

## Primitive recursive functions

Peter Smith

November 6, 2009

In our preamble, it might be helpful this time to give a story about where we are going, rather than (as in previous episodes) review again where we've been. So, at the risk of spoiling the excitement, here's what's going to happen in this and the following three Episodes.

1. The formal theories of arithmetic that we've looked at so far have (at most) the successor function, addition and multiplication built in. Why stop there? Even school arithmetic acknowledges many more numerical functions. This episode describes a very wide class of familiar such functions, the so-called primitive recursive ones. We also define the primitive recursive properties and relations (i.e. those with a p.r. 'characteristic function').
2. The next episode shows that  $L_A$ , the language of basic arithmetic, can express all p.r. functions and relations. Moreover  $Q$  and hence  $PA$  can capture all those functions and relations. So  $PA$ , despite having only successor, addition and multiplication 'built in', can actually deal with a vast range of functions.
3. Then we look at the 'arithmetization of syntax' by Gödel-numbering. We'll do this in particular for  $PA$  and define various properties/relations like this

$Wff(n)$  iff  $n$  is the code number of a PA-wff.

$Sent(n)$  iff  $n$  is the code number of a PA-sentence.

$Prf(m, n)$  iff  $m$  is the code number of a PA-proof of the sentence with code number  $n$ .

Moreover, these properties/relations are primitive recursive.

4. Since  $Prf$  is p.r., and  $PA$  can capture all p.r. relations, there is a wff  $Prf(x, y)$  which captures the relation  $Prf$ . And we'll use this fact – or a closely related one – to construct a Gödel sentence which sort-of-says 'I am not provable in  $PA$ ', and hence prove Gödel first incompleteness theorem for  $PA$ .

We'll then want to generalize our result beyond  $PA$ , and finally get back to something like the grand sweeping claims about incompleteness stated in Episode 1. *Now read on . . .*

## 14 Introducing the primitive recursive functions

We'll start with two more functions that are familiar from elementary arithmetic. Take the factorial function  $y!$ , where e.g.  $4! = 1 \times 2 \times 3 \times 4$ . This can be defined by the following two equations:

$$0! = S0 = 1$$

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

The first clause tells us the value of the function for the argument  $y = 0$ ; the second clause 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). So by applying and reapplying the second clause, we can successively calculate  $1!$ ,  $2!$ ,  $3!$ , . . . . Hence our two-clause definition fixes the value of ' $y!$ ' for all numbers  $y$ .

For our second example – this time a two-place function – consider the exponential, standardly written in the form ' $x^y$ '. This can be defined by a similar pair of equations:

$$x^0 = S0$$

$$x^{Sy} = (x^y \times x)$$

Again, the first clause gives the function's value for a given value of  $x$  and  $y = 0$ , and – keeping  $x$  fixed – the second clause gives the function's value for the argument  $Sy$  in terms of its value for  $y$ .

We've seen this two-clause pattern before, of course, in our formal Axioms in Q/PA for the addition and multiplication functions. Presented in the style of everyday informal mathematics (leaving quantifiers to be understood) – and note, everything in this episode *is* just informal mathematics – we have:

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

Three comments about our examples so far:

- i. In each definition, the second clause fixes the value of a function for argument  $Sn$  by invoking the value of the *same* function for argument  $n$ . This kind of procedure where we evaluate a function by calling the same function is standardly termed 'recursive' – or more precisely, 'primitive recursive'. So our two-clause definitions are examples of *definition by primitive recursion*.<sup>1</sup>
- ii. Note, for example, that  $(Sn)!$  is defined as  $n! \times Sn$ , so it is evaluated by evaluating  $n!$  and  $Sn$  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 series of examples illustrates 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 then defined in terms of successor and addition; then the factorial (or, on the other chain, exponentiation) is defined in terms of multiplication and successor.

Here's another little definitional chain:

$$\begin{aligned}P(0) &= 0 \\P(Sx) &= x \\x \dot{-} 0 &= x \\x \dot{-} Sy &= P(x \dot{-} y) \\|x - y| &= (x \dot{-} y) + (y \dot{-} x)\end{aligned}$$

' $P$ ' signifies the predecessor function (with zero being treated as its own predecessor); ' $\dot{-}$ ' signifies 'subtraction with cut-off', i.e. subtraction restricted to the non-negative integers (so  $m \dot{-} n$  is zero if  $m < n$ ). And  $|m - n|$  is of course the absolute difference between  $m$  and  $n$ . This time, our third definition doesn't involve recursion, only a simple composition of functions.

These examples motivate the following initial gesture towards a definition:

A *primitive recursive function* is one that can be similarly characterized using a chain of definitions by recursion and composition.<sup>2</sup>

That is a quick-and-dirty characterization, though it should be enough to get across the basic idea. Still, we really need to pause to do better. In particular, we need to nail down more carefully the 'starter pack' of functions that we are allowed to take for granted in building a definitional chain.

<sup>1</sup>Strictly speaking, we need a proof of the claim that primitive recursive definitions really do well-define functions: such a proof was first given by Richard Dedekind in 1888.

<sup>2</sup>The basic idea is there in Dedekind and highlighted by Skolem in 1923. But the modern terminology 'primitive recursion' seems to be due to Rózsa Péter in 1934; and 'primitive recursive function' was first used by Stephen Kleene in 1936.

## 15 Defining the p.r. functions more carefully

On the one hand, do read this section! On the other hand, don't get lost in the techie details. All we are trying to do here is give a careful, explicit, presentation of the ideas we've just been sketching.

### 15.1 Definition by primitive recursion – one and two place functions

Consider the recursive definition of the factorial again:

$$\begin{aligned}0! &= 1 \\ (Sy)! &= y! \times Sy\end{aligned}$$

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

$$\begin{aligned}f(0) &= g \\ f(Sy) &= h(y, f(y))\end{aligned}$$

Here,  $g$  is just a number, while  $h$  is – crucially – a function 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 like the successor function; or perhaps it's because we've already given recursion clauses to define  $h$ ; or perhaps  $h$  is a composite function constructed by plugging one known function into another – as in the case of the factorial, where  $h(y, u) = u \times Sy$ .

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:

$$\begin{aligned}f(x, 0) &= g(x) \\ f(x, Sy) &= h(x, y, f(x, y))\end{aligned}$$

where now  $g$  and  $h$  are both functions that we already know about. Three points about this:

- i. To get the definition of addition to fit this pattern, we have to 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 even more trivial zero function  $Z(x) = 0$ .
- iii. Again, to get the definition of addition to fit the pattern, we have to take  $h(x, y, u)$  to be the function  $Su$ . As this illustrates, we must allow  $h$  not to care what happens to some of its arguments. One neat way of doing this is to help ourselves to some further trivial identity functions that serve to select out particular arguments. Suppose, for example, we have the three-place function  $I_3^3(x, y, u) = u$  to hand. Then, in the definition of addition, we can put  $h(x, y, u) = SI_3^3(x, y, u)$ , so  $h$  is defined by composition from previously available functions.

### 15.2 The initial functions

With that motivation, we will now officially define the full 'starter pack' of functions as follows:

**Defn. 23.** *The initial functions are the successor function  $S$ , the zero function  $Z(x) = 0$  and all the  $k$ -place identity functions,  $I_i^k(x_1, x_2, \dots, x_k) = x_i$  for each  $k$ , and for each  $i$ ,  $1 \leq i \leq k$ .*

The identity functions are also often called *projection* functions. They 'project' the vector with components  $x_1, x_2, \dots, x_k$  onto the  $i$ -th axis.

### 15.3 Definition by primitive recursion – generalizing

We next want to generalize the idea of recursion from the case of one-place and two-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  $x_1, x_2, \dots, x_k$ . Then we can generalize as follows:

**Defn. 24.** *Suppose that the following holds:*

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

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.

### 15.4 Definition by composition

We need 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  $g, g', g'', \dots$  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 compounding multi-place functions. But one standard 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. 25.** *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  $h(g(x, y), g'(y, z))$ .

### 15.5 Putting everything together

We informally defined the primitive recursive (henceforth, p.r.) functions as those that can be defined by a chain of definitions by recursion and composition. Working backwards down a definitional chain, it must bottom out with members of an initial 'starter pack' of trivially simple functions. At the outset, we highlighted the successor function among the given simple functions. But we've since noted that, to get our examples to fit our official account of definition by primitive recursion, we need to acknowledge some other, even more trivial, initial functions. So putting everything together, let's now offer this more formal characterization:

**Defn. 26.** *The p.r. function are as follows*

1. *The initial functions  $S, Z$ , and  $I_i^k$  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.) Note, by the way, that the initial functions are total functions of numbers, defined for every numerical argument; also, primitive recursion and composition both build total functions out of total functions. Which means that all p.r. functions are total functions, defined for all natural number arguments.

## 16 The p.r. functions are computable

### 16.1 The basic argument

To repeat, a p.r. function  $f$  is one that *can* be specified by a chain of definitions by recursion and composition, leading back ultimately to initial functions. But (a) it is trivial that the initial functions  $S, Z$ , and  $I_i^k$  are effectively computable by a simple algorithm. (b) The composition of two computable functions  $g$  and  $h$  is computable (you just feed the output from whatever algorithmic routine evaluates  $g$  as input into the routine that evaluates  $h$ ). And (c) – the key observation – if  $g$  and  $h$  are algorithmically computable, and  $f$  is defined by primitive recursion from  $g$  and  $h$ , then  $f$  is computable too. So as we build up longer and longer chains of definitions for p.r. functions, we always stay within the class of effectively computable functions.

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

$$\begin{aligned} 0! &= 1 \\ (Sy)! &= y! \times Sy \end{aligned}$$

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 encapsulates an algorithm for calculating the function's value for any number, and corresponds exactly to a certain simple kind of computer routine.

### 16.2 Computing p.r. functions by 'for'-loops

Compare our p.r. definition of the factorial with the following schematic program:

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

Here  $fact$  is a memory register that we initially prime with the value of  $0!$ . Then the program enters a loop: and the crucial thing about executing a 'for' loop is that the total number of iterations to be run through is fixed in advance: we number the loops from 0, and in executing the loop, you increment the counter by one on each cycle. So in this case, on loop number  $k$  the program replaces the value in the register with  $Sk$  times the previous value (we'll assume the computer already knows how to find the successor of  $k$  and do that multiplication). When the program exits the loop after a total of  $n$  iterations, the value in the register  $fact$  will be  $n!$ .

More generally, for any one-place function  $f$  defined by recursion in terms of  $g$  and the computable function  $h$ , the same program structure always does the trick for calculating  $f(n)$ . Thus compare

$$\begin{aligned} f(0) &= g \\ f(Sy) &= h(y, f(y)) \end{aligned}$$

with the corresponding program

1.  $func := g$
2. For  $y = 0$  to  $n - 1$
3.      $func := h(y, func)$
4. Loop

So long as  $h$  is already computable, the value of  $f(n)$  will be computable using this 'for' loop that terminates with the required value in the register  $func$ .

Similarly, of course, for many-place functions. For example, the value of the two-place function defined by

$$\begin{aligned} f(x, 0) &= g(x) \\ f(x, Sy) &= h(x, y, f(x, y)) \end{aligned}$$

is calculated by the algorithmic program

1.  $func := g(m)$
2. For  $y = 0$  to  $n - 1$
3.  $func := h(m, y, func)$
4. Loop

which gives the value for  $f(m, n)$  so long as  $g$  and  $h$  are computable.

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

factorial  $\Rightarrow$  multiplication  $\Rightarrow$  addition  $\Rightarrow$  successor

there’s a program for the factorial containing nested ‘for’ loops, which ultimately calls the primitive operation of incrementing the contents of a register by one (or other operations like setting a register to zero, corresponding to the zero function, or copying the contents of a register, corresponding to an identity function).

The point obviously generalizes, giving us

**Theorem 18.** *Primitive recursive functions are effectively computable by a series of (possibly nested) ‘for’ loops.*

### 16.3 If you can compute it using ‘for’-loops, it is p.r.

The converse is also true. Take a ‘for’ loop which computes the value of a function  $f$  for given arguments, a loop which calls on two prior routines, one which computes a function  $g$  (used to set the value of  $f$  with some key argument set to zero), the other which computes a function  $h$  (which is used on each loop to fix the next value of  $f$  as that argument is incremented). This plainly corresponds to a definition by recursion of  $f$  in terms of  $g$  and  $h$ . And generalizing, if a function can be computed by a program using just ‘for’ loops as its main programming structure – with the program’s ‘built in’ functions all being p.r. – then the newly defined function will also be primitive recursive.

This gives us a quick-and-dirty way of convincing ourselves that a new function is p.r.: *sketch out a routine for computing it and check that it can all be done with a succession of (possibly nested) ‘for’ loops which only invoke already known p.r. functions: then the new function will be primitive recursive.*

## 17 Not all computable numerical functions are p.r.

We have seen that any p.r. function is mechanically computable. *But not all effectively computable numerical functions are primitive recursive.* In this section, we first make the claim that there are computable-but-not-p.r. numerical functions look plausible. Then we’ll cook up an example.

First, then, some plausibility considerations. We’ve just seen that the values of a given primitive recursive function can be computed by a program involving ‘for’ loops as its main programming structure. Each loop goes through a specified number of iterations. However, we do allow computations to involve *open-ended searches*, with no prior bound on the length of search. We made essential use of this permission when we showed that negation-complete theories are decidable – for we allowed the process ‘enumerate the theorems and wait to see which of  $\varphi$  or  $\neg\varphi$  turns up’ to count as a computational decision procedure.

Standard computer languages of course have programming structures which implement just this kind of unbounded search. Because as well as ‘for’ loops, they allow ‘do until’ loops (or equivalently, ‘do while’ loops). In other words, they allow some process to be iterated until a given condition is satisfied – *where no prior limit is put on the the number of iterations to be executed.*

If we count what are presented as unbounded searches as computations, then it looks very plausible that not everything computable will be primitive recursive.

True, that is as yet only a plausibility consideration. Our remarks so far leave open the possibility that computations can always somehow be turned into procedures using ‘for’ loops with a bounded limit on the number of steps. But in fact we can now show that isn’t the case:

**Theorem 19.** *There are effectively computable numerical functions which aren’t primitive recursive.*

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

This holds because, by definition, every p.r. function has a ‘recipe’ in which it 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 recipes. Then we can

	0	1	2	3	...
$f_0$	<u><math>f_0(0)</math></u>	$f_0(1)$	$f_0(2)$	$f_0(3)$	...
$f_1$	$f_1(0)$	<u><math>f_1(1)</math></u>	$f_1(2)$	$f_1(3)$	...
$f_2$	<u><math>f_2(0)</math></u>	$f_2(1)$	<u><math>f_2(2)</math></u>	$f_2(3)$	...
$f_3$	$f_3(0)$	$f_3(1)$	$f_3(2)$	<u><math>f_3(3)</math></u>	...
...	...	...	...	...	↘

effectively generate ‘in alphabetical order’ all possible strings of symbols from this language; and as we go along, we select the strings that obey the rules for being a recipe for a p.r. function (that’s a mechanical procedure). That generates a list of recipes which effectively enumerates the p.r. functions, repetitions allowed.

Now consider our table. Down the table we list off the p.r. functions  $f_0, f_1, f_2, \dots$ . An individual row then gives the values of  $f_n$  for each argument. Let’s 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 recipes 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  $\delta$  can’t be primitive recursive. For suppose otherwise. Then  $\delta$  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  $\delta$  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 ‘diagonalized out’ of the class of p.r. functions to get a new function  $\delta$  which is effectively computable but not primitive recursive. □

‘But hold on! *Why* is the diagonal function 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). 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.

## 18 Defining p.r. properties and relations

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

Now, quite generally, we can tie talk of functions and talk of properties and relations together by using the notion of a *characteristic function*. Here's a definition.

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 (less usual) selection of 0 rather than 1 for *true* is merely for later convenience in *IGT*.

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 in what follows, we'll frequently 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:

1. We can officially say that a numerical property is *effectively decidable* – i.e. a suitably programmed computer can decide whether the property obtains – just if its *characteristic function* is (*total and*) *effectively computable*. (The characteristic function needs to be total because it needs to deliver a verdict about each number as to whether it has the property in question.)

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

2. A *p.r. property* is a property with a p.r. characteristic function, and likewise a *p.r. relation* is a relation with a p.r. characteristic function.

Given that any p.r. function is effectively computable, p.r. properties and relations are among the effectively decidable ones.

Now read *IGT*, Ch. 11.