Chapter 4

The Partial Recursive Functions

You have seen that more and more nested recursion yields more and more functions, *ad infinitum*. And lest one suspect that there is a concept of "mega-recursion" that captures effectiveness altogether, you have seen how such functions could be effectively indexed (add a new clause for mega-recursion) and then effectively diagonalized to yield an intuitively computable function that is not mega-recursive. The only solution is to not leave "targets" everywhere in the Cantorian table of function values for the impending diagonalization to strike. (Think of the children's game Battleship$^{\text{TM}}$. It's easy to win if every ship of your opponent has to sit on the diagonal of the grid!)

The moral is that the only way to effectively enumerate all effectively enumerable functions is to avoid diagonalization by allowing for some functions that are not defined on some arguments. In the mathematical Battleship$^{\text{TM}}$game, these *partial* computable functions leave gaps that save them from diagonal arguments.

In computational terms, these undefined values correspond to the program or description of the function going into an "infinite loop" or running off the end of an unbounded search. Infinite loops and partial functions are the price that must be paid for a computer language capable of computing all intuitively computable total functions. To catch all the flounder, you have to put up with catching some old tires.

4.1 Partial Functions

First, consider some concepts pertaining to partial functions in general. The notation is not standardized, so choices must be clarified at the outset.

1. A relation R on $A \times B$ is just a subset of $A \times B$.
2. We write $R(a, b)$ to mean $(a, b) \in R$.

3. $dom(R) = \{a \in A : (\exists b \in B)((a,b) \in R)\}$.
4. $range(R) = \{b \in B : (\exists a \in A)((a,b) \in R)\}$.
5. A function $f : A \to B$ is a single-valued relation in $A \times B$.
6. If $x \notin dom(f)$ then we say that $f(x)$ is undefined.
7. A function f is total with respect to some set A (usually understood from context) just in case $dom(f) = A$.
8. A standard convention is to denote total functions with lower-case Latin letters like f, g, h and partial functions with lower-case Greek letters like ϕ, ψ.

The following points are *very important*.

1. One writes $\phi(x) \simeq y$ to mean that $(x,y) \in \phi$. *One is not entitled in the theory of partial functions to assume that closed terms like $f(6)$ denote.* Since the logical proof rules you learned in logic class that govern function symbols and identity *presuppose* that function symbols in the language denote total functions, this means that *the notation $\phi(x) \simeq y$ must be eliminated as shown before such proof rules are applied in a proof.*
2. The composition operator on partial functions must be re-interpreted so that if any function in a composition fails to return a value, the whole composition is undefined as well. More precisely, the whole composition

   $$C(\psi, \phi_1, \ldots, \phi_n)(\vec{x})$$

   is undefined just in case any one of the component applications $\phi_1(x)$ is undefined or they are all defined but

   $$\psi(\phi_1(\vec{x}), \ldots, \phi_n(\vec{x}))$$

   is undefined.
3. Similarly, the primitive recursion $R(g,f)(n,\vec{x})$ is undefined if any intermediate value involved in the computation of $R(g,f)(n,\vec{x})$ is undefined.

4.2 minimization

There are many ways to introduce partial yet calculable functions. One such way is to close $Prim$ under the minimization operator M, where $M(\phi)$ denotes the unique, partial function:

$$(\mu x)\phi(x,\vec{y}) \simeq 0,$$

which returns the least x such that $\phi(x,\vec{y}) \simeq 0$ and $\phi(z,\vec{y})$ is *defined* and nonzero for all $z < x$. Note that whenever an "infinite loop" is hit among the successive

computations $\phi(0,\vec{y}), \phi(1,\vec{y}), \phi(2,\vec{y}), \ldots$ the whole minimization crashes. You may think of minimization as a stupid, "serial" computational search that goes off the deep end if any successive computation doesn't halt or if there is no x such that $\phi(x,\vec{y}) \simeq 0$. As in the case of bounded minimization, let

$$(\mu x)R(x,\vec{y}) \simeq (\mu x)[\overline{sg}(\chi_R(x,\vec{y})) \simeq 0].$$

4.3 The Zen of Dovetailing

Suppose you want to find a, b such that $R(a,b,\vec{z})$ is true. It would be a bad idea to check $a = 0$ for all possible values of b prior to checking $a = 1$, for it may be that $R(0,n,\vec{z})$ is false for all n, but that $R(1,1,\vec{z})$ is true, in which case the search would run on forever before finding the pair $a = 1, b = 1$.

It is a curious historical fact that while medieval Western philosophy was employed by the Church to come up with static demonstrations of timeless Church dogmas, Zen masters in medieval Japan made their way in the world by giving fencing advice to samurai warriors. Since samurai wore thin armor and carried extremely lethal blades, life or death could be measured in fractions of a second. In a field of expert enemies, it would be a (truly) fatal error to focus on one enemy to the exclusion of the others. This fatal focus of attention on an isolated feature of battle was called a "suki" or "gap". Zen Buddhists recognize "suki" as a special symptom of the general human sin of trying to conceptualize and partition reality into objects of desire. There is a Zen koan, or parable, about the "thousand armed" Kannon. If Kannon were to fixate on one task prior to beginning the others, her arms would be useless; but since she avoids "suki" and works on all problems with all arms at once (parallel computation) she can achieve marvels of kindness.

In a similar spirit, you can use the n-ary primitive recursive encoding to avoid needless infinite loops in multi-dimensional searches. Instead of searching for the first component and then for the second, search for the code number of a pair whose components satisfy the relation. Then return the desired component of the number we find.

$$a(\vec{z}) \simeq ((\mu w)R((w)_0,(w)_1,\vec{z}))_0;$$

$$b(\vec{z}) \simeq ((\mu w)R((w)_0,(w)_1,\vec{z}))_1.$$

Searching in parallel by minimization over a coded tuple is called *dovetailing* because infinite searches are interleaved together to avoid infinite loops. This is the basic programming technique in recursive function theory and is involved in almost every interesting construction. It will come into play shortly.

4.4 The Partial Recursive Functions

Although composition and primitive recursion produce total functions from total functions, once minimization is applied, partial functions end up in the mix, so

one must employ the extended notions of composition and primitive recursion introduced above, which apply to partial functions.

Let $Part$ denote the least set X such that

1. the basic functions o, s, p_k^i are all in X;

2. X is closed under the operators C, R, M.

Evidently, all primitive recursive functions are partial recursive (why?). Since primitive recursive functions are total, there are total, partial recursive functions, so this standard terminology is awkward, but mathematicians are creatures of habit regarding terminology, however unfortunate.

minimization clearly generates functions not in $Prim$. It is an interesting question whether minimization also takes over some of the work done by primitive recursion inside of $Prim$. The answer is that primitive recursion contributes nothing after the G/"odel decoding function $(x)_y$ is defined, so that if $(x)_y$ were a basic function, primitive recursion could be dropped from the definition of $Part$ altogether.

Proposition 4.1 *$Part$ is the least set X such that*

1. *the basic functions o, s, p_k^i are all in X;*

2. *the decoding function $(x)_y$ is in X;*

3. *X is closed under operators C, M.*

Exercise 4.1 *Prove the preceding proposition. Hint: show that any function defined by means of R can be defined by means of M, using the decoding functions. Suppose that $f(x, \overrightarrow{y})$ is defined by $R(g, h)$. To compute $f(x, \overrightarrow{y})$, use minimization to search for a code number of the value sequence $(f(0, \overrightarrow{y}), \ldots, f(x, \overrightarrow{y})$. Then read off the last item in the sequence from the code number.*

The same coding idea can be applied to show that $Part$ is closed under double recursion, triple recursion, etc. That suggests that virtually any kind of recursion you might define in terms of partial recursive functions yields partial recursive functions and "explains" why allowing for unbounded searches transcends the power of primitive recursion.

4.5 Recursive Relations

A relation $R(\overrightarrow{x})$ is *recursive* iff the characteristic function of R is partial recursive. Since characteristic functions are always total, it follows that the characteristic function is total recursive.

4.6 Indexing the Partial Recursive Functions

Suppose there were an effective indexing g_i^k of the intuitively total, effectively computable functions of each arity k. Then you could construct the intuitively effective and total binary function $h(x,y) = g_x^1(y)$. Then $g(x) = h(x,x) + 1$ is intuitively total and effectively computable and differs from each intuitively computable, total function (since all of them are caught by the enumeration). Contradiction. So if Tot is to include all intuitively effectively computable functions, Tot cannot be effectively enumerated. So if we are to effectively enumerate all such functions, we must also include some non-total, effectively computable functions. That is why computability theorists assign code numbers to functions in $Part$ rather than merely to those in Tot. As was described earlier, the "holes" in partial recursive functions avoid the diagonal argument just rehearsed— the same argument that showed that some intuitively effectively computable functions are not primitive recursive.

To index $Part$, you simply need to add a single clause for minimization to the earlier indexing of $Prim$. The arity of the function minimalized should be one plus the arity k of the function we want to obtain. To reflect the fact that the indexed functions may end up being partial, we write ϕ_x^k instead of f_x^k.

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

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

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

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

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

$$lh(x) \geq 5 \quad \Rightarrow \quad \phi_x^k = M(\phi_{(x)_0}^{k+1}).$$

4.7 The "Universal Machine" Theorem

In the case of primitive recursive functions, the function

$$h(n, x_1, \ldots, x_k) = f_n^k(x_1, \ldots, x_k)$$

is not primitive recursive, (recall exercise 3.6). Since the effectiveness of the enumeration implies that h is intuitively effective, this showed that $Prim$ doesn't contain all intuitively computable, total functions. If $Part$ is to contain all

intuitively effective partial functions, then it had better be the case that the partial function

$$\upsilon(n, x_1, \ldots, x_k) \simeq \phi_n^k(x_1, \ldots, x_k)$$

is partial recursive. Better yet, one would like a single "interpreter" that can handle argument lists of all arities, by means of sequence coding:

$$\upsilon(n, x) \simeq \phi_n^{lh(x)}((x)_0, \ldots, (x)_{lh(n)\dot{-}1}).$$

The arity parameter k is dropped because the code number of the argument list effectively "tells" the program how many inputs to expect. The function υ is called the *universal function* for the indexing $\phi_{\_}^k$ of $Part$, since it can simulate the performance of ϕ_n^k if it is given the index n.

In computer science, programming languages are implemented in one of two ways. "Compiled" languages are translated directly into the machine code that runs directly on the hardware in question. "Interpreted" languages are not translated into machine code. They are "simulated", step by step, by a program written in machine code. Sometimes an interpreter is called a "virtual machine". It is amusing to consider that the universal function is actually an interpreter written in our partial recursive formalism that interprets a bizarre programming language in which each program is simply a natural number.

It remains to show that the universal function is partial recursive. Intuitively, interpret quaternary relation $U(n, t, y, x)$ to say that the computation directed by a program with index n returns output y on input sequence $((x)_0, \ldots, (x)_{lh(x)\dot{-}1})$ after consuming no more than t "resources". Under this interpretation, it follows that for each n, t, y, x,

$$\phi_n^k((x)_0, \ldots, (x)_{lh(x)\dot{-}1}) \simeq y \iff (\exists t)U(n, t, y, x),$$

for a computation that halts always halts after some finite "resources". A relation satisfying the preceding biconditional is said to be *universal* for the indexing $\phi_{\_}^k$. Universal relations are also called *Kleene* predicates.

Given a universal relation for $\phi_{\_}^k$, it is easy to define the universal function ψ by dovetailing the search for the output y with the search for a suitable resource bound t and then returning the component that represents the output.

$$\upsilon(n, x) \simeq ((\mu z)U(n, (z)_0, (z)_1, x)_1.$$

Since a resource bound and output exist if the computation halts with that output, and since the predicate will be shown to be primitive recursive, the minimization is guaranteed to find the pair and return the correct output. If the simulated computation never halts, there is no output, so the minimization runs forever.

It remains only to exhibit a total recursive characteristic function for

$$U(n, t, y, x).$$

In fact, something stronger is true: there exists a *primitive recursive* universal relation for the indexing $\phi_{\_}^k$. In the following definition, the "resource bound"

will concern the sizes of code numbers of tuples of outputs of intermediate computations. The tuples will be counted "simultaneously" in the case of composition and "diachronically" in the case of recursion and minimization. Since only minimization can produce new "infinite loops", bounding it's travels by t is the essential trick in the construction. Strictly speaking, this is a course-of-values recursion (on which variable?), so aren't you happy you already know that course-of-values recursion is a primitive recursive operator?

$$\begin{aligned} U(n,t,y,i) \quad = \quad & [lh(n) = 0 \wedge y = 0] \vee \\ & [lh(n) = 1 \wedge (lh(i) = 0 \rightarrow y = 0) \wedge (lh(i) > 0 \rightarrow y = (i)_0 + 1)] \vee \\ & [lh(n) = 2 \wedge \ldots \text{your ideas here}] \vee \\ & [lh(n) = 3 \wedge (\exists z \leq t)[lh(z) = lh((n)_1) \wedge \\ & (\forall w < lh((n)_1))U(((n)_1)_w, t, (z)_w, i) \wedge U((n)_0, t, y, z)] \vee \\ & [lh(n) = 4 \wedge \ldots \text{your ideas here}] \vee \\ & [lh(n) = 5 \wedge \ldots \text{your ideas here}] \vee \\ & [lh(n) > 5 \wedge \ldots \text{your ideas here}] \vee \end{aligned}$$

Exercise 4.2 *Complete the preceding definition.*

So it has been shown that:

Proposition 4.2 ***(Kleene Normal Form)*** *There exists a primitive recursive relation U such that for each n, x,*

$$\phi_n^{lh(x)}((x)_0, \ldots, (x)_{lh(x) \dot{-} 1}) \simeq ((\mu z)\ U(n, (z)_0, (z)_1, x))_1.$$

The right-hand-side of the equation is said to be in Kleene Normal Form. The theorem says that every partial recursive function can be written in Kleene normal form. Kleene normal form is interesting, because the full power of minimization can be obtained by applying it just once over a primitive recursive relation. Complicated interleaving of minimization with recursion and composition can safely be ignored!

Not only that. The definition of U is very simple in the sense that its derivation tree doesn't involve many primitive recursions. The moral is that minimization is an extremely powerful construction. Adding to it just a tiny bit of "seed material" from primitive recursion yields all partrial recursive functions.

The universal machine theorem is a weak corollary of the Kleene normal form theorem.

Proposition 4.3 ***(Universal Machine Theorem)***

$$(\exists u)(\forall n, k, x)\ (\phi_n^{lh(x)}((x)_0, \ldots, (x)_{lh(x) \dot{-} 1}) = \phi_u^2(n, \langle \vec{x} \rangle)).$$

Proof. Just set $\phi_u^2(n, \langle\vec{x}\rangle) = ((\mu z)U(n, (z)_0, (z)_1, \langle\vec{x}\rangle))_1$, and observe that the right-hand side is a partial recursive derivation tree. $\dashv$

4.8 The *s-m-n* Theorem

Despite the nearly kinky name, the *s-m-n* theorem is a pretty tame fact. You already know how to show that total recursive functions are closed under substitution of constants, for given binary partial recursive function $\phi_i^2(x, y)$,and a natural number n, the function

$$\begin{aligned}\psi(x) &= \phi_i^2(n, x)\\ &= \phi_i^2(c_n(x), x)\\ &= C(\phi_i^2, c_n, p_1^1).\end{aligned}$$

is partial recursive. So since every partial recursive function has an index, there exists j such that $\psi(x) = \phi_j(x)$. But notice that this construction is "non-uniform" in the sense that we haven't *effectively computed* j from i and n.

In a modern, functional programming language, one simply substitutes constants for variables in an argument list, so if $M(x, y)$ is the given program, the new program that results when 25 is put in for x is just $M(25, y)$. In such a language, that is the (almost trivial) procedure for finding a program that does what a given program would have had a given argument been given. But when your programs are just *numbers*, it isn't *that* easy. In our "programming language", a program is just a number i. One can write $\phi_i(25, y)$, but that is not a program in our programming system; it is an open formula of mathematics. Our version of $M(25, y)$ is a partial recursive *index* j such that $\phi_j(y) = \phi_i(25, y)$. So the problem of effectively substituting a constant into a program amounts in our system to finding a total recursive function $s(i, n)$ such that:

$$\phi_{s(i,n)}^1(x) = \phi_i^2(n, x).$$

More generally, you could simultaneously substitute constants for $m \geq 0$ out of $m+n \geq 0$ variables in an $m+n$-ary partial recursive function. The substitution procedure has to know what m and n are in order to know what program to produce, so it is called s_m^n.

Proposition 4.4 series*s-m-n* Theorem *Let $m, n \geq 0$. There exists a primitive recursive function $s_m^n(n, \vec{x})$, where $\vec{x}$ is m-ary, such that for each n-ary $\vec{y}$,*

$$\phi_{s_m^n(i,\vec{x})}^n(\vec{y}) \simeq \phi_i^{m+n}(\vec{x}, \vec{y}).$$

Notice that nothing here precludes $m = n = 0$, in which case:

$$\phi_{s_0^0(i)}^n() \simeq \phi_i^0(),$$

so s_0^0 function is just the identity function, since substituting no constants in an empty argument list doesn't change the zero-ary function given. More interestingly, our indices for zero-ary functions allow us to substitute constants for *all* variables to produce an index of a zero-ary function.

$$\phi^0_{s^0_m(i,\vec{x})}() \simeq \phi^m_i(\vec{x}).$$

Hence, computing $s^0_m(i,\vec{n})$ denotes the formal result of providing inputs $\vec{n}$ to program i without actually starting the computation. Maybe an output will be produced and maybe not. On the other hand, $\phi_i(\vec{n})$ denotes the outcome of this computation, if there is one, which is another matter entirely.

This prosaic business about a procedure for substituting constants into a function turns out to be extremely important, for the the universal theorem and the *s-m-n* property essentially codify everything about our indexing that is essential for computability theory. So by simply assuming the universal and *s-m-n* theorems as axioms, you can ditch all the fussy details once and for all! But not until after you prove it for our numbering and show that it has the pivotal significance just claimed for it.

So how does one prove it? Well, you want:

$$\phi^n_{s^n_m(i,\vec{x})}(\vec{y}) \simeq \phi^{m+n}_i(c_{x_1}(p^1_n(\vec{y})),\ldots,c_{x_m}(p^1_n(\vec{y})),p^1_n(\vec{y}),\ldots,p^n_n(\vec{y})).$$

First, you have to define a primitive recursive function *proj* such that for all n-ary $\vec{x}$,

$$\phi^n_{proj(i,n)}(\vec{x}) \simeq x_i.$$

But that's easy, since the indexing ϕ^k_- doesn't care about arity:

$$\begin{aligned} proj(i,n) &= proj(i) \\ &= \langle i,0\rangle. \end{aligned}$$

Next, you need to be able to compute the index of a constant function from the constant. Easy, but annoying.

Exercise 4.3 *Define a primitive recursive function const such that for all x,*

$$\phi^n_{const(n)}(x) \simeq n.$$

And you have to write a primitive recursive program that knows how to return the index of a composition from the indices of the functions involved.

Exercise 4.4 *Define a primitive recursive function comp such that for all j, and m-ary i,*

$$\phi^k_{comp(j,\vec{\imath})} = C(\phi^k_{i_1},\ldots,\phi^k_{i_m}).$$

Exercise 4.5 *Now it's easy to use the above to define $s^n_m(i,\vec{x})$.*