9 Primitive recursive functions

As just announced in the Interlude, the primitive recursive functions form a major subclass of the effectively computable functions. This chapter explains which functions these are, and proves some elementary results about them.

9.1 Introducing the primitive recursive functions

Let's start by revisiting the basic axioms for *addition* and *multiplication*. We adopted formal versions in Q and PA. But now, throughout this chapter, we will be doing everyday informal mathematics. So – in keeping with our notational convention, §4.1 – here are the axioms re-presented informally, with the variables running over the natural numbers, but following the common practice of leaving quantifiers to be understood:

$$x + 0 = x$$
$$x + Sy = S(x + y)$$

$$x \times 0 = 0$$
$$x \times Sy = (x \times y) + x$$

The first of the pair of equations for addition tells us the result of adding zero to any number x. The second tells us the result of adding Sy in terms of the result of adding y. Hence these equations – as we pointed out before – together tell us how to add any of $0, S0, SS0, SSS0, \ldots$, i.e. they tell us how to add *any* number to a given number x. Similarly, the first of the pair of equations for multiplication tells us the result of multiplying by zero. The second equation tells us the result of multiplying by Sy in terms of the result of multiplying by y and doing some addition. Hence the equations for multiplication and addition taken together tell us how to multiply a given number x by any of $0, S0, SS0, SSS0, \ldots$, i.e. they tell us how to multiply by *any* number.

Let's have two more very elementary examples. Take, then, the *factorial* function $y!$, which can be defined by the following two equations:

$$0! = 1$$
$$(Sy)! = y! \times Sy$$

The first equation assigns a value to the factorial function for the argument 0; the second equation tells us how to work out the value of the function for Sy

once we know its value for y (assuming we already know about multiplication). By laboriously applying and reapplying the second equation, we can successively calculate 1!, 2!, 3!, 4! . . . , as follows:

$$\begin{aligned}&1! = 0! \times 1 = 1\\&2! = 1! \times 2 = 2\\&3! = 2! \times 3 = 6\\&4! = 3! \times 4 = 24\end{aligned}$$

And so on and on it goes. Our two-equation definition is indeed properly called a definition because it fixes the value of 'y!' for all numbers y.

For our next example – this time another two-place function – consider the *exponential*. Using the standard notation, this can be defined by a similar pair of equations:

$$\begin{aligned}&x^0 = S0\\&x^{Sy} = (x^y \times x)\end{aligned}$$

The first equation gives the function's value for a given value of x when $y = 0$, and – keeping x fixed – the second equation gives the function's value for the argument Sy in terms of its value for y (still assuming we already know about multiplication). The equations determine, e.g., that $3^4 = 3 \times 3 \times 3 \times 3 = 81$.

Of course, you knew all that! But we don't need more exotic examples to introduce the points we need:

i. Note that, in each of these definitional pairs of equations, the second one fixes the value of a function for argument Sy by invoking the value of the *same* function for argument y. A procedure where we evaluate a function for one input by calling the *same* function for a smaller input is standardly termed 'recursive' – and the particularly simple pattern we've illustrated is called, more precisely, 'primitive recursive'. So our two-clause definitions are examples of *definition by primitive recursion.*[1]

ii. Next note, for example, that $(Sy)!$ is defined as $y! \times Sy$, so it is evaluated by evaluating $y!$ and Sy and then feeding the results of these computations into the multiplication function. This involves, in a word, the *composition* of functions, where evaluating a composite function involves taking the output(s) from one or more functions, and treating these as inputs to another function.

iii. Our four examples can then be arranged into two short *chains* of definitions by recursion and functional composition. Working from the bottom up, addition is defined in terms of the successor function; multiplication is

[1] "Surely, defining a function in terms of that very same function is circular!" But think of the relevant function as being defined in successive stages. The value of the function for input 0 is given. For each n in turn, the value of the function for Sn is then fixed by its already-settled value for n. So – by induction! – the function thereby does indeed get a determinate value for every numerical input.

then defined in terms of successor and addition; then the factorial (or, in the second chain, exponentiation) is defined in terms of multiplication and successor.

With these simple motivating thoughts in mind, here is our first, quick-and-dirty, way of specifying the p.r. functions:

Defn. 24. *Roughly: a primitive recursive function is one that can be similarly characterized using a chain of definitions by recursion and composition, starting from trivial 'initial functions' like the successor function.*[2]

9.2 Defining the p.r. functions more carefully

We must now do better. (a) We need to tidy up the idea of defining a function by primitive recursion. (b) We need to tidy up the idea of defining a new function by composing old functions. And (c) we need to say more about the 'starter pack' of initial functions which we can use in building up a chain of definitions by primitive recursion and/or composition.

(a) Consider the recursive definition of the factorial again:

$$0! = 1$$
$$(Sy)! = y! \times Sy$$

This is an example of the following general scheme for defining a one-place function f:

$$f(0) = g$$
$$f(Sy) = h(y, f(y))$$

Here, g is just a number, while h is a two-place function which – crucially – *we are assumed already to know about* prior to the definition of f. Maybe that's because h is an 'initial' function that we are allowed to take for granted; or maybe it's because we've already given recursion clauses to define h; or maybe h is a composite function constructed by plugging one known function into another – as in the case of the factorial, where $h(y, z) = z \times Sy$ (where we take the output from the successor function as one input into the multiplication function).

Likewise, with a bit of massaging, the recursive definitions of addition, multiplication and the exponential can all be treated as examples of the following general scheme for defining two-place functions:

$$f(x, 0) = g(x)$$
$$f(x, Sy) = h(x, y, f(x, y))$$

where now g is a one-place function, and h is a three-place function, both of them functions that we already know about. Three points about this:

[2]The basic idea is there in Dedekind and highlighted by Skolem in 1923. But the modern terminology 'primitive recursion' seems to be due to Rósza Péter in 1934; and 'primitive recursive function' was first used by Stephen Kleene in 1936. I'll use the abbreviation 'p.r.': but note that the same abbreviation is quite often used elsewhere as short for '*partial* recursive' – a quite different notion!

i To get the definition of addition to fit this pattern, with a unary function on the right of the first equation, we take $g(x)$ to be the trivial *identity function* $I(x) = x$.

ii To get the definition of multiplication to fit the pattern, $g(x)$ has to be treated as the equally trivial *zero function* $Z(x) = 0$.

iii How do we get the definition of addition $x + Sy = S(x + y)$ to fit the pattern $f(x, Sy) = h(x, y, f(x, y))$?

We have to take $h(x, y, z)$ to be the function Sz. So, as this illustrates, we must allow h not to care what happens to some of its arguments, while operating on some other argument(s). The conventional way of doing this is to help ourselves to some further trivial identity functions that serve to select out particular arguments. For example, the function I^3_3 takes three arguments, and just returns the third of them, so $I^3_3(x, y, z) = z$. Then, in the definition of addition, we can put $h(x, y, z) = SI^3_3(x, y, z)$, so h is defined by composition from initial functions which we can take for granted.

We can now generalize the idea of a definition by recursion from the case of one-place and two-place functions to cover the case of many-place functions. There's a standard notational device that helps to put things snappily: we write $\vec{x}$ as short for the array of k variables $x_1, x_2, \ldots, x_k$ (taking the relevant k to be fixed by context). Then we can say:

Defn. 25. *Suppose that the following holds:*

$$f(\vec{x}, 0) = g(\vec{x})$$
$$f(\vec{x}, Sy) = h(\vec{x}, y, f(\vec{x}, y))$$

Then f is defined from g and h by primitive recursion.

This covers the case of one-place functions $f(y)$ like the factorial if we allow $\vec{x}$ to be empty, in which case $g(\vec{x})$ is a 'zero-place function', i.e. a constant.

(b) Now to tidy up the idea of definition by composition. The basic idea, to repeat, is that we form a composite function f by treating the output value(s) of one or more given functions g, g', g'', ..., as the input argument(s) to another function h. For example, we set $f(x) = h(g(x))$. Or, to take a slightly more complex case, we could set $f(x, y, z) = h(g(x, y), g'(y, z))$.

There's a number of equivalent ways of covering the manifold possibilities of composing multi-place functions. But one simple way is to define what we might call one-at-a-time composition (where we just plug *one* function g into another function h), thus:

Defn. 26. *If $g(\vec{y})$ and $h(\vec{x}, u, \vec{z})$ are functions – with $\vec{x}$ and $\vec{z}$ possibly empty – then* f is defined by composition by substituting g into h *just if* $f(\vec{x}, \vec{y}, \vec{z}) = h(\vec{x}, g(\vec{y}), \vec{z})$.

We can then think of generalized composition – where we plug more than one function into another function – as just iterated one-at-a-time composition. For example, we can substitute the function $g(x, y)$ into $h(u, v)$ to define the function $h(g(x, y), v)$ by composition. Then we can substitute $g'(y, z)$ into the defined function $h(g(x, y), v)$ to get the composite function $h(g(x, y), g'(y, z))$.

(c) Now, the quick-and-dirty Defn. 24 tells us that the primitive recursive functions are built up by recursion and composition, beginning from some 'starter pack' of trivial basic functions.

But which functions will count as basic? We have in fact met all the ones we need:

Defn. 27. *The initial functions are the successor function S, the zero function $Z(x) = 0$ and all the k-place identity functions, $I_i^k(x_1, x_2, \ldots, x_k) = x_i$ for each k, and for each i, $1 \leq i \leq k$.*

These identity functions are also often called *projection* functions (they 'project' the vector with components $x_1, x_2, \ldots, x_k$ onto the i-th axis).

Given (a) to (c), we can now put everything together, and give the following official account:

Defn. 28. *The p.r. functions are the following:*

1. *The successor function S, zero function Z, and all the identity functions I_i^k are p.r.;*
2. *if f can be defined from the p.r. functions g and h by composition, substituting g into h, then f is p.r.;*
3. *if f can be defined from the p.r. functions g and h by primitive recursion, then f is p.r.;*
4. *nothing else is a p.r. function.*

(We allow g in clauses (2) and (3) to be zero-place, i.e. be a constant.)

A p.r. function f is therefore one that *can* be specified by a chain of definitions by recursion and composition, leading back to initial functions. Which accords with the informal characterization Defn. 24.

Let's have another quick example of such a chain, this time to show that the absolute difference function is primitive recursive. So consider:

$$P(0) = 0$$
$$P(Sx) = x$$

$$x \dot{-} 0 = x$$
$$x \dot{-} Sy = P(x \dot{-} y)$$

$$|x - y| = (x \dot{-} y) + (y \dot{-} x)$$

Here, 'P' signifies the predecessor function (with zero being treated as its own predecessor) – and its two-part definition is a limiting case of a definition by primitive recursion according to our official account (why?). Next, '$\dot{-}$' signifies 'subtraction with cut-off', so $m \dot{-} n$ is zero if $m < n$. Then $|m - n|$ is the absolute difference between m and n.

9.3 How to prove a result about all p.r. functions

Let's introduce an obvious bit of terminology:

Defn. 29. *A* definition chain *for the p.r. function f is a sequence of functions $f_0, f_1, f_2, \ldots, f_k$ where each f_j is either an initial function or is defined from previous functions in the sequence by composition or recursion, and $f_k = f$.*

The closure condition (4) in the previous definition means that every p.r. function is required to have a definition chain in this sense (the chain need not be unique, but a function must have at least one to be p.r.). Which means in turn that there is a simple method of proving that every p.r. function shares some feature. For suppose that, for some given property P, we can show the following:

P1. The initial functions have property P.

P2. If the functions g and h have property P, and f is defined by composition from g and h, then f also has property P.

P3. If the functions g and h have property P, and f is defined by primitive recursion from g and h, then f also has property P.

It then follows that all primitive recursive functions have property P.

Why? Take any p.r. function f. It must have a definition chain. Now trek along such a chain for f. Each initial function we encounter has property P by P1. By P2 and P3, each definition by recursion or composition which is used in the chain takes us from functions which have property P to another function with property P. So, every function we define as we go along the chain has property P, including the final target function f.

In sum, then: to prove that all p.r. functions have some property P, it suffices to prove the relevant versions of P1, P2 and P3.

For a very simple (but important!) first example, take the property of being a *total function* of the natural numbers, i.e. being a function which successfully outputs a natural number value for any given numerical input. The initial functions are, trivially, total functions of numbers, defined for every numerical argument; also, primitive recursion and composition both build total functions out of total functions (why? check this claim!). Which means that p.r. functions are always total functions, defined for all natural number arguments.

9.4 The p.r. functions are computable

We now show that every p.r. function is effectively computable. And we'll take the discussion in stages.

(a) Given the general strategy just described, it is enough to show:

C1. The initial functions are computable.

C2. If f is defined by composition from computable functions g and h, then f is also computable.

C3. If f is defined by primitive recursion from the computable functions g and h, then f is also computable.

For C1 it is enough to remark that the trivial initial functions S, Z, and I_i^k are computable if any are! For C2, note that to compute the composition of two computable functions g and h you just feed the output from whatever algorithmic routine evaluates g as input into the routine that evaluates h.

To illustrate C3, return once more to our example of the factorial. Here is its p.r. definition again,

$$0! = 1$$
$$(Sy)! = y! \times Sy$$

The first clause gives the value of the function for the argument 0; then – as we said – you can repeatedly use the second recursion clause to calculate the function's value for $S0$, then for $SS0$, $SSS0$, etc. So the definition in fact encapsulates an algorithm for effectively calculating the function's value for each successive number n (given that we already know how to compute multiplications).

This last observation evidently generalizes to establish C3.

(b) But there's more to be said. Let's think again about the kind of algorithm needed to compute $n!$ (for $n > 0$, and assuming that we can already handle multiplication). We need a computational routine which takes n as input, and proceeds like this:

```
1. Set the variable fact to the initial value 1.
2. Set the counter $y$ to the initial value 0 and enter a looping routine.
3. if $y$ equals $n$ then exit the loop, else
       compute $fact \times Sy$, and make the output the new value of fact.
4. Increment the value of $y$ by 1, and go back to (3).
```

The routine terminates when the counter reaches n with the variable *fact* now having the value $n!$.

In snappier summary form, we can set out the routine something like this:

```
1. $fact := 1$
2. For $y = 0$ to $n - 1$
3.     $fact := (fact \times Sy)$
4. Next $y$
```

And note: the crucial thing about executing this kind of loop is that the total number of iterations to be run through is fixed in advance of entering the loop (or at least, a maximum bound is set in advance, if we allow for early exits). It will be useful to have a label for such bounded-in-advance looping structures: let's call them *basic 'for' loops.*[3]

[3]Two reasons. Such loops are indeed a basic computational structure. And they are instantiated by loops introduced by the keyword 'for' – as in our snippet of code – in some early languages like Basic.

Now, our mini-program for the factorial calls the multiplication function which can itself be computed by a similar basic 'for' loop (invoking addition). And addition can be computed by another such 'for' loop (invoking the successor). So reflecting the downward chain of recursive definitions

$$\text{factorial} \Rightarrow \text{multiplication} \Rightarrow \text{addition} \Rightarrow \text{successor}$$

there will be a composite program for the factorial containing *nested* basic 'for' loops, which ultimately calls primitive operations like incrementing the contents of a variable or setting a variable to zero.

The point obviously generalizes, giving us

Theorem 25. *Primitive recursive functions are effectively computable by a program which invokes a series of (possibly nested) basic 'for' loops.*

(c) The crucial thing here is that each of the looping procedures required in computing a p.r. function involves a fixed-at-the-point-of-entry bound to the number of circuits round the loop.

For a contrasting sort of computational procedure, just recall our proof of Theorem 6, about the decidability of negation-complete, effectively decidable, theories. There, we allowed the process *enumerate the theorems and wait to see which of φ or $\neg\varphi$ turns up* to count as a computational decision procedure. Which illustrates that we do also permit procedures involving open-ended searches to count as effective computations, even if there is no prior bound given on the length of search.

In other words, we allow for computations which involve 'do while' loops, where a routine can be repeatedly iterated while some condition continues to be fulfilled, as many times as it takes, with no prior bound set.[4]

(d) The converse of the last theorem is also true. Suppose we have a program which sets a value for $f(0)$, and then goes into a basic 'for' loop which computes the value of a one-place function $f(n)$ (for $n > 0$), a loop which calls on some already-known function(s) which are used on loop number y (counting from zero) to fix the value of $f(Sy)$ in terms of the value of $f(y)$. This plainly corresponds to a definition by recursion of f.

Generalizing:

Theorem 26. *If a function can be computed by a program without open-ended searches – using just basic (i.e. bounded-in-advance) 'for' loops for iterative procedures and with the program's 'built in' functions all being p.r. – then the newly defined function will also be primitive recursive.*

[4]It would be misleading to baldly contrast 'for' loops with 'do while' loops. For a start, a basic 'for' loop can be thought of as a special kind of 'do while' loop where we increment a loop-counter on each iteration and continue for a fixed number of cycles round the loop while the counter has yet to reach its given target. And conversely, in many computer languages, loops introduced by the keyword 'for' can be open-ended.

What matters, then, is not the particular keywords used, but – to re-emphasize – the difference between looping structures where the number of iterations is given a fixed numerical bound in advance, and those where it isn't.

In fact we can elaborate this a bit to explicitly allow e.g. conditionally branching computations (as long as the test condition for the branching is itself p.r. computable) – but we needn't fuss about that here.

This gives us a quick way of convincing ourselves that a new function is p.r.: sketch out a routine for computing it and check that the needed looping computations only invoke already known p.r. functions and the number of iterations in any loop is always bounded in advance, so we do not need to set off on any open-ended searches. Then the new function will be primitive recursive.

For a quick example, take the two-place function $gcd(x, y)$ which outputs the greatest common divisor of the two inputs. Evidently, a bounded search through cases is enough to do the trick: at its crudest and most inefficient, we can look in turn at all the numbers up to (and including) the smaller of x and y and see if it divides both. That sketched algorithmic procedure is enough to assure us that $gcd(x, y)$ is p.r. without going through the palaver of actually writing down a suitable definition chain.

9.5 Not all computable numerical functions are p.r.

We have seen that any p.r. function is effectively computable. And most of the ordinary computable numerical functions you already know about from elementary mathematics are in fact primitive recursive.

But to repeat, the values of a given primitive recursive function can be computed by a program involving basic 'for' loops as its main programming structure. Each loop goes through a specified number of iterations, set in advance. On the other hand, as we reminded ourselves, we do count procedures that involve unbounded searches as computations. So it looks very plausible that not everything computable will be primitive recursive.

And in fact we can do better than offer plausibility considerations. We will now prove:

Theorem 27. *There are effectively computable numerical functions which aren't primitive recursive.*

Proof. The p.r. functions are effectively enumerable. That is to say, there is an effective way of numbering off functions f_0, f_1, f_2, ..., such that each of the f_i is p.r., and each p.r. function appears somewhere on the list.

Why? A p.r. function is defined by recursion or composition from other functions which are defined by recursion or composition from other functions which are defined ... ultimately in terms of some primitive starter functions. So choose some standard formal specification language for representing these chains of definitions. Then we can effectively generate all possible strings of symbols from this specification language (arranged by length, then 'in alphabetical order'); and as we go along, we select the strings that obey the rules for being a definition chain for a p.r. function. This generates a list which effectively enumerates the p.r. functions, $f_0, f_1, f_2, f_3, \ldots$, repetitions allowed. So consider the following table:

$$\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & \ldots \\
\hline
f_0 & \underline{f_0(0)} & f_0(1) & f_0(2) & f_0(3) & \ldots \\
f_1 & f_1(0) & \underline{f_1(1)} & f_1(2) & f_1(3) & \ldots \\
f_2 & f_2(0) & f_2(1) & \underline{f_2(2)} & f_2(3) & \ldots \\
f_3 & f_3(0) & f_3(1) & f_3(2) & \underline{f_3(3)} & \ldots \\
\ldots & \ldots & \ldots & \ldots & \ldots & \searrow
\end{array}$$

Down the table we list off the p.r. functions. An individual row then gives the values of a particular f_n for each successive argument. Let's now define the corresponding *diagonal* function, by putting $\delta(n) = f_n(n) + 1$. To compute $\delta(n)$, we just run our effective enumeration of the definition chains for p.r. functions until we get to the recipe for f_n. We follow the instructions in that recipe to evaluate that function for the argument n. We then add one. Each step is entirely mechanical. So our diagonal function is effectively computable, using a step-by-step algorithmic procedure.

By construction, however, the function δ can't be primitive recursive. For suppose otherwise. Then δ must appear somewhere in the enumeration of p.r. functions, i.e. be the function f_d for some index number d. But now ask what the value of $\delta(d)$ is. By hypothesis, the function δ is none other than the function f_d, so $\delta(d) = f_d(d)$. But by the initial definition of the diagonal function, $\delta(d) = f_d(d) + 1$. Contradiction.

So we have, as they say, 'diagonalized out' of the class of p.r. functions to define a new function δ which is still effectively computable but not primitive recursive. ⊠

"But hold on! *Why* is δ not a p.r. function?" Well, consider evaluating $\delta(n)$ for increasing values of n. For each new argument, we will have to evaluate a *different* function f_n for that argument (and then add 1). Evaluating these different functions f_n for input n will call on computations involving loops nested to varying different depths. We have no reason to expect there will be a nice pattern in the successive computations of all the different functions f_n which enables them to be wrapped up into a single p.r. definition. And our diagonal argument in effect shows that this can't be done.

9.6 Defining p.r. properties and relations

We have defined the class of p.r. *functions*. Finally in this chapter, we extend the scope of the idea of primitive recursiveness and introduce the ideas of *p.r. decidable (numerical) properties* and *relations*.

Now, quite generally, we can tie together talk of functions and talk of properties and relations by using the notion of a *characteristic function*:

Defn. 30. *The characteristic function of the numerical property P is the one-place function c_P such that if m is P, then $c_P(m) = 0$, and if m isn't P, then $c_P(m) = 1$.*

The characteristic function of the two-place numerical relation R is the two-place function c_R such that if m is R to n, then $c_R(m, n) = 0$, and if m isn't R to n, then $c_R(m, n) = 1$.

And similarly for many-place relations. The choice of values for the characteristic function is, of course, entirely arbitrary: any pair of distinct numbers would do. Our choice is supposed to be reminiscent of the familiar use of 0 and 1, one way round or the other, to stand in for *true* and *false*. And our selection of 0 rather than 1 for *true* follows Gödel.

The numerical property P partitions the numbers into two sets, the set of numbers that have the property and the set of numbers that don't. Its corresponding characteristic function c_P also partitions the numbers into two sets, the set of numbers the function maps to the value 0, and the set of numbers the function maps to the value 1. And these are the *same* partition. So in a good sense, P and its characteristic function c_P contain exactly the same information about a partition of the numbers: hence we can move between talk of a property and talk of its characteristic function without loss of information. Similarly, of course, for relations (which partition pairs of numbers, etc.). And we can use this link between properties and relations and their characteristic functions in order to carry over ideas defined for functions and apply them to properties/relations.

For example, without further ado, we now extend the idea of primitive recursiveness to cover properties and relations:

Defn. 31. *A p.r. decidable property is a property with a p.r. characteristic function, and likewise a p.r. decidable relation is a relation with a p.r. characteristic function.*

By way of casual abbreviation, we'll fall into saying that p.r. decidable properties and relations are themselves (simply) p.r.

For a quick example, consider the property of being a *prime* number. Take the characteristic function $pr(n)$ which has the value 0 when n is prime, and 1 otherwise. Now just note that we can evidently compute $pr(n)$ just using 'for' loops (we just do a bounded search through numbers less than n – indeed, no greater than $\sqrt{n}$ – and if we find a divisor of n other than 1, return the value 1, and otherwise return the value 0). So the property of being prime is p.r.