Chapter 3

Indexing and Diagonalization over *Prim*

The theory of uncomputability depends crucially on matters that might at first seem fussy or arcane, like efficient encoding of finite sequences, which has already been seen to reduce different forms of recursion to primitive recursion. In this chapter we consider another of these crucial but apparently fussy ideas: assigning code numbers to primitive recursive functions. Code numbers for functions are important because they allow us to think of primitive recursive operations on primitive recursive functions. Of course, no primitive recursive function operates on functions, but primitive recursive functions can operate on code numbers of functions. Since the code number determines the function, that achieves the same effect. As it happens, some of the deepest ideas in the theory of computability concern effective operations on code numbers of functions.

The general strategy for indexing functions (the input-output behaviors of programs) will be to effectively decode numbers into unique programs and to say that a function has a given index if decoding the index yields a program of the function. This will mean that, typically, each function has many indices, since many different programs can compute it. In primitive recursion this is readily seen to be the case, since any number of projections can be composed on the outside of a sensible program and the result is the same.

$$\begin{aligned} C(p_1^1, f) &= C(p_1^1, C(p_1^1, f)) \\ &= C(p_1^1, C(p_1^1, C(p_1^1, f))) \\ &= \ldots \end{aligned}$$

So there is no question of having a unique, effective code number for each function. The important properties of our numbering of the primitive recursive functions are:

- There is an intuitively effective procedure to decode numbers into unique programs.

- Every primitive recursive program has a number (i.e., the coding is surjective or "onto").

- Every number decodes into some program (i.e., the coding is total).

We have to index functions of every arity. There are two possible strategies:

1. One big enumeration of functions of all arities: one could number all the functions of all arities at once, so that the index determines the arity. But then one would have to go back and construct sub-enumerations of the unary functions, binary functions, etc. anyway. Also, the inductive clauses in the definition become inelegant because, for example, you can't apply primitive recursion to functions of the wrong arities.

2. Separate enumerations for different arities: one can have the same index pick out different functions for different specified arities. This approach is more natural when we get to Turing machines, since any number of arguments can be given to such a machine and the function computed depends on how many arguments are given.

We will pursue the second strategy, for the reasons given.

$f_x^k(y_1, \ldots, y_k) =$ the value of the x^{th} k-ary program on inputs $(y_1, \ldots, y_k)$.

Then the unary primitive recursive functions will be enumerated as:

$$f_0^1, f_1^1, f_2^1, \ldots.$$

3.1 An indexing of primitive recursive functions

There are lots of strategies for encoding derivation formulas for primitive recursive functions. The following approach is to use the length of the sequence encoded by the index to determine the top-level recursive operator applied to generate the function and to use the components of the coded sequence to determine which functions the operation is applied to. Some of the numbers occurring in the sequence may be "dummies" that do nothing but make the sequence longer.

In reading the following definition, don't forget that positions in coded sequences are counted starting from 0 and that the length of a code number is number of items occurring in the sequence (including the one indexed as 0), e.g.:

$$\begin{aligned} (< i, j, k >)_2 &= k; \\ lh(< i, j, k >) &= 3. \end{aligned}$$

Now define the function denoted by f_x^k by cases as follows:

$$lh(x) = 0 \quad \Rightarrow \quad f_x^k = \begin{cases} o' & \text{if } k = 0; \\ C(o, p_k^1) & \text{otherwise;} \end{cases}$$

$$lh(x) = 1 \quad \Rightarrow \quad f_x^k = \begin{cases} o' & \text{if } k = 0; \\ C(s, p_k^1) & \text{otherwise;} \end{cases}$$

$$lh(x) = 2 \quad \Rightarrow \quad f_x^k = \begin{cases} o' & \text{if } k = (x)_0; \\ p_k^{\min((x)_0+1,k)} & \text{otherwise;} \end{cases}$$

$$lh(x) = 3 \quad \Rightarrow \quad f_x^k = C(f_{(x)_0}^{lh((x)_1)}, f_{((x)_1)_0}^k, \ldots, f_{((x)_1)_{lh((x)_1)\dot{-}1}}^k);$$

$$lh(x) \geq 4 \quad \Rightarrow \quad f_x^k = \begin{cases} o' & \text{if } k = 0; \\ R(f_{(x)_0}^{k\dot{-}1}, f_{(x)_1}^{k+1}) & \text{otherwise.} \end{cases}$$

Observations:

1. After clause 4, we arbitrarily repeat clause 1, since every primitive recursive function is numbered already by clause 4. Remember, that it's hopeless to give a unique effective code number to each primitive recursive function anyway, so redundancy is all right.

2. In all clauses but 3, we simply insert o' as a placeholder when $k = 0$, because there is no zero-ary version of successor, projection, or primitive recursion. You may as well think of 0 as the true value of a zero-ary successor, projection or primitive recursion.

3. The most complicated clause is the composition case ($lh(x) = 3$). Given index x, decode x into $\langle i, j, k \rangle$ and treat $i = (x)_0$ as the index of the outer function and $j = (x)_1$ as an index of a list of numbers (possibly empty)

$$((x)_1)_0, ((x)_1)_1, \ldots, ((x)_1)_{lh((x)_1)\dot{-}1},$$

   each of which is interpreted as the index of a function embedded in the composition. The arity of the outer function is $lh((x)_1)$ and the arity of the inner functions is taken to be k, so that the resulting composition has the required arity. Composition can deal with zero-ary functions on the inside or the outside. For example:

$$\begin{aligned} C(o') &= o'; \\ C(s, o') &= c_1'. \end{aligned}$$

4. Each primitive recursive function of a given arity has a number (by induction on the depth of operator applications in primitive recursive definitions).

5. Each number codes some function (by induction on **N**).

6. Each map h_k such that $h_k(x, y_1, \ldots, y_k) = f_x^k(y_1, \ldots, y_k)$ is intuitively effective.

7. Component derivation trees always have lower indices than derivation trees of which they are components. Thus, induction on **N** corresponds to induction on function embedding depth.

Exercise 3.1 *Find the indices of a few primitive recursive functions. Present your code numbers in terms of the Gödel coding braces $\langle x_1, \ldots, x_n \rangle$ rather than as numerals, since the numerals will get too big due to all of the exponentiation in the coding scheme.*

Exercise 3.2 *Write down one case of the base case and one case of the induction case of the proof of observation 4.*

Exercise 3.3 *Write down one case of the base case and one case of the induction case of the proof of observation 5.*

Exercise 3.4 *Prove observation 7.*

3.2 Diagonalization

Also see Rogers, pp. 10-12.

Programming (= defining) can only show that a function is primitive recursive. Failure to find a program is like failure to find a logical proof. It leaves the following choice: either there is no proof or you aren't smart enough to find it. To prove that there is no program at all, one requires a mathematical specification of the possible programs and of the functions the programs define and then you have to show that a given function is different from each of those. Typically this is done by defining a function that differs from each programmable function in one place, depending on its position in an enumeration. If one thinks of each function's values as being listed as rows in an infinite table, then one technique is to construct the function to differ from each entry on the diagonal $T[0,0], T[1,1], \ldots$ of the table. So negative arguments are often called "diagonal arguments" even if the place where a given function differs isn't exactly on the diagonal. Here's an easily constructed, intuitively effective, total function that is not primitive recursive:

$$diag(x) = f_x^1(x) + 1.$$

Intuitively, decode x into a unary program, simulate the program on x itself and apply successor. But evidently, $diag(x)$ differs from the x^{th} unary primitive recursive function at position x, so $diag$ is not primitive recursive.

This is literally a diagonal argument, since if you think of a table T whose entry $T[x, y] = f_x^1(y) + 1$ then

$$diag(x) = T[x, x] + 1;$$

so *diag* modifies each cell of the table along the diagonal.

Thesis: intuitive effectiveness outruns the primitive recursive functions.

Exercise 3.5 *Describe an effectively computable function f such that $f(x) = T[x, x]$, for each $x \in \mathbf{N}$, and yet f can be shown not to be primitive recursive by "diagonalization" off of the diagonal.*

Exercise 3.6 *Is the table $T[x, y]$, itself, a primitive recursive function of two variables? Why or why not? Hint: you can't just assume that primitive recursive functions are closed under variable substitution! You have to first prove that if $g(x, y)$ is primitive recursive, then so is $h(x) = g(x, x)$. In chapter 2, you showed this only for substitution of a constant, which is different.*

Note that this conclusion is not (yet) a theorem because "effective" is (still) not mathematically definite. Could some kind of ultra-fancy recursion exhaust all of the intuitively effective functions?

No. For the logic of the diagonal argument will still work. Just add a clause for the new, fancy recursion into our enumeration to obtain enumeration

$$g_x^k(y).$$

Now define

$$diag(x) = g_x^1(x) + 1.$$

This will still be an intuitively effective total function that transcends the newly defined collection! So our thesis is even stronger:

Strengthened thesis: intuitive effectiveness outruns any effectively enumerable family of *total* functions.

The trouble is that recursion operators are guaranteed to produce total functions. Total functions leave "targets" everywhere along the diagonal for the diagonal argument to "hit". Defining total functions out of partial functions allows for the possibility that some cells along the diagonal are "missing". Then *diag* will also be undefined there (since it tries to add 1 to an undefined value) and may be identical to the function characterizing that row in the table. That is why the theory of computable, total functions inevitably leads one to study the theory of computable, partial functions, to which we now proceed.