Chapter 3

Indexing and Diagonalization

Our general strategy for indexing functions (the input-output behaviors of programs) will be to effectively decode numbers into unique programs and 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.

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

For this reason, we don't have to be too careful about each program having a unique index corresponding to it: functions will end up with infinitely many redundant numbers anyway! The important properties are:

- There is an effective procedure to decode numbers into unique programs.
- Every primitive recursive program has a number.
- Every number decodes into some program.

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

1. One big enumeration of functions of all arities: We could also number all the functions of all arities at once, so that the index determines the arity. But then we have to go back and construct sub-enumerations of the unary functions, binary functions, etc. in order to set up diagonal arguments. 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: We can have the same index pick out different functions for different specified arities. This approach is natural when we get to Turing machines, since what a Turing machine does depends on how many arguments it is given.

We will follow the second strategy.

$f_x^k(y_1, \ldots, y_k)$ = 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:

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

Here's one way to do it. Recall that $(n)_i$ is the primitive recursive decoding function for the primitive recursive finite sequence encoding $\langle x \rangle$.

3.1 An indexing of primitive recursive functions of given arities

The idea of our indexing is to use the length of the sequence encoded by the index to determine the last primitive 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.

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:

$$\begin{aligned}
&k = 0 \Rightarrow f_x^k = x; \\
&k > 0 \wedge \\
&\quad lh(x) = 0 \Rightarrow f_x^k = C(o, p_k^1); \\
&\quad lh(x) = 1 \Rightarrow f_x^k = C(s, p_k^1); \\
&\quad lh(x) = 2 \Rightarrow f_x^k = p_k^{\min((x)_0, k)}; \\
&\quad lh(x) = 3 \Rightarrow f_x^k = C(f_{(x)_0}^{lh((x)_1)}, f_{((x)_1)_0}^k, \ldots, f_{((x)_1)_{lh((x)_1) \dot{-} 1}}^k); \\
&\quad lh(x) = 4 \Rightarrow f_x^k = R(f_{(x)_0}^{k \dot{-} 1}, f_{(x)_1}^{k+1}); \\
&\quad lh(x) > 4 \Rightarrow f_x^k = C(o, p_k^1).
\end{aligned}$$

Observations:

- The most complicated clause is the composition case ($lh(x) = 3$). Given index x, we 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

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

  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.

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

- Each number codes some function (by induction on $\mathbf{N}$).

- Each map h_k such that $h_k(x, y) = f_x^k(y)$ is intuitively effective.

- Component functions always have lower indices than the functions of which they are components. Thus, induction on $\mathbf{N}$ corresponds to induction on function embedding depth.

- Notice how nicely having the arity given as an input together with our convention that constants are 0-ary functions allows us to steer past the restrictions on the domains of C and R without including them as special conditions. For example, the composition case works by treating the second item occurring in the sequence encoded by x as the code number of the sequence of indices of embedded functions. This nails down the arity of the outer function. The arities of the embedded functions are recovered from the given k.

Exercise 3.1 *Find the indices of a few primitive recursive functions. You can present them in terms of the encoding.*

Exercise 3.2 *Prove that all the functions occurring in a derivation tree for f_x^k have indices strictly less than x.*

Exercise 3.3 *Prove that each primitive recursive function has infinitely many indices.*

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 we aren't smart enough to find it (David Hume: which would be the greater miracle?). To prove that there is no program at all, we need to have a mathematical specification of the possible programs and of the functions the programs define and then we have to show that a given function is different from each of those. Typically this is done by making our function differ from each given function in one place, depending on its position in an enumeration. If we think of each function's values being listed as rows in an infinite table, then one technique is to show that the given function differs from a given row on the diagonal. 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 easy "logician's" example of an 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 we 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.

Note that this conclusion is not (yet) a theorem because "effective" is (still) not mathematically definite. Could some kind of ultra-fancy effective 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 kind of recursion into our 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 kind of effective primitive recursive operation that produces only 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. This is where the theory of computability is headed.

But first, we are going to look at a hierarchy based on primitive recursion.