Recursive Function Theory

Peter Suber, Philosophy Department, Earlham College

Recursive function theory, like the theory of Turing machines, is one way to make formal and precise the intuitive, informal, and imprecise notion of an effective method. It happens to identify the very same class of functions as those that are Turing computable. This fact is informal or inductive support for Church's Thesis, asserting that every function that is effectively computable in the intuitive sense is computable in these formal ways.

Like the theory of Turing machines, recursive function theory is vitally concerned with the converse of Church's Thesis: to prove that all the functions they identify as computable are effectively computable, that is, to make clear that they use only effective methods.

Recursive function theory begins with some very elementary functions that are intuitively effective. Then it provides a few methods for building more complicated functions from simpler functions. The building operations preserve computability in a way that is both demonstrable and (one hopes) intuitive. The initial functions and the building operations are very few in number, and very simple in character, so that they are entirely open to inspection. It is easy to trust the foundation, and by its nature if you trust the foundation you trust the superstructure it permits you to build.

Because recursive function theory was developed in part to capture the intuitive sense of effectiveness in a rigorous, formal theory, it is important to the theory that the class of recursive functions can be built up from intuitively effective simple functions by intuitively effective techniques. Ironically, in my view, no author that I have seen expounds recursive function theory for beginners so that it really looks as intuitive as it intends to be. My intention here is to make the theory intuitively clear without sacrificing rigor.

The Initial Functions

Logicians differ in their choice of the set of elementary functions that comprise the ultimate or primitive "ingredients" of all other functions. Picking them is comparable to picking axioms for a formal system that is to capture a predetermined body of mathematics. There is some slack, but not complete freedom.

If we continue the comparison to axiomatics, then we must revert to an ambition of axiomatics almost completely abandoned in the 20th century: the ambition to make the axioms "self-evident truths". The elementary functions

must be intuitively effective or computable. If they are not, one will still get an interesting, perhaps an identical, set of resulting functions; but one will have abandoned part of the programmatic motive of the theory. (It would be much like developing a "modified" Turing machine that did the same work as an ordinary Turing machine but whose method of computation was so complex that it was mysterious.)

Our elementary functions are total, not partial. The reason should be clear. Only total functions fit the intuitive sense of effective method. A partial function is not defined for some arguments, and hence computes no result for those values. It is part of the definition of an effective method that it will always produce the correct answer.

Our elementary functions are all functions of the natural numbers. Hence, they may take zero as input, but not a negative number, and not any rational or irrational fraction.

On most lists of "ingredient" functions are the zero function, the successor function, and the identity function.

$z(x) = 0$
$s(x)$ = successor of x (roughly, "x + 1")
$id(x) = x$

The zero function returns zero regardless of its argument. It is hard to imagine a more intuitively computable function.

The successor function returns the successor of its argument. Since successorship is a more primitive notion than addition, the "x + 1" that I've used to elucidate the function is simply for the convenience of readers. The addition function does not yet exist and must be built up later.

If successorship is obscure or non-intuitive in the absence of addition, remember that we have allowed ourselves to use the system of natural numbers. We can't calculate yet, but we can count, and that is all we need here. (Hence, the core of recursive function theory that must be intuitive includes not only simple functions and their building operations, but also the natural numbers.)

The zero and successor functions take only one argument each. But the identity function is designed to take any number of arguments. When it takes one argument (as above) it returns its argument as its value. Again, its effectiveness is obvious. When it takes more than one argument, it returns one of them.

$id_{2,1}(x,y) = x$
$id_{2,2}(x,y) = y$

The two subscripts to the identity function indicate, first, the number of arguments it takes, and second, which argument will be returned by the function. (We could write the identity function more generally so that it applied to any number of arguments, but to preserve simplicity we will not.)

While others could be added, these three elementary functions *suffice* (with the techniques to be introduced) to build up all the functions we believe to be computable in the intuitive sense. Given the simplicity of these "ingredient" functions, this is a remarkable simplification of computation.

The Building Operations

We will build more complex and interesting functions from the initial set by using only three methods. If we use the analogy to axiomatics again, the methods for building higher functions from our initial set correspond to rules of inference applied to axioms.

The three methods are composition, primitive recursion, and minimization.

Composition

If we start with the successor function,

s(x)

then we may replace its argument, x, with a function. If we replace the argument, x, with the zero function,

z(x)

then the result is the successor of zero.

s(z(x)) = 1
s(s(z(x))) = 2 and so on.

In this way, *compounding* some of the initial functions can describe the natural numbers.

This building operation is called "composition" (sometimes "substitution"). It rests on the simple fact that computable functions have values, which are numbers. Hence, where we would write a number (numeral) we can instead write any function with its argument(s) that computes that number. In particular, we may use a function with its argument(s) in the place of an argument.

If we think of a function and its arguments as an alternate *name* of its value, then composition simply allows us to use this kind of name as the argument to a second function.

In the language used in predicate logic, a function with its arguments is a *term*, and may be used wherever terms are used in predicate logic wffs. The arguments of functions are also terms. Hence, a function can have other functions as arguments.

We can define the function that adds 2 to a given number thus: add2(x) = s(s(x)). Introducing new names to abbreviate the compound functions created by composition is a great convenience since complex functions are made more intuitive by chunking their simpler components. But the essence of

composition is not the re-naming or abbreviating of compound functions, but the compounding itself.

Obviously, the computability or effectiveness of composition is not affected by the number of arguments in any of the component functions, provided it is finite.

If a function is a rule (for picking a member of the range when given a member of the domain), then it must be computed to reach its result. In that sense we may consider functions to be programs or software, and their arguments input or data. From this perspective, the act of composition replaces data with a program that, when run, gives as output the data one desired as input. Composition shows the interchangeability, under controlled circumstances, of software and data.

It should be clear that when composition is applied to computable functions, only computable functions will result. Even if this is obvious, it may be well to aid intuition with reason. Composition yields computable functions when applied to computable functions because it does nothing more than ask us to compute more than one function and to do so in a certain order (working from the inside of the compound expression to the outside). If the ingredient functions are computable, then the task of computing any finite number of them in sequence will be computable. If the number of nested functions is finite (if the length of wffs is finite), then the effectiveness of the overall computation is not diminished.

Primitive Recursion

The second building operation is called primitive recursion. To those not used to it, it can be far from intuitive; to those who have practiced with it, it is fundamental and elegant.

Function h is defined through functions f and g by primitive recursion when

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

Let's unpack this slowly. First, remember that f and g are known computable functions. Primitive recursion is a method of defining a new function, h, through old functions, f and g. Second, notice that there are two equations. When h's second argument is zero, the first equation applies; when it is not zero, we use the second. Notice how the successor function enforces the condition that the argument be greater than zero in the second equation. Hence, the first equation applies in the "minimal case" and the second applies in every other case. (It is not unusual for a function to be defined by a series of equations, rather than just one, in order to show its value for relevantly different inputs or arguments.)

So when (1) the second argument of h is zero, the function is equivalent to some known function f and we compute it; otherwise (2) it is equivalent to some known function g and we compute it. That's how we know that function h, or more generally, the functions built by primitive recursion, will be

computable.

It is the second case that is interesting. We already know how to compute function g, but look at its two arguments in this case. Instead of taking two numbers, function g is here a compound function and takes one number x and another function. Its second argument is expressed by function h again! This circle is the essence of primitive recursion. To calculate function h when neither argument is zero we must calculate function g, which requires us to calculate function h.

This would be impossible if the function took the *same arguments* in both instances. It would be like saying to someone who could not multiply 10 by 20, "Nothing could be simpler; one only has to multiply 10 by 20!" Fortunately, that is not the case here. Compare the arguments for function h that we see on the left and right sides of the second equation. To calculate h when its arguments are (x,s(y)) we must calculate h when its arguments are (x,y). So the function is calling itself with *different arguments*. Note the second argument in particular. It has decremented from the sucessor of y (= y+1) to y itself. This means that to calculate h, we must decrement one of its arguments by one and calculate that. So the function does not futilely call itself; it calls a variation of itself, which makes all the difference.

(We could write the equations for primitive recursion more generally to show how a function of any number of terms calls a variation of itself, but again for the sake of simplicity we will not.)

But how do we do calculate the decremented variation of the original? Look at the equations. To calculate h when its second argument is n, we calculate h when its second argument is n-1; and to do that we calculate h when its second argument is n-2; and so on; that is the procedure required by the two equations. But eventually by this means the second argument will equal zero, in which case we use the first equation, and calculate function f, and are done.

Let's look at an example. To calculate 5! (5 "factorial"), we multiply 5x4x3x2x1. How would we define the general factorial function recursively? To say that it is n(n-1)(n-2)...(n-(n-1)) would be correct but not recursive. In that formulation, the function never calls itself and it contains the mysterious ellipsis (three dots). Recursive functions not only have the peculiarity of calling themselves, but they eliminate the need to use the ellipsis. This is a considerable improvement in clarity, rigor, completeness of specification, and intuitive effectiveness, even if it is still "weird" from another standpoint.

So, the recursive definition of the factorial function would be constructed like this. Initially, we note that 1! = 1. Now, to compute n!, we do not multiply n by (n-1), for that would commit us to the non-recursive series we saw above. Instead we multiply n by (n-1)!. And that's it; there is no further series. But to calculate (n-1)! we refer to our equations; if we have not reached 1 yet, then we multiply n-1 by (n-2)!, and so on until we do reach 1, whose factorial value is already defined. So the general formula would be expressed by these two equations:

1! = 1

n! = n(n-1)!

If we had already bothered to build up the functions for multiplication and subtraction from our initial functions and the building operations, then we could show more particularly how primitive recursion works as a building operation. The factorial function is derived by primitive recursion from the functions for multiplication and subtraction. A stricter definition of the factorial function, f(n), then, consists of these two equations:

$$f(n) \quad \begin{array}{ll} f(n) = 1 & \text{when } n = 1 \\ f(n) = n(f(n-1)) & \text{when } n > 1 \end{array}$$

The first equation defines the condition under which the self-calling circular process "bottoms out". Without it we would have an infinite loop which never returns an answer.

This kind of self-calling would clearly work just as well if the "bottom" condition were greater than n, and we incremented instead of decremented until we reached it.

Primitive recursion is like mathematical induction. The first equation defines the basis, and the second defines the induction step.

Note that not all "recursive functions" use the building operation called "recursion". (Worse, not all "primitive recursive functions" use "primitive recursion".) They are called "recursive functions" because they are recursively defined, just as a formal system typically defines its wffs recursively, enabling it to deal with an infinity of instances through an effective method for deciding whether arbitrary candidates are cases. The use of the terms "recursive" and "primitive recursive" for components for the overall theory, and again for the the overall theory, is confusing and regrettable but that's the way the terminology has evolved.

Minimization

Let us take a function f(x). Suppose there is at least one value of x which makes f(x) = 0. Now suppose that we wanted to find the *least* value of x which made f(x) = 0. There is an obviously effective method for doing so. We know that x is a natural number. So we set x = 0, and then compute f(x); if we get 0 we stop, having found the least x which makes f(x) = 0; but if not, we try the next natural number 1. We try 0,1,2,3... until we reach the first value which makes f(x) = 0. When we know that *there is* such a value, then we know that this method will terminate in a finite time with the correct answer. If we say that g(x) is a function that computes the least x such that f(x) = 0, then we know that g is computable. We will say that g is produced from f by minimization.

The only catch is that we don't always know that there is any x which makes f(x) = 0. Hence the project of testing 0,1,2,3... may never terminate. If we run the test anyway, that is called *unbounded minimization*. Obviously, when g is produced by unbounded minimization, it is not effectively computable.

If we don't know whether there is any x which makes f(x) = 0, then there is still a way to make function g effectively computable. We can start testing 0,1,2,3... but set an upper bound to our search. We search until we hit the upper bound and then stop. If we find a value before stopping which makes f(x) = 0, then g returns that value; if not, then g returns 0. This is called *bounded minimization*.

(We could define functions f and g more generally so that each took any number of arguments, but again for simplicity we will not.)

Of course, as a building operation we must build only with computable functions. So we build g by minimization only if f(x) is already known to be computable.

While unbounded minimization has the disadvantages of a partial function which may never terminate, bounded minimization has the disadvantage of sometimes failing to minimize.

If you know any programming language, we can make a helpful comparison. Recall the distinction between loops that iterate a definite number of times (e.g. Pascal FOR loops) and those that iterate indefinitely until some special condition is met (e.g. Pascal REPEAT and WHILE loops). Let loops of these two kinds be given the task of testing natural numbers in search of the least value of x in the function f(x) = 0. For each given candidate number, n, starting with 0 and moving upwards, the loop checks to see whether f(n) = 0. The loops which iterate a definite number of times will test only a definite, finite number of candidates. To run such a loop is to take so many steps toward the computation of function g(x). This is bounded minimization at work. Indefinitely iterating loops represent unbounded minimization: they will keep checking numbers until they find one which makes the value of function f equal zero, or until the computer dies of old age or we turn it off.

In general we will have no way of knowing in advance whether either kind of loop will ever discover what it is looking for. But we will always know in advance that a bounded search will come to an end anyway, while an unbounded search may or may not. That is the decisive difference for effective computability.

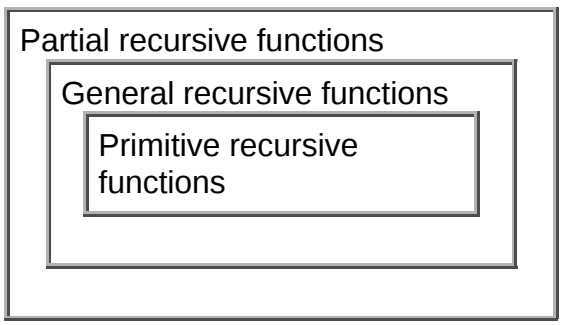
Suppose an unbounded search does terminate with a good value for y. Since the program did halt, it must have halted after a definite, finite number of steps. That means that we could have done the same work with a bounded search, if only we had known how large to make the upper bound. (In fact, we could have done the whole search without loops at all, if only we had known how many times to repeat the incrementing statements.) This fact is more important for theory than for practice. For theory it means that whatever unbounded minimization can actually compute can be computed by bounded minimization; and since the latter is effective, the ineffectiveness of the former is significantly mitigated. (We can increase the bound in bounded minimization to any finite number, in effect covering the ground effectively that unbounded minimization covers ineffectively.) For practical computation this means little, however. Since we rarely know where to set the upper bound, we will use unbounded searches even though that risks non-termination (infinite loops).

Our job is not made any easier by knowing that if our unbounded search should ever terminate, then a bounded search could have done the same work without the risk of non-termination.

Summary

Some terms will help us ring a sub-total. If we allow ourselves to use composition, primitive recursion, and bounded minimization as our building operations, then the functions we can build are called *primitive recursive* (even if we did not use the building operation of primitive recursion). If we add unbounded minimization to our toolkit, we can build a larger class of functions that are called *general recursive* or just plain *recursive*. More precisely, functions are general recursive only when they use unbounded minimization that happens to terminate. Functions using unbounded minimization that does not terminate are called *partial recursive* because they are partial functions.

Church's Thesis refers to general recursive functions, not to primitive recursive functions. There are effectively computable functions that are not primitive recursive, but that are general recursive (e.g. Ackermann's function). But the open challenge latent in Church's Thesis has elicited no case of an effectively computable function that is not general recursive (if general recursive subsumes and includes the primitive recursive).



The set of general recursive functions is demonstrably the same as the set of Turing computable functions. Church's Thesis asserts indemonstrably that it is the same as the set of intuitively computable functions. In Douglas Hofstadter's terms, the partial recursive functions are the BlooP computable functions, the general recursive functions are the terminating FlooP computable functions. In Pascal terms, the primitive recursive functions are the FOR loop computable functions, while the general recursive functions the terminating WHILE and REPEAT loop computable functions. The partial recursive functions that are not also general or primitive recursive are the WHILE and REPEAT (FlooP) functions that loop forever without terminating.

This file is an electronic hand-out for the course, Logical Systems.

Peter Suber, Department of Philosophy, Earlham College, Richmond, Indiana, 47374, U.S.A.
peters@earlham.edu.