃¹ cdr pr₃²]]]
                         one₃]]]]
```

The transition function for a Turing machine must specify such an action for each combination of a non-stopping state and a symbol from the tape alphabet. We can therefore represent a transition function as the encoding for a list of length $|Q - \{q_{\text{accept}}, q_{\text{reject}}\}| \cdot |\Gamma|$, in which the elements are Turing machine actions. In this list, we place the encoding for $\delta(q, a)$ at zero-based position $\bar{q} \cdot |\Gamma| + \texttt{predecessor}(\bar{a})$. (The application of the `predecessor` function compensates for the fact that the serial numbers for tape symbols begin with 1.)

In this implementation, the `TM-transition-function` function inputs the encoding for a Turing machine and outputs the encoding for its transition function; the `TM-transition-function?` predicate inputs three natural numbers, interprets the first as the number of states in a Turing machine and the second as the size of its tape alphabet, and determines whether the third would be a valid transition function for that Turing machine; and the `TM-transition-lookup` function inputs a Turing machine, a non-stopping state of that Turing machine, and a symbol from that Turing machine's tape alphabet, and outputs the encoding for the action that that Turing machine will perform as its next transition.

```
TM-transition-function ≡ [∘₁² list-ref pr₁⁰ three₁]
TM-transition-function? ≡ [∘₃² and
                             [∘₃² equal?
                                [∘₃¹ length pr₃²]
                                [∘₃² multiply
                                   pr₃¹
                                   [∘₃² subtract pr₃⁰ two₂]]]
                             [⃗Λ₂ TM-action?]]
```

$$\begin{aligned}\texttt{TM-transition-lookup} \equiv\ & [\circ_3^2\ \texttt{list-ref}\\ & \quad [\circ_3^1\ \texttt{TM-transition-function}\ \texttt{pr}_3^0]\\ & \quad [\circ_3^2\ \texttt{add}\\ & \qquad [\circ_3^2\ \texttt{multiply}\\ & \qquad\quad \texttt{pr}_3^1\\ & \qquad\quad [\circ_3^1\ \texttt{TM-tape-alphabet}\ \texttt{pr}_3^0]]\\ & \qquad [\circ_3^1\ \texttt{predecessor}\ \texttt{pr}_3^2]]]\end{aligned}$$

Finally, we must indicate the state in which the Turing machine is to be started. We cannot simply stipulate that the start state is the one that receives the serial number 0, because in some Turing machines the start state is the same state as q_{reject} or q_{accept}. So we shall add the start state's serial number explicitly to our encoding.

The `TM-start-state` function inputs the encoding for a Turing machine and outputs the encoding for its start state. The `TM-start-state?` predicate inputs two natural numbers, interprets the first as the number of states of a Turing machine, and determines whether the second input could be the encoding for the start state of that Turing machine.

$$\begin{aligned}\texttt{TM-start-state} &\equiv [\circ_1^2\ \texttt{list-ref}\ \texttt{pr}_1^0\ \texttt{four}_1]\\ \texttt{TM-start-state?} &\equiv \texttt{greater?}\end{aligned}$$

Thus we shall encode Turing machines as five-element lists $(n, j, k, \bar{\delta}, i)$, where n is the number of states in the machine, j the number of symbols in the input alphabet, k the number of symbols in the tape alphabet, $\bar{\delta}$ the encoding of the transition function, and i the serial number of the start state.

$$\begin{aligned}\texttt{encode-TM} \equiv\ & [\circ_5^2\ \texttt{cons}\\ & \quad \texttt{pr}_5^0\\ & \quad [\circ_5^2\ \texttt{cons}\\ & \qquad \texttt{pr}_5^1\\ & \qquad [\circ_5^2\ \texttt{cons}\\ & \qquad\quad \texttt{pr}_5^2\\ & \qquad\quad [\circ_5^2\ \texttt{cons}\ \texttt{pr}_5^3\ [\circ_5^2\ \texttt{cons}\ \texttt{pr}_5^4\ \texttt{nil}_5]]]]]\end{aligned}$$

```
TM? ≡ [∘₁² and
          [∘₁² equal? length five₁]
          [∘₁² and
              [∘₁¹ TM-states? [∘₁² list-ref pr₁⁰ zero₁]]
              [∘₁² and
                  [∘₁¹ TM-input-alphabet? [∘₁² list-ref pr₁⁰ one₁]]
                  [∘₁² and
                      [∘₁² TM-tape-alphabet?
                          [∘₁² list-ref pr₁⁰ one₁]
                          [∘₁² list-ref pr₁⁰ two₁]]
                      [∘₁² and
                          [∘₁² TM-transition-function?
                              [∘₁² list-ref pr₁⁰ zero₁]
                              [∘₁² list-ref pr₁⁰ two₁]
                              [∘₁² list-ref pr₁⁰ three₁]]
                          [∘₁² TM-start-state?
                              [∘₁² list-ref pr₁⁰ zero₁]
                              [∘₁² list-ref pr₁⁰ four₁]]]]]]]
```

For example, consider a very simple Turing machine on the input alphabet {a}. When started, it inspects the first cell of the tape and transitions to state q_{accept} if it finds a blank there, or to state q_{reject} if it finds an a, printing a blank and moving the read-write head rightwards in either case. Thus this Turing machine decides the language $\{\varepsilon\}$ (that is, it accepts only the null string).

The three states of this machine will receive the serial numbers 0, 1, and 2—respectively, the start state, q_{accept}, and q_{reject}. The input alphabet contains one symbol and the tape alphabet contains two. The transition function is completely defined by the equations

$$\begin{aligned} \delta(q_0, \text{␣}) &= (q_{\text{accept}}, \text{␣}, \text{R}), \\ \delta(q_0, \text{a}) &= (q_{\text{reject}}, \text{␣}, \text{R}). \end{aligned}$$

(where '␣' is the blank symbol). The encoding for the triple on the right-hand side of the first of these equations is $\texttt{cons}(1, \texttt{cons}(0, \texttt{cons}(1, \texttt{nil}())))$, since q_{accept}, ␣, and R are all encoded as 1, and we take the predecessor of the encoding for the symbol to get the middle element of the triple. This value works out to be 22. Similarly, the encoding for the triple on the right-hand side of the second equation is $\texttt{cons}(2, \texttt{cons}(0, \texttt{cons}(1, \texttt{nil}())))$, which is 44. So the transition function δ is represented by the list $(22, 44)$, which is encoded as cons(22, cons(44, nil())), or 147573952589680607232.

Assembling the pieces, then, we find that the encoding for the Turing machine is equal to the encoding for the list $(3, 1, 2, 147573952589680607232, 0)$, which works out to be $3 \cdot 2^{147573952589680607241} + 296$. This may seem like a largish number for such a simple device. The magnitudes of the encodings,

however, are of little or no significance. What is important is that every Turing machine has a unique representation and that we can extract any desired information about the Turing machine from that representation.

Exercises

16–1 Define a predicate `possibly-restarting?` that takes the encoding for a Turing machine M as its input and outputs 1 if any of the transitions in M return the machine to its start state, 0 if none do.

16–2 Define a predicate `rightwards-only?` that takes the encodings for a Turing machine M as its input and outputs 1 if all of the transitions in M cause the tape head to move to the right, 0 otherwise.

16–3 Define a function `add-new-start-state` that takes the encoding for a Turing machine M as its input and returns the encoding for a Turing machine M' similar to M, but with one additional state. The new state should be the start state of M' and should have the same out-transitions as the start state of M. All of the states of M should also be states of M' and their out-transitions should be unchanged.

Chapter 17

Encoding Configurations

To represent a configuration of a Turing machine, we can use a triple in which the first element is the serial number of the machine's current state, the second is the number of the tape cell that is under the read-write head, tape, and the third is a string of symbols from the tape alphabet. The string should record the contents of each cell that either contains a non-blank symbol or has been (or is now) under the read-write head. Since there are only a finite number of such cells when the Turing machine is started, and the number of such cells cannot increase by more than 1 in any single execution cycle, the string is always of finite length, bounded by the length of the input or the number of cycles that have occurred so far, whichever is greater.

Again, we'll define a constructor for the data type, a selector for each component, and a type predicate:

$$\texttt{encode-configuration} \equiv [\circ_3^2\ \texttt{cons}\ \texttt{pr}_3^0\ [\circ_3^2\ \texttt{cons}\ \texttt{pr}_3^1\ [\circ_3^2\ \texttt{cons}\ \texttt{pr}_3^2\ \texttt{nil}_3]]]]$$

$$\texttt{current-state} \equiv [\circ_1^2\ \texttt{list-ref}\ \texttt{pr}_1^0\ \texttt{zero}_1]$$

$$\texttt{head-position} \equiv [\circ_1^2\ \texttt{list-ref}\ \texttt{pr}_1^0\ \texttt{one}_1]$$

$$\texttt{tape-contents} \equiv [\circ_1^2\ \texttt{list-ref}\ \texttt{pr}_1^0\ \texttt{two}_1]$$

```
configuration? ≡ [∘₂² and
                    [∘₂² equal? [∘₂¹ length pr₂¹] three₂]
                    [∘₂² and
                        [∘₂² greater?
                            [∘₂¹ TM-states pr₂⁰]
                            [∘₂¹ current-state pr₂¹]]
                        [∘₂² less?
                            [∘₂¹ head-position pr₂¹]
                            [∘₂¹ string-length
                                [∘₂¹ TM-tape-alphabet pr₂⁰]
                                [∘₂¹ tape-contents pr₂¹]]]]]
```

As an example of the use of the constructor, let's write a function that constructs and returns the encoding for the initial configuration of a Turing machine M that is about to process the input string s, given the encodings $\bar{M}$ and $\bar{s}$ of M and s respectively. The first input to `make-configuration` should be the start state of M, which is `TM-start-state`$(\bar{M})$. The second should be the position of the read-write head, which by definition is always 0 in the initial configuration. And the third input should simply be the input string, s, but expressed in the tape-alphabet encoding rather than the input-alphabet encoding. (An exception arises, however, when $s = \varepsilon$; in that case, the third input to `make-configuration` should be 1, the encoding for the string consisting of a single blank, rather than 0, the encoding for the null string. This exception is needed to establish the invariant that the initial position of the read-write head is strictly less than the length of the string representing the tape contents.)

The `reencode-input` function converts the string s (provided that it is not ε) from an encoding based on an m-symbol input alphabet to an encoding based on an n-symbol tape alphabet, allowing also for the presence of the blank symbol at the beginning of the tape alphabet (that is, with serial number 1). It presupposes that the symbols of the input alphabet have the same relative order within the tape alphabet and precede all of the other non-blank symbols of the tape alphabet.

$$\texttt{reencode-input}(m, n, \bar{s}) = \sum_{i=0}^{k-1} \left((\texttt{string-ref}(m, \bar{s}, i) + 1) \cdot n^i \right),$$

where $k = \texttt{string-length}(m, \bar{s})$. This function is primitive recursive:

```
reencode-input ≡ [∘₃⁴ [Σ̄₃ [∘₄² multiply
                            [∘₄¹ successor
                                [∘₄³ string-ref pr₄⁰ pr₄² pr₄³]]
                            [∘₄² raise-to-power pr₄¹ pr₄³]]]
                  pr₃⁰
                  pr₃¹
                  pr₃²
                  [∘₃¹ predecessor [∘₃² string-length pr₃⁰ pr₃²]]]
```

So we can define the `boot` function that inputs the encoding for a Turing machine and the encoding for an input string (in that Turing machine's input alphabet) and outputs the configuration of the Turing machine when the tape is loaded, just before the first execution cycle:

```
boot ≡ [∘₂³ make-configuration
            [∘₂¹ TM-start-state pr₂⁰]
            zero₂
            [¿₂ [∘₂¹ zero? pr₂¹]
                one₂
                [∘₂³ reencode-input
                    [∘₂¹ TM-input-alphabet pr₂⁰]
                    [∘₂¹ TM-tape-alphabet pr₂⁰]
                    pr₂¹]]]
```

Note that the possibility that the input string is ε is handled as a special case.

Given the encoding $\bar{M}$ for a Turing machine M and the encoding $\bar{c}$ for its current configuration, the `step` function computes its next configuration, as follows:

If the current state of the machine is q_{accept} or q_{reject}, $\texttt{step}(\bar{M}, \bar{c}) = \bar{c}$, since M halts when it enters either of these states.

Otherwise, `step` recovers the encoding for the symbol currently under the read-write head, using $\texttt{head-position}(\bar{c})$ as an index into $\texttt{tape-contents}(\bar{c})$. By transmitting that symbol, along with the Turing machine and its current state, to `TM-transition-lookup`, `step` determines the new state, the symbol to be printed onto the tape, and the direction in which the read-write head should move. It uses `string-update` to compute the revised tape contents; it computes the new head position; and, finally, it uses `make-configuration` to assemble the new configuration.

Before defining `step`, it will be helpful to define several intermediate functions, all of which input the encoding for a Turing machine and the encoding for a configuration of that machine. The first of these helper functions, `symbol-under-head`, determines which of the Turing machine's tape symbols is in the cell on which the Turing machine's read-write head is positioned.

$$\begin{aligned}\texttt{symbol-under-head} \equiv [\circ_2^3\ &\texttt{string-ref}\\ &[\circ_2^1\ \texttt{TM-tape-alphabet}\ \texttt{pr}_2^0]\\ &[\circ_2^1\ \texttt{tape-contents}\ \texttt{pr}_2^1]\\ &[\circ_2^1\ \texttt{head-position}\ \texttt{pr}_2^1]]\end{aligned}$$

The `next-action` function recovers the triple containing the state into which M is about to move, the symbol that it is about to print onto its tape, and the direction in which it is about to move the read-write head. It finds this triple by doing a lookup, using M's current state and the symbol it is currently reading.

$$\begin{aligned}\texttt{next-action} \equiv [\circ_2^3\ &\texttt{TM-transition-lookup}\\ &\texttt{pr}_2^0\\ &[\circ_2^1\ \texttt{current-state}\ \texttt{pr}_2^1]\\ &\texttt{symbol-under-head}]\end{aligned}$$

The `next-head-position` function computes the next position of the read-write head, moving right (by applying `successor` to the current position) if the value of `direction` is 1 (encoding R), and left if it is 0. Note that moving left from position 0 yields a new position of 0, since $\texttt{predecessor}(0) = 0$ under our definitions.

$$\begin{aligned}\texttt{next-head-position} \equiv [\text{¿}_2\ &[\circ_2^1\ \texttt{TM-action-direction next-action}]\\ &[\circ_2^1\ \texttt{successor}\ [\circ_2^1\ \texttt{head-position}\ \texttt{pr}_2^1]]\\ &[\circ_2^1\ \texttt{predecessor}\ [\circ_2^1\ \texttt{head-position}\ \texttt{pr}_2^1]]]\end{aligned}$$

The `revised-tape-contents` function computes the string that results from the replacement of the character under the read-write head with the one that will be printed there during the next transition.

$$\begin{aligned}\texttt{revised-tape-contents} \equiv [\circ_2^4\ &\texttt{string-update}\\ &[\circ_2^1\ \texttt{TM-tape-alphabet}\ \texttt{pr}_2^0]\\ &[\circ_2^1\ \texttt{tape-contents}\ \texttt{pr}_2^1]\\ &[\circ_2^1\ \texttt{head-position}\ \texttt{pr}_2^1]\\ &[\circ_2^1\ \texttt{TM-action-symbol-to-print}\\ &\quad\texttt{next-action}]]\end{aligned}$$

If the read-write head is about to move farther to the right than it has ever previously been, we should append a blank to the string representing the tape contents, so as to preserve the invariant that the position of the read-write head is strictly less than the length of that string and can legitimately be used as an index into that string. The `next-tape-contents` function reconciles the outputs of the `next-head-position` and `revised-tape-contents` by appending the blank if necessary.

```
next-tape-contents ≡ [¿₂ [∘₂² equal?
                            [∘₂¹ string-length
                               revised-tape-contents]
                            next-head-position]
                         [∘₂³ string-append
                            [∘₂¹ TM-tape-alphabet pr₂⁰]
                            revised-tape-contents
                            one₂]
                         revised-tape-contents]
```

Finally, we can define `step` thus:

```
step ≡ [¿₂ [∘₂² TM-stopping-state? pr₂⁰ [∘₂¹ current-state pr₂¹]]
           pr₂¹
           [∘₂³ encode-configuration
              [∘₂¹ TM-action-target-state next-action]
              next-head-position
              next-tape-contents]]
```

The function `[◎₂ step]` can be used to drive a Turing machine forwards through a specified number of steps; for instance,

$$\texttt{[}\circledcirc_2\ \texttt{step]}(\bar{M}, \bar{c}, 23)$$

is the encoding for the configuration produced by starting a Turing machine M in the configuration c and letting it run for twenty-three steps (or until it halts, whichever comes first).

Exercises

17–1 Define a predicate `completely-blank?` that takes as inputs a configuration of a Turing machine and the size of that Turing machine's tape alphabet and determines whether any non-blank symbols appear on the tape described in the given configuration, yielding 1 if every symbol is a blank and 0 otherwise.

17–2 How, exactly, does `boot` handle the case in which the input string is ε? How does the configuration that `boot` returns reflect this?

17–3 Using `boot` and `step`, write an expression whose value encodes the configuration of a Turing machine M that has been started on input w and has just completed its second step of operation.

17–4 In the definition of `next-tape-contents` above, the third input to `string-append` should be the encoding for a *string of length 1* in which the sole symbol is a blank, yet the next-to-last line of the definition indicates that the third input will always be 1, the encoding for the *b*lank symbol itself. Reconcile this seeming inconsistency in the data types.

Chapter 18

Simulating Turing Machines in Operation

The *running time* of a Turing machine M on input s is the number of steps it takes M to halt, that is, to enter q_{accept} or q_{reject}. The partial recursive function `running-time` computes this number of steps, given the encodings for M and s:

$$\begin{aligned}\texttt{running-time} \equiv [\mu_2\ & [\circ_3^2\ \texttt{TM-stopping-state?} \\ & [\circ_3^1\ \texttt{current-state} \\ & \quad [\circ_3^3\ [\circledcirc_2\ \texttt{step}] \\ & \quad\ \texttt{pr}_3^0 \\ & \quad\ [\circ_3^2\ \texttt{boot}\ \texttt{pr}_3^0\ \texttt{pr}_3^1] \\ & \quad\ \texttt{pr}_3^2]]]].\end{aligned}$$

Since some Turing machines, on some inputs, never enter a stopping state, `running-time` is a *partial* recursive function.

The final configuration of the Turing machine in this setup is easily computed:

$$\texttt{final-configuration} \equiv [\circ_2^3\ [\circledcirc_2\ \texttt{step}]\ \texttt{pr}_2^0\ \texttt{boot}\ \texttt{running-time}]$$

We can determine whether M accepts s by comparing the state component of the final configuration to q_{accept}:

$$\begin{aligned}\texttt{accepts?} \equiv [\circ_2^2\ & \texttt{equal?} \\ & [\circ_2^1\ \texttt{current-state final-configuration}] \\ & [\circ_2^1\ \texttt{TM-accept-state}\ \texttt{pr}_2^0]].\end{aligned}$$

Since `final-configuration` and `accepts?` depend on `running-time`, they too are partial recursive functions, but not primitive recursive ones. Also, they too are undefined at certain inputs—specifically, in those cases where M, processing s as input, never reaches a stopping state.

We can define a similar `rejects?` predicate either by comparing the state component of the final configuration to q_{reject} rather than to q_{accept}, or more simply as

$$\texttt{rejects?} \equiv [\circ_2^1 \texttt{ not accepts?}].$$

Either way, we get a partial recursive function, and `rejects?` is undefined at exactly the same inputs as `accepts?`. It would be nice if we could separate out the cases in which the Turing machine does not reach any stopping state, by defining a predicate `runs-forever?` such that

$$\texttt{runs-forever?}(\bar{M}, \bar{s}) = \begin{cases} 1 & \text{if } \texttt{accepts?}(\bar{M}, \bar{s})\!\uparrow, \\ 0 & \text{otherwise.} \end{cases}$$

It is tempting to try to define `runs-forever?` as something like $[\circ_2^1$ `not` $[\circ_2^2$ `or accepts? rejects?]]`, but of course this function too is undefined in the cases where M does not halt. In fact, the `runs-forever?` predicate is not a partial recursive function.

Exercises

18–1 Could `running-time`$(\bar{M}, \bar{s})$ ever be less than `string-length`$(\bar{s})$? If so, give an example; if not, explain why not.

18–2 If the sequence of configurations of a Turing machine M, as it processes an input string s, ever includes a duplicate configuration, then M will never halt on input s, but will forever repeat the cycle of the configurations. Write a function `repeating-configuration?` that takes $\bar{M}$ and $\bar{s}$ as inputs, and outputs 1 if M ever enters the same configuration more than once while processing s (without halting). If M halts on input s without ever repeating a configuration, `repeating-configuration?` should yield 0. If M runs forever on input s without repeating a configuration, `repeating-configuration`$(\bar{M}, \bar{s})\!\uparrow$.

18–3 A *busy beaver* is a Turing machine that, when started on an initially all-blank tape, accumulates at least as much running time before halting as any other Turing machine with the same number of states and the same tape alphabet. How would you search for a busy beaver with, say, five states and a tape alphabet consisting of just the blank and the symbol `*`? Could there be a `busy-beaver` function that takes a number n of states and the size m of a tape alphabet as inputs and outputs the encoding of the n-state, m-symbol busy beaver?

Chapter 19

The Models Are Equivalent

That Turing machines can be simulated within the recursive-function model of computation suggests that there may be a relation between recursively enumerable sets and Turing-recognizable languages. In fact, except for the difference that the elements of languages are strings and those of recursively enumerable sets are natural numbers, the relation is identity.

Theorem 19.1 *The set of encodings of members of any Turing-recognizable language is recursively enumerable.*

Proof: Let L be any Turing-recognizable language, let M be a Turing machine that recognizes L, let $\bar{M}$ be the encoding for M, and let `bar-M-constant` be a nullary function that outputs $\bar{M}$. Then

$$[\mu_1\ [\circ_2^2\ \texttt{accepts?}\ \texttt{bar-M-constant}_2\ \texttt{pr}_2^0]]$$

is an acceptance function for the set of encodings of members of L, and (by construction) it is partial recursive.

For any string s, applying this function to the encoding for s in effect simulates the execution of M on input s. If M accepts s, then the unbounded minimization succeeds immediately with 0 as the second (ignored) input to the minimized predicate, so the function outputs 0; if M fails to accept s, then the unbounded minimization searches forever, and the function is undefined at input $\bar{s}$. Since there is a partial recursive acceptance function for the set of encodings of members of L, it is recursively enumerable. ■

Theorem 19.2 *If the set of encodings of members of a language is recursively enumerable, then the language is Turing-recognizable.*

Proof: Let L be any language, and suppose that the set of encodings of members of L is recursively enumerable. Let f be a partial recursive acceptance function for that set. Here is the construction plan for a Turing machine that recognizes L: "On input s, (1) attempt to compute $f(\bar{s})$; (2) *accept*."

Since f is partial recursive, it can be defined entirely in terms of the primitive functions, composition, recursion, and unbounded minimization. Automata theorists have shown (by construction) that a Turing machine can implement all of these patterns of computation, encoding natural numbers as strings of digits in some standard system of numeration chosen by the designer. For any string s that is a member of L, $f(\bar{s})\downarrow$, such a Turing machine will complete step (1), continue to step (2), and accept s. For any string s that is not a member of L, $f(\bar{s})\uparrow$, so the Turing machine described above will never complete step (1) and so will not accept s. So this Turing machine recognizes L, and hence L is Turing-recognizable. ■

Theorem 19.3 *A language is decidable if, and only if, the set of encodings of its members is recursive.*

Proof: Suppose first that a language L is decidable. Then there is a Turing machine M that decides it, and therefore also recognizes it. Also, the Turing machine M' that is exactly like M except that the roles of q_{accept} and q_{reject} are swapped decides (and therefore recognizes) the complement $\tilde{L}$ of L. Thus both L and $\tilde{L}$ are Turing-recognizable. Hence, by Theorem 19.1, both the set S_L of encodings of members of L and the set $S_{\tilde{L}}$ of encodings of members of $\tilde{L}$ are recursively enumerable. But every natural number encodes either a member of L or a member of $\tilde{L}$, so $S_{\tilde{L}}$ is the complement of S_L. So both S_L and its complement are recursively enumerable; hence S_L is recursive.

Conversely, suppose that S_L is recursive. Then both S_L and its complement, which is $S_{\tilde{L}}$, are recursively enumerable; hence, by 19.2, both L and $\tilde{L}$ are Turing-recognizable. A Turing machine that combines the recognizers for L and $\tilde{L}$, therefore, can decide L. ■

Exercises

19–1 Let's say that a Turing machine M *calculates* a given partial recursive function f if (and only if), when started on a tape that has a string of the form $\texttt{:1}^{n_0}\texttt{:1}^{n_1}\cdots\texttt{:1}^{n_{k-1}}$ at its left end (where k is the valence of f), and is otherwise blank, M halts if and only if $f(n_0, n_1, \ldots, n_{k-1})\downarrow$, and when M halts its tape is blank except for the string $\texttt{1}^{f(n_0,n_1,\ldots,n_{k-1})}$. Describe a Turing machine that calculates (a) `zero`; (b) `successor`; (c) $\texttt{pr}_4^2$.

19–2 Given Turing machines that calculate a function f of valence m and m functions $g_0, \ldots, g_{m-1}$ of valence n, describe how to construct a Turing machine that calculates $\texttt{[}\circ_n^m\ f\ g_0\ \ldots\ g_{m-1}\texttt{]}$.

19–3 Show that a set S of natural numbers is recursive if, and only if, the language $\{\texttt{1}^k \mid k \in S\}$ is decidable.

Chapter 20

The Parameter Theorem

The functions for encoding and decoding functions are similar to facilities for reflection in programming languages like Java and Python. They provide us with a way to construct partial recursive functions "on the fly," functions that can depend on inputs to the "meta-function" that constructs them.

As a simple warm-up example, let's define a function `constantify` that takes one input, a natural number n, and outputs the encoding for a nullary function that itself outputs n. (In other words, we want `constantify`(0) to be the encoding for `zero`, `constantify`(1) the encoding for `one`, `constantify`(2) the encoding for `two`, and so on.)

It is important to realize that this is a purely arithmetic function. We specified earlier that the encoding for `zero` is 0; we defined `one` as $[\circ_0^1$ `successor zero]`, so the encoding of `one` is $4k + 3$, where k is the encoding for the list $\texttt{cons}(1, \texttt{cons}(0, \texttt{cons}(1, \texttt{cons}(0, \texttt{nil}()))))$. (The first 1 is the upper composition index, the first 0 is the lower composition index, the second 1 is the encoding for `successor`, and the second 0 is the encoding for `zero`.) It turns out that the encoding for this list is 54, so that the encoding for `one` is 219. That's the value that we want `constantify`(1) to have.

Similarly, since we defined `two` as $[\circ_0^1$ `successor one]`, the encoding of `two` is $4k + 3$, where k encodes $\texttt{cons}(1, \texttt{cons}(0, \texttt{cons}(1, \texttt{cons}(219, \texttt{nil}()))))$. Computation reveals that k is

$$168499666669691498716668844293872691710232152640878578006897564059 8,$$

so that $4k + 3$ is

$$673998666678765994866675377175490766840928610563514312027590256239 5,$$

which is therefore the encoding for `two` and the value of `constantify`(2).

By doing some algebra involving the arithmetic definition of the `cons` function, we can determine the numerical relationship between successive values of `constantify`, and even write recursion equations that specify it:

$$\texttt{constantify}(0) \quad = \quad 0,$$

$$\texttt{constantify}(t+1) \quad = \quad 2^{\texttt{constantify}(t)+7} + 91.$$

Thus it is possible to define `constantify` by means of a direct recursion, using `raise-to-power` and `add`. In order to illustrate reflection, however, we'll write these equations in a different way:

$$\begin{aligned}
\texttt{constantify}(0) &= \texttt{encode-zero-function}(), \\
\texttt{constantify}(t+1) &= \texttt{encode-composition}(1, 0, \\
&\qquad \texttt{cons}(\texttt{encode-successor-function}(), \\
&\qquad\quad \texttt{cons}(\texttt{constantify}(t), \texttt{nil}())))
\end{aligned}$$

This shows that a recursion that defines `constantify` can also indicate the structure of the reflected code. When the input to `constantify` is positive, the number that results encodes a composition in which the upper index is 1, the lower index is 0, and the components are the successor function and the result of applying `constantify` recursively to the next smaller natural number (recovered through the projection function $\texttt{pr}_2^1$).

$$\begin{aligned}
\texttt{constantify} \equiv [&\Upsilon_0\ \texttt{encode-zero-function} \\
&[\circ_2^3\ \texttt{encode-composition} \\
&\quad \texttt{one}_2 \\
&\quad \texttt{zero}_2 \\
&\quad [\circ_2^2\ \texttt{cons} \\
&\qquad \texttt{encode-successor-function}_2 \\
&\qquad [\circ_2^2\ \texttt{cons}\ \texttt{pr}_2^1\ \texttt{nil}_2]]]]
\end{aligned}$$

A subtler use of reflection functions is exemplified by the proof of the *parameter theorem*:

Theorem 20.1 *For any natural numbers m and n, there is a primitive recursive function $\texttt{section}_n^m$ such that, for any function f of valence $m+n$, any encoding e of f, and any natural numbers $u_0, \ldots, u_{n-1}$, $\texttt{section}_n^m(e, u_0, \ldots, u_{n-1})$ encodes a function g such that*

$$f(x_0, \ldots, x_{m-1}, u_0, \ldots, u_{n-1}) = g(x_0, \ldots, x_{m-1})$$

for any natural numbers $x_0, \ldots, x_{m-1}$.

The idea is that $\texttt{section}_n^m$ reworks the encoding for f into an encoding for a function g that is similar to f, except that the values of the last n inputs have been hard-wired into g (they are $u_0, \ldots, u_{n-1}$). Thus g is a "parameterized" version of f. The name `section` comes from functional programming languages.

It is not too difficult to convert the preceding equation relating f and g into a definition of g in terms of f and the nullary functions u_0`-constant`, ..., u_{n-1}`-constant` that output $u_0, \ldots, u_{n-1}$ respectively:

$$g \equiv [\circ_m^{m+n}\ f\ \texttt{pr}_m^0\ \ldots\ \texttt{pr}_m^{m-1}\ u_0\texttt{-constant}_m\ \ldots\ u_{n-1}\texttt{-constant}_m]$$

The function $\texttt{section}_n^m$, in effect, automates the process of building this definition from the encoding for f and the inputs $u_0, \ldots, u_{n-1}$, using the reflective constructors to build the compositions and the projection functions, and applying `constantify` to turn each u_i into the corresponding u_i`-constant`.

The exact details of the definition depend, of course, on m and n, so what we give here is a template that indicates how to construct any member of the family. It presupposes that we have already defined nullary functions `zero`, `one`, ..., $(m-1)$`-constant`, m`-constant`, and n`-constant` that return $0, 1, \ldots, m-1, m$, and n respectively.

$\texttt{section}_n^m \equiv$
[$\circ_{n+1}^3$ encode-composition
  [$\circ_{n+1}^2$ add m-constant$_{n+1}$ n-constant$_{n+1}$]
  m-constant$_{n+1}$
  [$\circ_{n+1}^2$ cons
    $\texttt{pr}_{n+1}^0$
    [$\circ_{n+1}^2$ cons
      [$\circ_{n+1}^2$ encode-projection-function
        zero$_{n+1}$
        m-constant$_{n+1}$]
        $\vdots$
    [$\circ_{n+1}^2$ cons
      [$\circ_{n+1}^2$ encode-projection-function
        $(m-1)$-constant$_{n+1}$
        m-constant$_{n+1}$]
      [$\circ_{n+1}^2$ cons
        [$\circ_{n+1}^3$ encode-composition
          zero$_{n+1}$
          m-constant$_{n+1}$
          [$\circ_{n+1}^2$ cons
            [$\circ_{n+1}^1$ constantify $\texttt{pr}_{n+1}^1$]
            nil$_{n+1}$]]
          $\vdots$

(continued on next page)

$$
\begin{array}{l}
[\circ^2_{n+1}\ \texttt{cons} \\
\quad [\circ^3_{n+1}\ \texttt{encode-composition} \\
\qquad \texttt{zero}_{n+1} \\
\qquad m\text{-}\texttt{constant}_{n+1} \\
\qquad [\circ^2_{n+1}\ \texttt{cons} \\
\qquad\quad [\circ^1_{n+1}\ \texttt{constantify}\ \texttt{pr}^n_{n+1}] \\
\qquad\quad \texttt{nil}_{n+1}]] \\
\quad \texttt{nil}_{n+1}] \cdots]] \cdots]]]
\end{array}
$$

Since we use only primitive recursive functions in the definition of $\texttt{section}^m_n$, it is itself primitive recursive. This completes the proof of the theorem. ∎

To illustrate the use of the parameter theorem and the $\texttt{section}^m_n$ functions that it defines, let's prove that there is a singulary, primitive recursive function `converse` that takes as its input the encoding for any binary, partial recursive function f and returns the encoding for a binary, partial recursive function g such that $g(x_0, x_1) = f(x_1, x_0)$.

To begin with, notice that we can use the universality theorem of § 13 to get a function in which the encoding for f (let's call it e) is an explicit parameter, based on the equation

$$f(x_1, x_0) = \texttt{apply-binary}(e, x_1, x_0).$$

By composing `apply-binary` with suitable projection functions, we can rearrange the inputs in any way we like. In this particular case, the objective is to put x_0 and x_1 at the beginning of the inputs (so that they can play the role of the surviving variables in the parameter theorem), while placing e at the end (because we want it to be the parameter that is held constant).

$$
\begin{aligned}
f(x_1, x_0) &= \texttt{apply-binary}(e, x_1, x_0) \\
&= [\circ^3_3\ \texttt{apply-binary}\ \texttt{pr}^2_3\ \texttt{pr}^1_3\ \texttt{pr}^0_3](x_0, x_1, e)
\end{aligned}
$$

Now, $[\circ^3_3\ \texttt{apply-binary}\ \texttt{pr}^2_3\ \texttt{pr}^1_3\ \texttt{pr}^0_3]$ is, by construction, a partial recursive function, so it too will have an encoding, say t. Let `t-encoded` be a nullary function that outputs t.

It is now straightforward to define the `converse` function:

$$\texttt{converse} \equiv [\circ^2_1\ \texttt{section}^2_1\ \texttt{t-encoded}_1\ \texttt{pr}^0_1]$$

To see that this is the correct definition, consider what happens when we apply it to the encoding e of a binary, partial recursive function f:

$$
\begin{aligned}
\texttt{converse}(e) &= [\circ^2_1\ \texttt{section}^2_1\ \texttt{t-encoded}_1\ \texttt{pr}^0_1](e) \\
&= \texttt{section}^2_1(t, e)
\end{aligned}
$$

By the definition of $\texttt{section}_1^2$, $\texttt{section}_1^2(t, e)$ encodes a function g such that

$$\begin{aligned} g(x_0, x_1) &= [\circ_3^3\ \texttt{apply-binary}\ \texttt{pr}_3^2\ \texttt{pr}_3^1\ \texttt{pr}_3^0](x_0, x_1, e) \\ &= \texttt{apply-binary}(e, x_1, x_0) \\ &= f(x_1, x_0) \end{aligned}$$

as required.

Exercises

20–1 Suppose that r encodes the function `multiply`. Describe the function that $\texttt{section}_1^1(r, 12)$ encodes.

20–2 Using the parameter theorem, prove that there is a singulary, primitive recursive function `bumper` that takes as input the encoding for any singular, partial recursive function f and outputs the encoding for a singulary, partial recursive function g such that, for every natural number x, $g(x) = f(x) + 1$.

20–3 Using the parameter theorem, prove that there is a singulary, primitive recursive function `fuse-inputs` that takes as input the encoding for any binary, partial recursive function f and outputs the encoding for a singulary, partial recursive function g such that, for every natural number x, $g(x) = f(x, x)$.

Chapter 21

The Recursion Theorem

Theorem 21.1 *There is a nullary, primitive recursive function* `self` *that outputs its own encoding.*

First, let's develop a helper function, `compose-with-constantification`. This is a singulary function that inputs the encoding for any singulary partial recursive function f, computes (as an intermediate result) the encoding for a nullary function $f^\bullet$ that outputs the encoding for f, and finally computes and outputs the encoding for the composition of f and $f^\bullet$:

$$\begin{aligned}&\texttt{compose-with-constantification} \equiv \\ &\quad [\circ_1^3\ \texttt{encode-composition} \\ &\qquad \texttt{one}_1 \\ &\qquad \texttt{zero}_1 \\ &\qquad [\circ_1^2\ \texttt{cons}\ \texttt{pr}_1^0\ [\circ_1^2\ \texttt{cons constantify nil}_1]]]\end{aligned}$$

Since `compose-with-constantification` is primitive recursive, there is a natural number c that encodes it, and there is a nullary function `constant-c` that outputs c. That's all we need in order to define `self`:

$$\texttt{self} \equiv [\circ_0^1\ \texttt{compose-with-constantification constant-c}]$$

■

Theorem 21.2 *For any binary, partial recursive function t, there is a singulary, partial recursive function r such that, for every natural number x, $r(x) = t(\bar{r}, x)$, where $\bar{r}$ is the encoding for r.*

Proof: Since t is a partial recursive function, it has an encoding. Let `constant-t` be a nullary function that returns the encoding for t.

Our helper function this time is a little more complicated, because of the need to carry along the input x and the consequent adjustments to valences:

```
recursion-theorem-helper ≡
    [∘₁³ encode-composition
        two₁
        one₁
        [∘₁² cons
            [∘₁⁰ constant-t]
            [∘₁² cons
                [∘₁³ encode-composition
                    one₁
                    one₁
                    [∘₁² cons
                        pr₁⁰
                        [∘₁² cons
                            [∘₁³ encode-composition
                                zero₁
                                one₁
                                [∘₁² cons constantify nil₁]]
                            nil₁]]]
                [∘₁² cons
                    [∘₁² encode-projection-function zero₁ one₁]
                    nil₁]]]]
```

For any fixed choice of t, `recursion-theorem-helper` has an encoding h, and so there is a nullary, primitive recursive function `constant-h` that outputs h. Then we can define r thus:

```
r ≡ [∘₁² t
        [∘₁¹ recursion-theorem-helper [∘₁⁰ constant-h]]
        pr₁⁰]
```

Careful comparison of the definition of `recursion-theorem-helper` with the definition of r shows that, when `recursion-theorem-helper` is given its own encoding h as input, it outputs the encoding for r. Since r works by feeding h to `recursion-theorem-helper` and transmitting the result, along with its own input, to t, the construction guarantees that r responds to any input just the way t responds if given the encoding for r as its first input and the input to r as its second, as required. ■

Exercises

21–1 Prove that there is a singulary, primitive recursive function, which we might call `add-to-self`, that takes as input any natural number n and outputs the sum of n and its own encoding.

21–2 Prove that there is a singulary, partial recursive function, which we might call `apply-to-self`, that takes as input the encoding e for any singulary, partial recursive function f and outputs the result of applying f to its own encoding (that is, to `apply-to-self`'s encoding). (If the result of applying f to `apply-to-self`'s encoding is undefined, then `apply-to-self`$(e)\uparrow$ as well.)

21–3 Let a be the encoding of the `apply-to-self` function in the preceding exercise. Is `apply-to-self`(a) defined?

Chapter 22

Rice's Theorem

We have seen that the class of recursive functions includes an immense variety of useful numerical functions and provides a computational framework within which many kinds of algorithms can be expressed and studied. We have also seen that it is possible to deploy this framework reflexively, finding explicit data representations, "programs," for all partial recursive functions and developing tools for analyzing such representations.

It is true that we have encountered some limitations on this approach. We found that such predicates as `defined?` and `runs-forever?` are not recursive or even partial recursive, and that the characteristic function of K is not recursive. However, it is tempting to suppose that this phenomenon is limited to a few extreme or pathological cases, and that the reflexive application of recursive function theory could succeed in other, less overtly paradoxical cases. Unfortunately, the very power of this approach limits its utility. It turns out that the most interesting properties of partial recursive functions cannot be expressed by recursive predicates or recursive sets.

Let's say that a set S of natural numbers is *functionally consistent* if, for every partial recursive function f, either all of the encodings of f are members of S or none of them are. The idea is that we want to think of S as a set of *partial recursive functions,* not just as a set of encodings for such functions, so we want to exclude cases in which f belongs to the set S when it is encoded one way but fails to belong when it is encoded some other way. Requiring S to be functionally consistent ensures that we can think of S in this way without running into contradictions.

(A minor technical point: Since some natural numbers don't encode partial recursive functions at all, the preceding definition of functional consistency doesn't constrain their membership or non-membership. Adding a clause to deal with these non-encoding numbers is technically convenient and simplifies proofs about functionally consistent sets. Since $\texttt{apply}(e, \texttt{nil})\uparrow$ when e does not encode any function, it seems most natural to regard all such natural numbers as "pseudo-encodings" for `mist`. We'll stipulate, then, that to count as functionally consistent S must contain all of the non-encoding natural numbers if it

contains the encodings for `mist`, and must not contain any of the non-encoding natural numbers if it does not contain the encodings for `mist`.)

Now we're ready to state and prove Rice's Theorem:

Theorem 22.1 *No functionally consistent set S of natural numbers other than $\emptyset$ and $\mathbb{N}$ is recursive.*

Proof. Let S be any functionally consistent set S of natural numbers other than $\emptyset$ and $\mathbb{N}$. The proof is by contradiction. We'll start by assuming that that S is recursive, and on that basis derive the conclusion that the set K (as defined in §15) is also recursive. Since this conclusion contradicts Theorem 15.2, thus disproving the assumption that S is recursive.

Case A. Suppose, first, that the encodings for $\texttt{mist}_1$ are not members of S. Let f be any partial recursive function whose encodings are members of S. (There must be at least one such function, since otherwise S would be $\emptyset$.) Let g be the binary function defined by

$$\texttt{[}\circ_2^2\ \texttt{pr}_2^1\ \texttt{[}\circ_2^1\ \texttt{[}\circ_1^2\ \texttt{apply car cdr]}\ \texttt{pr}_2^1\texttt{]}\ \texttt{[}\circ_2^1\ f\ \texttt{pr}_2^0\texttt{]]}$$

and let $\bar{g}$ be an encoding for g.

Given two arguments, n and x, the function g tries to compute two values and selects the second of them if both are defined; otherwise, $g(n,x)\uparrow$. The first of these two values is $\texttt{[}\circ_1^2\ \texttt{apply car cdr]}(x)$ and the second is $f(n)$:

$$\begin{aligned} g(n,x) &= \texttt{[}\circ_2^2\ \texttt{pr}_2^1\ \texttt{[}\circ_2^1\ \texttt{[}\circ_1^2\ \texttt{apply car cdr]}\ \texttt{pr}_2^1\texttt{]}\ \texttt{[}\circ_2^1\ f\ \texttt{pr}_2^0\texttt{]]}(n,x) \\ &= \texttt{pr}_2^1(\texttt{[}\circ_2^1\ \texttt{[}\circ_1^2\ \texttt{apply car cdr]}\ \texttt{pr}_2^1\texttt{]}(n,x), \texttt{[}\circ_2^1\ f\ \texttt{pr}_2^0\texttt{]}(n,x)) \\ &= \texttt{pr}_2^1(\texttt{[}\circ_1^2\ \texttt{apply car cdr]}(\texttt{pr}_2^1(n,x)), f(\texttt{pr}_2^0(n,x))) \\ &= \texttt{pr}_2^1(\texttt{[}\circ_1^2\ \texttt{apply car cdr]}(x), f(n)). \end{aligned}$$

Now, if $x \in K$, $\texttt{[}\circ_1^2\ \texttt{apply car cdr]}(x)\downarrow$, in which case $g(n,x) = f(n)$ when $f(n)\downarrow$, and $g(n,x)\uparrow$ when $f(n)\uparrow$. If we regard x as a fixed parameter in a sectioning operation and allow only n to vary, the resulting section of g is precisely f. But, by the parameter theorem (20.1), $\texttt{section}_1^1(\bar{g},x)$ is the encoding for that section of g, which is therefore also an encoding for f and hence a member of S. So if $x \in K$, $\texttt{section}_1^1(\bar{g},x) \in S$.

On the other hand, if $x \notin K$, $\texttt{[}\circ_1^2\ \texttt{apply car cdr]}(x)\uparrow$, so that $g(n,x)\uparrow$ regardless of what happens with $f(n)$. This time, if we regard x as a fixed parameter in a sectioning operation and allow only n to vary, the resulting section of g is precisely $\texttt{mist}_1$. But encodings for $\texttt{mist}_1$ are non-members of S, so that $\texttt{section}_1^1(\bar{g},x) \notin S$.

Combining these results, $x \in K$ if, and only if, $\texttt{section}_1^1(\bar{g},x) \in S$. Now, if S were recursive, its characteristic function p would also be recursive. But in that case

$$\texttt{[}\circ_1^1\ p\ \texttt{[}\circ_1^2\ \texttt{section}_1^1\ \texttt{g-bar-encoded}_1\ \texttt{pr}_1^0\texttt{]]}$$

(where `g-bar-encoded` is a nullary function that returns $\bar{g}$) would be a recursive characteristic function for K, since p, $\texttt{section}_1^1$, `g-bar-encoded`, and $\texttt{pr}_1^0$ are all

both partial recursive and total, and so therefore are their compositions. A set that has a recursive characteristic function is recursive, so K must be recursive. Yet we know from Theorem 15.2 that K is not recursive, so the assumption that S is recursive is disproven.

Case B. Alternatively, then, suppose that the encodings for $\texttt{mist}_1$ *are* members of S, and this time f be any partial recursive function whose encodings are *not* members of S. (Again, there must be at least one such function, because otherwise S would be $\mathbb{N}$.) Essentially the same line of reasoning now shows that $x \in K$ if, and only if, $\texttt{section}_1^1(\bar{g}, x) \notin S$ (since the section of g in which x is a fixed parameter matches f when $x \in K$ and $\texttt{mist}_1$ when $x \notin K$), but this time the encodings of $\texttt{mist}_1$ are members of S and the encodings of f are non-members of S.

As before, if S were recursive, it would have a recursive characteristic function p. But then

$$\texttt{[}\circ_1^1\ \texttt{[}\circ_1^1\ \texttt{not}\ p\texttt{]}\ \texttt{[}\circ_1^2\ \texttt{section}_1^1\ \texttt{g-bar-encoded}_1\ \texttt{pr}_1^0\texttt{]]}$$

would be a recursive characteristic function for K, implying that K is recursive, which again contradicts Theorem 15.2. ■

The import of Rice's Theorem is that any property of partial recursive functions that is non-trivial (in the sense that some partial recursive functions have the property and others do not) cannot be tested by applying a recursive predicate to an encoding for a candidate function, nor can one represent the property as a recursive set containing exactly the encodings of the functions that have the property. No such recursive predicate or recursive set exists.

Exercises

22–1 Let S be the set of natural numbers that encode partial recursive functions f such that $f(n) = 0$ for some $n \in \mathbb{N}$. Prove that S is not recursive.

.

22–2 Let p be the predicate defined by

$$p(n) = \begin{cases} 0 & \text{if } n \text{ encodes a function } f \text{ such that } f(0) = 42, \\ 1 & \text{otherwise.} \end{cases}$$

Prove that p is not partial recursive.

22–3 Prove that the set of encodings for Turing machines that recognize the empty language is not a recursive set.

22–4 The set S of natural numbers n that satisfy both `function?` and `checked-composition-function?` includes at least one encoding for every partial recursive function that can be defined by composition, and does not contain any encoding for any partial recursive function that cannot be defined by composition. Show that S is nevertheless a recursive set, and explain why this result does not contradict Rice's Theorem.

I am indebted to students and mentors in my course "Automata, formal languages, and computational complexity" at Grinnell College for their constructive suggestions, and specifically to Doug Babcock, Ziwen Chen, Martin Dluhos, Arjun Guha, Davis Hart, Lindsey Kuper, Avram Lyon, Angeline Namai, Ogechi Nnadi, Norman Perlmutter, Russel Steinbach, Jonathan Wellons, and Zelealem Yilma for finding and correcting errors.