

Expressing and capturing the primitive recursive functions

Peter Smith

University of Canterbury, Christchurch, NZ

April 7, 2010

-
- L_A can express all p.r. functions
 - The role of the β -function trick in proving that result
 - In fact, Σ_1 wffs suffice to express all p.r. functions
 - Q can capture all p.r. functions
 - Expressing and capturing p.r. properties and relations.
-

The last episode wasn't about logic or *formal* theories at all: it was about common-or-garden arithmetic and the informal notion of computability.

We noted that addition can be defined in terms of repeated applications of the successor function. Multiplication can be defined in terms of repeated applications of addition. The exponential and factorial functions can be defined, in different ways, in terms of repeated applications of multiplication. There's already a pattern emerging here!

The main task in the last episode was to get clear about this pattern. So first we said more about the idea of defining one function in terms of repeated applications of another function. Tidied up, that becomes the idea of *defining a function by primitive recursion* (Defn. 28).

Then we want the idea of a definitional chain where we define a function by primitive recursion from other functions which we define by primitive recursion from other functions, and so on down, until we bottom out with the successor function and other trivia. We also of course allow composition of functions – i.e. feeding the output of one already-defined function into another already-defined function – along the way. Tidied up, this gives us the idea of a *primitive recursive function*, i.e. one that can be defined by such a definitional chain (Defn. 30).

We then noted three key facts:

1. Every p.r. function is intuitively computable – moreover it is computable without going in for open-ended searches. It can be computed using only 'for' loops and not open-ended 'do until' loops. That's Theorem 20.
2. Conversely, if a numerical function can be computed from a starter pack of trivial functions using only 'for' loops, then it is primitive recursive. That's Theorem 21.
3. But not every intuitively computable numerical function is primitive recursive. That's Theorem 22.

Let's just comment on the proof of the third result.

We noted that we can effectively list (the recipes for) p.r. functions, specifying the functions f_0, f_1, f_2, \dots . We can then define the function $d(n) = f_n(n) + 1$. This is effectively computable, because we just go along the effectively generated list of recipes till the n -th one, and use that recipe applied to input n to compute $f_n(n)$ and then add one. But this computable function is distinct from all the f_j . So it isn't primitive recursive.

The argument evidently generalizes. Suppose we can effectively list the recipes for some other class C of computable total (i.e. everywhere-defined) functions, specifying the functions $f_0^C, f_1^C, f_2^C, \dots$. Then again we can define $d^C(n) = f_n^C(n) + 1$, which will be everywhere-defined because f_j^C is everywhere defined, is computable, but not in C . In a slogan, we can 'diagonalize out' of class C . So that gives us a theorem:

Theorem 23. *No effective listing of algorithms can include algorithms for all the intuitively computable total functions.*¹

OK, so the situation is now this. We've been talking about *formal* arithmetics with just three functions – successor, addition, multiplication – built in. We've reminded ourselves that ordinary *informal* arithmetic talks about heaps more elementary functions like the exponential, the factorial, the-number-of-prime-divisors-of, and so on and so forth: and we generalized the sort of way these functions can be defined to specify the whole class of primitive recursive functions. A gulf seems to have opened up between the modesty of the resources of our formal theories (including the strongest so far, PA) and the richness of the world of p.r. functions (and we know that those aren't even all the computable arithmetical functions). In this episode, we show the gulf is merely apparent. The language L_A in fact can *express* all p.r. functions; and even the weak theory Q can *capture* them all too. So, in fact, our formal theories – despite their modest basic resources – can deal with a lot more than you might at first sight suppose.

Now recall the idea of a sufficiently strong theory which we introduced in Defn. 16. That was the idea of capturing all decidable numerical properties. That's equivalent to the idea of capturing all computable one-place functions (by the link between properties and their characteristic functions). Well, what we are claiming is that we can show at least that Q and hence PA can capture all primitive recursive computable functions. That will be enough for Gödel's argument for incompleteness to fly.

22 L_A can express all p.r. functions

We want to show that if the one-place function f is p.r., then there is a two-place L_A wff $\varphi(x, y)$, such that $\varphi(\bar{m}, \bar{n})$ is true if and only if $f(m) = n$. And similarly, of course, for many-place p.r. functions.

22.1 Proof strategy

Suppose that the following three propositions are all true:

- E1. L_A can express the initial functions, and addition and multiplication. (See Defn. 27.)
- E2. If L_A can express the functions g and h , then it can also express a function f defined by composition from g and h . (See Defn. 29.)
- E3. If L_A can express the functions g and h , then it can also express a function f defined by primitive recursion from g and h . (See Defn. 28.)

Then by the argument of §18, that establishes

Theorem 24. *L_A can express all p.r. functions.*

But it is trivial to prove E1. Just look at cases. The successor function $Sx = y$ is of course expressed by the open wff $Sx = y$. The addition function $x + y = z$ is expressed by $x + y = z$. Similarly for multiplication.

The zero function, $Z(x) = 0$ is expressed by the wff $Z(x, y) =_{\text{def}} (x = x \wedge y = 0)$.

Finally, the three-place identity function $I_2^3(x, y, z) = y$, to take just one example, is expressed by the wff $I_2^3(x, y, z, u) =_{\text{def}} (x = x \wedge y = u \wedge z = z)$. Likewise for all the other identity functions. [Check those claims!]

So that just leaves E2 and E3 to prove.

¹Note that the restriction to total functions is doing essential work here. Consider algorithms for *partial* computable functions (the idea is that when the algorithm for the partial function φ_i 'crashes' on input n , $\varphi_i(n)$ is undefined). And consider a listing of algorithms for partial functions. $\delta(n) = \varphi_n(n) + 1$ could then be e.g. φ_d , if $\varphi_d(d)$ and hence $\varphi_d(d) + 1$ are both undefined.

22.2 Proving E2

This result is pretty trivial too. Suppose g and h are one-place functions, expressed by the wffs $G(x, y)$ and $H(x, y)$ respectively. Then, the function $f(x) = h(g(x))$ is evidently expressed by the wff $\exists z(G(x, z) \wedge H(z, y))$.

For suppose $g(m) = k$ and $h(k) = n$, so $f(m) = n$. Then by hypothesis $G(\bar{m}, \bar{k})$ and $H(\bar{k}, \bar{n})$ will be true, and hence $\exists z(G(\bar{m}, z) \wedge H(z, \bar{n}))$ is true, as required. Conversely, suppose $\exists z(G(\bar{m}, z) \wedge H(z, \bar{n}))$ is true. Then since the quantifiers run over numbers, $(G(\bar{m}, \bar{k}) \wedge H(\bar{k}, \bar{n}))$ must be true for some k . So we'll have $g(m) = k$ and $h(k) = n$, and hence $f(m) = h(g(m)) = n$ as required.

Other cases where g and/or h are multi-place functions can be handled similarly.

22.3 What it takes to define the factorial

Proving E3 is the tricky case.² We'll illustrate the general strategy by first taking a particular case of a definition by primitive recursion, and then we'll generalize. So consider the primitive recursive definition of the factorial function again:

$$\begin{aligned} 0! &= 1 \\ (Sx)! &= x! \times Sx \end{aligned}$$

The multiplication and successor functions here are of course expressible in L_A : but how can we express our defined function in L_A ?

Think about the p.r. definition for the factorial in the following way. It tells us how to construct a sequence of numbers $0!, 1!, 2!, \dots, x!$, where we move from the u -th member of the sequence (counting from zero) to the next by multiplying by Su . Putting $y = x!$, the p.r. definition thus says

- A. There is a sequence of numbers k_0, k_1, \dots, k_x such that: $k_0 = 1$, and if $u < x$ then $k_{Su} = k_u \times Su$, and $k_x = y$.

So the question of how to reflect the p.r. definition of the factorial inside L_A comes to this: how can we express facts about *finite sequences of numbers* using the limited resources of L_A ?

What we need to do is to wrap up a finite sequence into a single code number c , and then have a decoding function *decode* such that if you feed *decode* the code number c and the index i it spits out the i -th member of the sequence which c codes! In other words, if c is the code number for the sequence k_0, k_1, \dots, k_x we want: $\text{decode}(c, i) = k_i$.

If we can find a coding scheme and a decoding function that does the trick, then we can rewrite the p.r. definition of the factorial as

- B. There is a code number c such that: $\text{decode}(c, 0) = 1$, and if $u < x$ then $\text{decode}(c, Se) = \text{decode}(c, u) \times Su$, and $\text{decode}(c, x) = y$.

And if *decode* can be expressed in L_A then we can define the factorial in L_A .

22.4 Coding sequences if we have a factorizing function to play with

Let's just note – before giving a decode function that can be expressed in pure L_A – that if we were working in a slightly richer language the task would be easy.

Suppose $\pi_0, \pi_1, \pi_2, \pi_3, \dots$ is the series of prime numbers $2, 3, 5, 7, \dots$. Now consider the number

$$b = \pi_0^{k_0} \cdot \pi_1^{k_1} \cdot \pi_2^{k_2} \cdot \dots \cdot \pi_n^{k_n}.$$

²Don't worry if you find the ensuing argument a bit boggling (though really you shouldn't, as the basic proof idea is not hard even if its implementation takes a bit of a trick). As far as understanding Gödel's theorems are concerned, what you really need to know is that Theorem 24 *can* be proved, and not the details about *how* it is proved.

b can be thought of as encoding the whole sequence $k_0, k_1, k_2, \dots, k_n$. And we can recover the coded sequence from b by using the (primitive recursive) decoding function *power* where $power(b, i)$ is the power of the i -th prime in the prime factorization of b . (That's unique by the fundamental theorem of arithmetic that says that prime factorizations are unique.)

So *there is nothing at all mysterious about a coding scheme for finite sequences and a decoding function that recovers elements of the sequence from its code*: the decoding function *power* would do the job. And a language L_A^+ with an expression for *power* built in would be able to define the factorial function in the way explained.

But of course, *power* isn't built into L_A or obviously definable in it. The question now is: can we construct a different coding scheme with a decoding function which can be constructed from the successor, addition and multiplication functions which *are* built into L_A ?

22.5 Gödel's β -function

It turns out to simplify things if we liberalize our notion of coding/decoding just a little. So we'll now allow *three*-place decoding-functions, which take *two* code numbers c and d , as follows:

A three-place decoding function is a function of the form $decode(c, d, i)$ such that, for *any* finite sequence of natural numbers $k_0, k_1, k_2, \dots, k_n$ there is a pair of code numbers c, d such that for every $i \leq n$, $decode(c, d, i) = k_i$.

A three-place decoding-function will do just as well as a two-place function to help us express facts about finite sequences.

Even with this liberalization, though, it still isn't obvious how to define a decoding-function in terms of the functions built into basic arithmetic. But Gödel neatly solved our problem with the following little trick. Put

$$\beta(c, d, i) =_{\text{def}} \text{the remainder left when } c \text{ is divided by } d(i + 1) + 1.$$

Then, given any sequence k_0, k_1, \dots, k_n , we can find a suitable pair of numbers c, d such that for $i \leq n$, $\beta(c, d, i) = k_i$.

This claim should look intrinsically plausible. As we divide c by $d(i + 1) + 1$ for different values of i ($0 \leq i \leq n$), we'll get a sequence of $n + 1$ remainders. Vary c and d , and the sequence of $n + 1$ remainders will vary. The permutations as we vary c and d without limit *appear* to be simply endless. We just need to check, then, that appearances don't deceive, and we *can* always find a (big enough) c and a (smaller) d which makes the sequence of remainders match a given $n + 1$ -term sequence of numbers (mathmos: see *IGT*, 13.4, fn. 6 for proof that this works!)

But now reflect that the concept of a remainder on division can be elementarily defined in terms of multiplication and addition. Thus consider the following open wff:

$$B(c, d, i, y) =_{\text{def}} (\exists u \leq c)[c = \{S(d \times Si) \times u\} + y \wedge y \leq (d \times Si)].$$

This, as we want, expresses our Gödelian β -function in L_A (for remember, we can define ' \leq ' in L_A).

22.6 Defining the factorial in L_A

We've just claimed: given any sequence of numbers k_0, k_1, \dots, k_x , there are code numbers c, d such that for $i \leq x$, $\beta(c, d, i) = k_i$. So we can reformulate

- A. There is a sequence of numbers k_0, k_1, \dots, k_x such that: $k_0 = 1$, and if $u < x$ then $k_{Su} = k_u \times Su$, and $k_x = y$,

as follows:

- C. There is some pair c, d such that: $\beta(c, d, 0) = 1$, and if $u < x$ then $\beta(c, d, Su) = \beta(c, d, u) \times Su$, and $\beta(c, d, x) = y$.

But we've seen that the β -function can be expressed in L_A by the open wff we abbreviated B. So we can translate (C) into L_A as follows:

$$D. \exists c \exists d \{ \mathbf{B}(c, d, 0, \bar{1}) \wedge (\forall u \leq x) [u \neq x \rightarrow \exists v \exists w \{ (\mathbf{B}(c, d, u, v) \wedge \mathbf{B}(c, d, Su, w)) \wedge w = v \times Su \}] \wedge \mathbf{B}(c, d, x, y) \}.$$

Abbreviate all that by ‘ $\mathbf{F}(x, y)$ ’, and we’ve arrived! For this evidently expresses the factorial function.

22.7 Generalizing to prove E3

Finally, we need to show that we can use the same β -function trick and prove more generally that, if the function f is defined by recursion from functions g and h which are already expressible in L_A , then f is also expressible in L_A .

So here, just for the record, is the entirely routine generalization we need (there are no new ideas here – just unavoidable clutter).

We are assuming that

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

This definition amounts to fixing the value of $f(\vec{x}, y) = z$ thus:

$$A^* \text{ There is a sequence of numbers } k_0, k_1, \dots, k_y \text{ such that: } k_0 = g(\vec{x}), \text{ and if } u < y \text{ then } k_{u+1} = h(\vec{x}, u, k_u), \text{ and } k_y = z.$$

So using a three-place β -function again, that comes to

$$C^* \text{ There is some } c, d, \text{ such that: } \beta(c, d, 0) = g(\vec{x}), \text{ and if } u < y \text{ then } \beta(c, d, Su) = h(\vec{x}, u, \beta(c, d, u)), \text{ and } \beta(c, d, y) = z.$$

Suppose we can already express the n -place function g by a $(n+1)$ -variable expression \mathbf{G} , and the $(n+2)$ -variable function h by the $(n+3)$ -variable expression \mathbf{H} . Then – using ‘ \vec{x} ’ to indicate a suitable sequence of n variables – (B^*) can be rendered into L_A by

$$D^* \exists c \exists d \{ \exists k [\mathbf{B}(c, d, 0, k) \wedge \mathbf{G}(\vec{x}, k)] \wedge (\forall u \leq y) [u \neq y \rightarrow \exists v \exists w \{ (\mathbf{B}(c, d, u, v) \wedge \mathbf{B}(c, d, Su, w)) \wedge \mathbf{H}(\vec{x}, u, v, w) \}] \wedge \mathbf{B}(c, d, y, z) \}.$$

Abbreviate this defined wff $\varphi(\vec{x}, y, z)$; it is then evident that φ will serve to express the p.r. defined function f . Which gives us the desired result E3.

So, we’ve shown how to establish each of the claims E1, E2 and E3 from the start of §22.1. Hence every p.r. function can be expressed in L_A .

Theorem 24 is in the bag!

23 Primitive recursive functions can be canonically expressed by Σ_1 wffs

In this section, we extract more information out of the proof of Theorem 24. In particular we show that the L_A wff needed to express a p.r. function is logically not very complex – a Σ_1 wff (in the sense of Defn. 25) is enough to do the job.

23.1 The proof of Theorem 24 is constructive

What we showed, in effect, is how to take a chain of definitions by composition and primitive recursion – starting with the initial functions and building up to a full definition for f – and then step-by-step reflect it in building up to a wff that expresses f .

Remember: a full definition for f is in effect a recipe for defining a whole sequence of functions $f_0, f_1, f_2, \dots, f_k$ where each f_j is either an initial function or is constructed out of previous functions by composition or recursion, and $f_k = f$. Corresponding to that sequence of functions

we can write down a sequence of L_A wffs which express those functions. In the terms of §22.1, we write down the E1 expression corresponding to an initial function. If f_j comes from two previous functions by composition, we use the existential construction in E2 to write down a wff built out of the wffs expressing the two previous functions. If f_j comes from two previous functions by recursion, we use the β -function trick and write down a D^* -style expression built out of the wffs expressing the two previous functions.

So that means we've not only proved that for any given p.r. function f *there exists* an L_A -wff which expresses it. We've shown how to *construct* such a wff by recapitulating the structure of a definitional 'history' for f . The proof is, in a good sense, a constructive one.

For brevity, let's now say that

Defn. 34. *An L_A wff canonically expresses the p.r. function f if it recapitulates a full definition for f by being constructed in the manner described in the proof of Theorem 24.*

23.2 Canonical wffs for expressing p.r. functions are Σ_1

The canonical wff which reflects a full definition of f is built up starting from wffs expressing initial wffs (and addition and multiplication). Those starter wffs are Δ_0 wffs, and hence Σ_1 .

Suppose g and h are one-place functions, expressed by the Σ_1 wffs $G(x, y)$ and $H(x, y)$ respectively. Then, the function $f(x) = h(g(x))$ is expressed by the wff $\exists z(G(x, z) \wedge H(z, y))$ which is Σ_1 too. For that is equivalent to a wff with the existential quantifiers pulled from the front of the Σ_1 wffs G and H out to the very front of the new wff. Similarly for other cases of composition.

Finally, if we can already express the one-place function g by a two-variable Σ_1 expression G , and the two-place function h by the *three*-variable Σ_1 expression H . Then if f is defined from g and h by primitive recursion, f can be expressed by

$$D^* \exists c \exists d \{ \exists k [B(c, d, 0, k) \wedge G(x, k)] \wedge \\ (\forall u \leq y) [u \neq y \rightarrow \exists v \exists w \{ (B(c, d, u, v) \wedge B(c, d, Su, w)) \wedge H(x, u, v, w) \}] \wedge \\ B(c, d, y, z) \}.$$

And this too is Σ_1 . For B is Σ_1 : and D^* is equivalent to what we get when we drag all the existential quantifiers buried at the front of each of B , G and H to the very front of the wff. (Yes, dragging existentials past a universal is usually wicked! – but here the only universal here is a bounded universal, which is 'really' just a tame conjunction, and simple tricks explained in *IGT* allow us to get the existentials all at the front). Again this generalizes to other cases of definition by recursion.

So in fact our recipe for building a canonical wff in fact gives us a Σ_1 wff. Which yields

Theorem 25. *L_A can express any p.r. function f by a Σ_1 canonical wff which recapitulates a full definition for f .*

24 Q can capture all p.r. functions

We now want to show that not only can the language of Q express all p.r. functions, but also:

Theorem 26. *The theory Q can capture any p.r. function by a Σ_1 wff.*

Recall, 'capturing' a function here means being able to case-by-case prove formulae that in effect assign the right values to the function (see Defn. 15). So the formula $\chi(x, y)$ captures the one-place predicate in Q if, when $f(m) = n$, $Q \vdash \chi(\bar{m}, \bar{n})$, and when $f(m) \neq n$, $Q \vdash \neg\chi(\bar{m}, \bar{n})$. Similarly for many-place functions.

Now there's more than one route to Theorem 26. I'll mention *the direct assault* and *the clever trick*. In *IGT*, Ch. 13, I go for the clever trick. Which I now rather regret – for while clever tricks can give you a theorem they may not necessarily give you real understanding. So what I'll describe here is the direct assault. And this is just to once more replicate the overall strategy for proving results about all p.r. functions which we described in §18, and deployed already in §19.1 and §22.1.

Suppose then that we can prove

- C1. \mathcal{Q} can capture the initial functions.
- C2. If \mathcal{Q} can capture the functions g and h , then it can also capture a function f defined by composition from g and h .
- C3. If \mathcal{Q} can capture the functions g and h , then it can also capture a function f defined by primitive recursion from g and h .

where in each case the capturing wffs are Σ_1 . Then – by just the same sort of argument as in §22.1 – it follows that \mathcal{Q} can capture any p.r. function by a Σ_1 wff.

So how do we prove C1? We just check that the formulae we said in §22.1 *express* the initial functions in fact serve to *capture* the initial functions in \mathcal{Q} .

How do we prove C2? Again we track the proof in §22.2. Suppose g and h are one-place functions, captured by the wffs $G(x, y)$ and $H(x, y)$ respectively. Then we prove that the function $f(x) = h(g(x))$ is captured by the wff $\exists z(G(x, z) \wedge H(z, y))$ (which is Σ_1 if G and H are).

And how do we prove C3? This is the tedious case that takes hard work! We need to show the formula B not only expresses but captures Gödel's β -function. And then we use that fact to prove that if the n -place function g is captured by a $(n + 1)$ -variable expression G , and the $(n + 2)$ -variable function h by the $(n + 3)$ -variable expression H , then the rather horrid wff D^* in §22.7 captures the function f defined by primitive recursion from g and h . (If you want the gory details establishing that, then you can consult e.g. Elliott Mendelson, *Introduction to Mathematical Logic*, 4th edn, Prop. 3.24. And you can check that the result again is Σ_1 if G and H are.)

So the basic story is this. Take a full definition for defining a p.r. function, ultimately out of the initial functions. Follow the step-by-step instructions implicit in §22 about how to build up a canonical wff which in effect recapitulates that recipe. You'll get a wff which expresses the function, and that same wff captures the function in \mathcal{Q} (and in any stronger theory with a language which includes L_A). Moreover the wff in question will be Σ_1 .

25 Expressing/capturing properties and relations

Just a brief coda, linking what we've done in this episode with the last section of the previous one.

We said in §21, Defn. 32 that the characteristic function c_P of a monadic numerical property P is defined by setting $c_P(m) = 0$ if m is P and $c_P(m) = 1$ otherwise. And a property P is said to be p.r. decidable if its characteristic function is p.r.

Now, suppose that P is p.r.; then c_P is a p.r. function. So, by Theorem 24, L_A can express c_P by a two-place open wff $c_P(x, y)$. So if m is P , then $c_P(m) = 0$, then $c_P(\bar{m}, 0)$ is true. And if m is not P , then $c_P(m) \neq 0$, then $c_P(\bar{m}, 0)$ is not true. So, by the definition of expressing-a-property, the wff $c_P(x, 0)$ serves to express the p.r. property P . The point trivially generalizes from monadic properties to many-place relations. So we have as an easy corollary of Theorem 24 that

Theorem 27. *L_A can express all p.r. decidable properties and relations.*

Similarly, suppose again that P is p.r. so c_P is a p.r. function. So, by Theorem 26, \mathcal{Q} can capture c_P by a two-place open wff $c_P(x, y)$. So if m is P , then $c_P(m) = 0$, so $\mathcal{Q} \vdash c_P(\bar{m}, 0)$. And if m is not P , then $c_P(m) \neq 0$, then $\mathcal{Q} \vdash \neg c_P(\bar{m}, 0)$. So, by the definition of capturing-a-property, the wff $c_P(x, 0)$ serves to capture the p.r. property P in \mathcal{Q} . The point trivially generalizes from monadic properties to many-place relations. So we have as an easy corollary of Theorem 26 that

Theorem 28. *\mathcal{Q} can capture all p.r. decidable properties and relations.*

Now read *IGT*, Chap 13.