Chapter udf

Recursive Functions

These are Jeremy Avigad's notes on recursive functions, revised and expanded by Richard Zach. This chapter does contain some exercises, and can be included independently to provide the basis for a discussion of arithmetization of syntax.

rec.1 Introduction

cmp:rec:int:sec

In order to develop a mathematical theory of computability, one has to first of all develop a *model* of computability. We now think of computability as the kind of thing that computers do, and computers work with symbols. But at the beginning of the development of theories of computability, the paradigmatic example of computation was *numerical* computation. Mathematicians were always interested in number-theoretic functions, i.e., functions $f\colon \mathbb{N}^n \to \mathbb{N}$ that can be computed. So it is not surprising that at the beginning of the theory of computability, it was such functions that were studied. The most familiar examples of computable numerical functions, such as addition, multiplication, exponentiation (of natural numbers) share an interesting feature: they can be defined *recursively*. It is thus quite natural to attempt a general definition of *computable function* on the basis of recursive definitions. Among the many possible ways to define number-theoretic functions recursively, one particulalry simple pattern of definition here becomes central: so-called *primitive recursion*.

In addition to computable functions, we might be interested in computable sets and relations. A set is computable if we can compute the answer to whether or not a given number is an element of the set, and a relation is computable iff we can compute whether or not a tuple $\langle n_1, \ldots, n_k \rangle$ is an element of the relation. By considering the *characteristic function* of a set or relation, discussion of computable sets and relations can be subsumed under that of computable functions. Thus we can define primitive recursive relations as well, e.g., the relation "n evenly divides m" is a primitive recursive relation.

Primitive recursive functions—those that can be defined using just primitive recursion—are not, however, the only computable number-theoretic functions. Many generalizations of primitive recursion have been considered, but the most powerful and widely-accepted additional way of computing functions is by unbounded search. This leads to the definition of *partial recursive functions*, and a related definition to *general recursive functions*. General recursive functions are computable and total, and the definition characterizes exactly the partial recursive functions that happen to be total. Recursive functions can simulate every other model of computation (Turing machines, lambda calculus, etc.) and so represent one of the many accepted models of computation.

rec.2 Primitive Recursion

cmp:rec:pre:sec

A characteristic of the natural numbers is that every natural number can be reached from 0 by applying the successor operation "+1" finitely many times—any natural number is either 0 or the successor of . . . the successor of 0. One way to specify a function $f\colon \mathbb{N} \to \mathbb{N}$ that makes use of this fact is this: (a) specify what the value of f is for argument 0, and (b) also specify how to, given the value of $f(x)$, compute the value of $f(x+1)$. For (a) tells us directly what $f(0)$ is, so f is defined for 0. Now, using the instruction given by (b) for $x = 0$, we can compute $f(1) = f(0+1)$ from $f(0)$. Using the same instructions for $x = 1$, we compute $f(2) = f(1+1)$ from $f(1)$, and so on. For every natural number x, we'll eventually reach the step where we define $f(x)$ from $f(x+1)$, and so $f(x)$ is defined for all $x \in \mathbb{N}$.

For instance, suppose we specify $h\colon \mathbb{N} \to \mathbb{N}$ by the following two equations:

$$\begin{aligned} h(0) &= 1 \\ h(x+1) &= 2 \cdot h(x). \end{aligned}$$

If we already know how to multiply, then these equations give us the information required for (a) and (b) above. Successively the second equation, we get that

$$\begin{aligned} h(1) &= 2 \cdot h(0) = 2, \\ h(2) &= 2 \cdot h(1) = 2 \cdot 2, \\ h(3) &= 2 \cdot h(2) = 2 \cdot 2 \cdot 2, \\ &\vdots \end{aligned}$$

We see that the function h we have specified is $h(x) = 2^x$.

The characteristic feature of the natural numbers guarantees that there is only one function d that meets these two criteria. A pair of equations like these is called a *definition by primitive recursion* of the function d. It is so-called because we define f "recursively," i.e., the definition, secifically the second equation, involves f itself on the right-hand-side. It is "primitive" because in

defining $f(x+1)$ we only use the value $f(x)$, i.e., the immediately preceding value. This is the simplest way of defining a function on $\mathbb{N}$ recursively.

We can define even more fundamental functions like addition and multiplication by primitive recursion. In these cases, however, the functions in question are 2-place. We fix one of the argument places, and use the other for the recursion. E.g, to define $\mathrm{add}(x, y)$ we can fix x and define the value first for $y = 0$ and then for $y+1$ in terms of y. Since x is fixed, it will appear on the left and on the right side of the defining equations.

$$\begin{aligned} \mathrm{add}(x, 0) &= x \\ \mathrm{add}(x, y+1) &= \mathrm{add}(x, y) + 1 \end{aligned}$$

These equations specify the value of add for all x *and* y. To find $\mathrm{add}(2, 3)$, for instance, we apply the defining equations for $x = 2$, using the first to find $\mathrm{add}(2, 0) = 2$, then using the second to successively find $\mathrm{add}(2, 1) = 2 + 1 = 3$, $\mathrm{add}(2, 2) = 3 + 1 = 4$, $\mathrm{add}(2, 3) = 4 + 1 = 5$.

In the definition of add we used + on the right-hand-side of the second equation, but only to add 1. In other words, we used the successor function $\mathrm{succ}(z) = z+1$ and applied it to the previous value $\mathrm{add}(x, y)$ to define $\mathrm{add}(x, y+1)$. So we can think of the recursive definition as given in terms of a single function which we apply to the previous value. However, it doesn't hurt—and sometimes is necessary—to allow the function to depend not just on the previous value but also on x and y. Consider:

$$\begin{aligned} \mathrm{mult}(x, 0) &= 0 \\ \mathrm{mult}(x, y+1) &= \mathrm{add}(\mathrm{mult}(x, y), x). \end{aligned}$$

This is a primitive recursive definition of a function mult by applying the function add to both the preceding value $\mathrm{mult}(x, y)$ and the first argument x. It also defines the function $\mathrm{mult}(x, y)$ for all arguments x and y. For instance, $\mathrm{mult}(2, 3)$ is determined by successively computing $\mathrm{mult}(2, 0)$, $\mathrm{mult}(2, 1)$, $\mathrm{mult}(2, 2)$, and $\mathrm{mult}(2, 3)$:

$$\begin{aligned} &\mathrm{mult}(2, 0) = 0 \\ &\mathrm{mult}(2, 1) = \mathrm{mult}(2, 0+1) = \mathrm{add}(\mathrm{mult}(2, 0), 2) = \mathrm{add}(0, 2) = 2 \\ &\mathrm{mult}(2, 2) = \mathrm{mult}(2, 1+1) = \mathrm{add}(\mathrm{mult}(2, 1), 2) = \mathrm{add}(2, 2) = 4 \\ &\mathrm{mult}(2, 3) = \mathrm{mult}(2, 2+1) = \mathrm{add}(\mathrm{mult}(2, 2), 2) = \mathrm{add}(4, 2) = 6. \end{aligned}$$

The general pattern then is this: to give a primitive recursive definition of a function $h(x_0, \ldots, x_k, y)$, we provide two equations. The first defines the value of $h(x_0, \ldots, x_k, 0)$ without reference to f. The second defines the value of $h(x_0, \ldots, x_k, y+1)$ in terms of $h(x_0, \ldots, x_k, y)$, the other arguments $x_0, \ldots, x_k$, and y. Only the immediately preceding value of h may be used in that second equation. If we think of the operations given by the right-hand-sides of these two equations as themselves being functions f and g, then the pattern to define

a new function h by primitive recursion is this:

$$h(x_0, \ldots, x_k, 0) = f(x_0, \ldots, x_k)$$
$$h(x_0, \ldots, x_k, y+1) = g(x_0, \ldots, x_k, y, h(x_0, \ldots, x_k, y)).$$

In the case of add, we have $k = 0$ and $f(x_0) = x_0$ (the identity function), and $g(x_0, y, z) = z + 1$ (the 3-place function that returns the successor of its third argument):

$$\text{add}(x_0, 0) = f(x_0) = x_0$$
$$\text{add}(x_0, y+1) = g(x_0, y, \text{add}(x_0, y)) = \text{succ}(\text{add}(x_0, y))$$

In the case of mult, we have $f(x_0) = 0$ (the constant function always returning 0) and $g(x_0, y, z) = \text{add}(z, x_0)$ (the 3-place function that returns the sum of its first and last argument):

$$\text{mult}(x_0, 0) = f(x_0) = 0$$
$$\text{mult}(x_0, y+1) = g(x_0, y, \text{mult}(x_0, y)) = \text{add}(\text{mult}(x_0, y), x_0).$$

rec.3 Composition

cmp:rec:com:sec

If f and g are two one-place functions of natural numbers, we can compose them: $h(x) = g(f(x))$. The new function $h(x)$ is then defined by *composition* from the functions f and g. We'd like to generalize this to functions of more than one argument.

Here's one way of doing this: suppose f is a k-place function, and g_0, …, g_{k-1} are k functions which are all n-place. Then we can define a new n-place function h as follows:

$$h(x_0, \ldots, x_{n-1}) = f(g_0(x_0, \ldots, x_{n-1}), \ldots, g_{k-1}(x_0, \ldots, x_{n-1})).$$

If f and all g_i are computable, so is h: To compute $h(x_0, \ldots, x_{n-1})$, first compute the values $y_i = g_i(x_0, \ldots, x_{n-1})$ for each $i = 0, \ldots, k-1$. Then feed these values into f to compute $h(x_0, \ldots, x_{k-1}) = f(y_0, \ldots, y_{k-1})$.

This may seem like an overly restrictive characterization of what happens when we compute a new function using some existing ones. For one thing, sometimes we do not use all the arguments of a function, as when we defined $g(x, y, z) = \text{succ}(z)$ for use in the primitive recursive definition of add. Suppose we are allowed use of the following functions:

$$P_i^n(x_0, \ldots, x_{n-1}) = x_i.$$

The functions P_i^k are called *projection* functions: P_i^n is an n-place function. Then g can be defined as

$$g(x, y, z) = \text{succ}(P_2^3)$$

Here the role of f is played by the 1-place function succ, so $k = 1$. And we have one 3-place function P^3_2 which plays the role of g_0. The result is a 3-place function that returns the successor of the third argument.

The projection functions also allow us to define new functions by reordering or identifying arguments. For instance, the function $h(x) = \text{add}(x, x)$ can be defined as

$$h(x_0) = \text{add}(P^1_0(x_0), P^1_0(x_0))$$

Here $k = 2$, $n = 1$, the role of $f(y_0, y_1)$ is played by add, and the roles of $g_0(x_0)$ and $g_1(x_0)$ are both played by $P^1_0(x_0)$, the one-place projection function (aka the identity function).

If $f(y_0, y_1)$ is a function we already have, we can define the function $h(x_0, x_1) = f(x_1, x_0)$ by

$$h(x_0, x_1) = f(P^2_1(x_0, x_1), P^2_0(x_0, x_1)).$$

Here $k = 2$, $n = 2$, and the roles of g_0 and g_1 are played by P^2_1 and P^2_0, respectively.

You may also worry that $g_0, \ldots, g_{k-1}$ are all required to have the same arity n. (Remember that the *arity* of a function is the number of arguments; an n-place function has arity n.) But adding the projection functions provides the desired flexibility. For example, suppose f and g are 3-place functions and h is the 2-place function defined by

$$h(x, y) = f(x, g(x, x, y), y).$$

The definition of h can be rewritten with the projection functions, as

$$h(x, y) = f(P^2_0(x, y), g(P^2_0(x, y), P^2_0(x, y), P^2_1(x, y)), P^2_1(x, y)).$$

Then h is the composition of f with P^2_0, l, and P^2_1, where

$$l(x, y) = g(P^2_0(x, y), P^2_0(x, y), P^2_1(x, y)),$$

i.e., l is the composition of g with P^2_0, P^2_0, and P^2_1.

rec.4 Primitive Recursion Functions

cmp:rec:prf:sec

Let us record again how we can define new functions from existing ones using primitive recursion and composition.

cmp:rec:prf:defn:primitive-recursion

Definition rec.1. Suppose f is a k-place function ($k \geq 1$) and g is a $(k+2)$-place function. The function defined by *primitive recursion from f and g* is the $(k+1)$-place function h defined by the equations

$$\begin{aligned} h(x_0, \ldots, x_{k-1}, y) &= f(x_0, \ldots, x_{k-1}) \\ h(x_0, \ldots, x_{k-1}, y+1) &= g(x_0, \ldots, x_{k-1}, y, h(x_0, \ldots, x_{k-1}, y)) \end{aligned}$$

Definition rec.2. Suppose f is a k-place function, and $g_0, \ldots, g_{k-1}$ are k functions which are all n-place. The function defined by *composition from f and $g_0, \ldots, g_{k-1}$* is the n-place function h defined by cmp:rec:prf: defn:composition

$$h(x_0, \ldots, x_{n-1}) = f(g_0(x_0, \ldots, x_{n-1}), \ldots, g_{k-1}(x_0, \ldots, x_{n-1})).$$

In addition to succ and the projection functions

$$P^n_i(x_0, \ldots, x_{n-1}) = x_i,$$

for each natural number n and $i < n$, we will include among the primitive recursive functions the function $\text{zero}(x) = 0$.

Definition rec.3. The set of primitive recursive functions is the set of functions from $\mathbb{N}^n$ to $\mathbb{N}$, defined inductively by the following clauses:

1. zero is primitive recursive.
2. succ is primitive recursive.
3. Each projection function P^n_i is primitive recursive.
4. If f is a k-place primitive recursive function and $g_0, \ldots, g_{k-1}$ are n-place primitive recursive functions, then the composition of f with $g_0, \ldots, g_{k-1}$ is primitive recursive.
5. If f is a k-place primitive recursive function and g is a $k+2$-place primitive recursive function, then the function defined by primitive recursion from f and g is primitive recursive.

explanation

Put more concisely, the set of primitive recursive functions is the smallest set containing zero, succ, and the projection functions P^n_j, and which is closed under composition and primitive recursion.

Another way of describing the set of primitive recursive functions keeps track of the "stage" at which a function enters the set. Let S_0 denote the set of starting functions: zero, succ, and the projections. Once S_i has been defined, let S_{i+1} be the set of all functions you get by applying a single instance of composition or primitive recursion to functions in S_i. Then

$$S = \bigcup_{i \in \mathbb{N}} S_i$$

is the set of all primitive recursive functions

Let us verify that add is a primitive recursive function.

Proposition rec.4. *The addition function* $\text{add}(x, y) = x + y$ *is primitive recursive.*

Proof. We already have a primitive recursive definition of add in terms of two functions f and g which matches the format of Definition rec.1:

$$\begin{aligned}\mathrm{add}(x_0, 0) &= f(x_0) = x_0\\ \mathrm{add}(x_0, y+1) &= g(x_0, y, \mathrm{add}(x_0, y)) = \mathrm{succ}(\mathrm{add}(x_0, y))\end{aligned}$$

So add is primitive recursive provided f and g are as well. $f(x_0) = x_0 = P^1_0(x_0)$, and the projection functions count as primitive recursive, so f is primitive recursive. The function g is the three-place function $g(x_0, y, z)$ defined by

$$g(x_0, y, z) = \mathrm{succ}(z).$$

This does not yet tell us that g is primitive recursive, since g and succ are not quite the same function: succ is one-place, and g has to be three-place. But we can define g "officially" by composition as

$$g(x_0, y, z) = \mathrm{succ}(P^3_2(x_0, y, z))$$

Since succ and P^3_2 count as primitive recursive functions, g does as well, since it can be defined by composition from primitive recursive functions. □

cmp:rec:prf: prop:mult-pr

Proposition rec.5. *The multiplication function* $\mathrm{mult}(x, y) = x \cdot y$ *is primitive recursive.*

Proof. Exercise. □

Problem rec.1. Prove Proposition rec.5 by showing that the primitive recursive definition of mult is can be put into the form required by Definition rec.1 and showing that the corresponding functions f and g are primitive recursive.

Example rec.6. Here's our very first example of a primitive recursive definition:

$$\begin{aligned}h(0) &= 1\\ h(y+1) &= 2 \cdot h(y).\end{aligned}$$

This function cannot fit into the form required by Definition rec.1, since $k = 0$. The definition also involves the constants 1 and 2. To get around the first problem, let's introduce a dummy argument and define the function h':

$$\begin{aligned}h'(x_0, 0) &= f(x_0) = 1\\ h'(x_0, y+1) &= g(x_0, y, h'(x_0, y)) = 2 \cdot h'(x_0, y).\end{aligned}$$

The function $f(x_0) = 1$ can be defined from succ and zero by composition: $f(x_0) = \mathrm{succ}(\mathrm{zero}(x_0))$. The function g can be defined by composition from $g'(z) = 2 \cdot z$ and projections:

$$g(x_0, y, z) = g'(P^3_2(x_0, y, z))$$

and g' in turn can be defined by composition as

$$g'(z) = \text{mult}(g''(z), P^1_0(z))$$

and

$$g''(z) = \text{succ}(f(z)),$$

where f is as above: $f(z) = \text{succ}(\text{zero}(z))$. Now that we have h' we can use composition again to let $h(y) = h'(P^1_0(y), P^1_0(y))$. This shows that h can be defined from the basic functions using a sequence of compositions and primitive recursions, so h is primitive recursive.

rec.5 Primitive Recursion Notations

cmp:rec:not:sec

One advantage to having the precise inductive description of the primitive recursive functions is that we can be systematic in describing them. For example, we can assign a "notation" to each such function, as follows. Use symbols zero, succ, and P^n_i for zero, successor, and the projections. Now suppose f is defined by composition from a k-place function h and n-place functions g_0, $\ldots$, g_{k-1}, and we have assigned notations H, G_0, $\ldots$, G_{k-1} to the latter functions. Then, using a new symbol $\text{Comp}_{k,n}$, we can denote the function f by $\text{Comp}_{k,n}[H, G_0, \ldots, G_{k-1}]$. For the functions defined by primitive recursion, we can use analogous notations of the form $\text{Rec}_k[G, H]$, where $k+1$ is the arity of the function being defined. With this setup, we can denote the addition function by

$$\text{Rec}_2[P^1_0, \text{Comp}_{1,3}[\text{succ}, P^3_2]].$$

Having these notations sometimes proves useful.

Problem rec.2. Give the complete primitive recursive notation for mult.

rec.6 Primitive Recursive Functions are Computable

cmp:rec:cmp:sec

Suppose a function h is defined by primitive recursion

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

and suppose the functions f and g are computable. (We use $\vec{x}$ to abbreviate x_0, $\ldots$, x_{k-1}.) Then $h(\vec{x}, 0)$ can obviously be computed, since it is just $f(\vec{x})$ which we assume is computable. $h(\vec{x}, 1)$ can then also be computed, since $1 = 0 + 1$ and so $h(\vec{x}, 1)$ is just

$$h(\vec{x}, 1) = g(\vec{x}, 0, h(\vec{x}, 0)) = g(\vec{x}, 0, f(\vec{x})).$$

We can go on in this way and compute

$$\begin{aligned}
h(\vec{x}, 2) &= g(\vec{x}, 1, h(\vec{x}, 1)) = g(\vec{x}, 1, g(\vec{x}, 0, f(\vec{x}))) \\
h(\vec{x}, 3) &= g(\vec{x}, 2, h(\vec{x}, 2)) = g(\vec{x}, 2, g(\vec{x}, 1, g(\vec{x}, 0, f(\vec{x})))) \\
h(\vec{x}, 4) &= g(\vec{x}, 3, h(\vec{x}, 3)) = g(\vec{x}, 3, g(\vec{x}, 2, g(\vec{x}, 1, g(\vec{x}, 0, f(\vec{x}))))) \\
&\vdots
\end{aligned}$$

Thus, to compute $h(\vec{x}, y)$ in general, successively compute $h(\vec{x}, 0)$, $h(\vec{x}, 1)$, ..., until we reach $h(\vec{x}, y)$.

Thus, a primitive recursive definition yields a new computable function if the functions f and g are computable. Composition of functions also results in a computable function if the functions f and g_i are computable.

Since the basic functions zero, succ, and P_i^n are computable, and composition and primitive recursion yield computable functions from computable functions, this means that every primitive recursive function is computable.

rec.7 Examples of Primitive Recursive Functions

cmp:rec:exa:sec

We already have some examples of primitive recursive functions: the addition and multiplication functions add and mult. The identity function $\mathrm{id}(x) = x$ is primitive recursive, since it is just P_0^1. The constant functions $\mathrm{const}_n(x) = n$ are primitive recursive since they can be defined from zero and succ by successive composition. This is useful when we want to use constants in primitive recursive definitions, e.g., if we want to define the function $f(x) = 2 \cdot x$ can obtain it by composition from $\mathrm{const}_n(x)$ and multiplication as $f(x) = \mathrm{mult}(\mathrm{const}_2(x), P_0^1(x))$. We'll make use of this trick from now on.

Proposition rec.7. *The exponentiation function* $\exp(x, y) = x^y$ *is primitive recursive.*

Proof. We can define exp primitive recursively as

$$\begin{aligned}
\exp(x, 0) &= 1 \\
\exp(x, y + 1) &= \mathrm{mult}(x, \exp(x, y)).
\end{aligned}$$

Strictly speaking, this is not a recursive definition from primitive recursive functions. Officially, though, we have:

$$\begin{aligned}
\exp(x, 0) &= f(x) \\
\exp(x, y + 1) &= g(x, y, \exp(x, y)).
\end{aligned}$$

where

$$\begin{aligned}
f(x) &= \mathrm{succ}(\mathrm{zero}(x)) = 1 \\
g(x, y, z) &= \mathrm{mult}(P_0^3(x, y, z), P_2^3(x, y, z) = x \cdot z
\end{aligned}$$

and so f and g are defined from primitive recursive functions by composition. □

Proposition rec.8. *The predecessor function* $\mathrm{pred}(y)$ *defined by*

$$\mathrm{pred}(y) = \begin{cases} 0 & \text{if } y = 0 \\ y - 1 & \text{otherwise} \end{cases}$$

is primitive recursive.

Proof. Note that

$$\begin{aligned} \mathrm{pred}(0) &= 0 \\ \mathrm{pred}(y+1) &= y \end{aligned}$$

This is almost a primitive recursive definition. It does not, strictly speaking, fit into the pattern of definition by primitive recursion, since that pattern requires at least one extra argument x. It is also odd in that it does not actually use $\mathrm{pred}(y)$ in the definition of $\mathrm{pred}(y+1)$. But we can first define $\mathrm{pred}'(x, y)$ by

$$\begin{aligned} \mathrm{pred}'(x, 0) &= \mathrm{zero}(x) = 0 \\ \mathrm{pred}'(x, y+1) &= P^3_1(x, y, \mathrm{pred}'(x, y)) = y \end{aligned}$$

and then define pred from it by composition, e.g., as $\mathrm{pred}(x) = \mathrm{pred}'(\mathrm{zero}(x), P^1_0(x))$. □

Proposition rec.9. *The factorial function* $\mathrm{fac}(x) = x! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot x$ *is primitive recursive.*

Proof. The obvious primitive recursive definition is

$$\begin{aligned} \mathrm{fac}(0) &= 1 \\ \mathrm{fac}(y+1) &= !y \cdot (y+1) \end{aligned}$$

Officially, we have to first define a two-place function h

$$\begin{aligned} h(x, 0) &= \mathrm{const}_1(x) \\ h(x, y) &= g(x, y, h(x, y)) \end{aligned}$$

where $g(x, y, z) = \mathrm{mult}(P^3_2(x, y, z), \mathrm{succ}(P^3_1(x, y, z)$ and then let

$$\mathrm{fac}(y) = h(P^1_0(y), P^1_0(y))$$

From now on we'll be a bit more lessez-faire and not give the official definitions by composition and primitive recursion. □

Proposition rec.10. *Truncated subtraction, $x \dot{-} y$, defined by*

$$x \dot{-} y = \begin{cases} 0 & \text{if } x > y \\ x - y & \text{otherwise} \end{cases}$$

is primitive recursive.

Proof. We have

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= \operatorname{pred}(x \dot{-} y) \end{aligned}$$

□

Proposition rec.11. *The distance between x and y, $|x - y|$, is primitive recursive.*

Proof. We have $|x - y| = (x \dot{-} y) + (y \dot{-} x)$, so the distance can be defined by composition from $+$ and $\dot{-}$, which are primitive recursive. □

Proposition rec.12. *The maximum of x and y, $\max(x, y)$, is primitive recursive.*

Proof. We can define $\max(x, y)$ by composition from $+$ and $\dot{-}$ by

$$\max(x, y) = x + (y \dot{-} x).$$

If x is the maximum, i.e., $x \geq y$, then $y \dot{-} x = 0$, so $x + (y \dot{-} x) = x + 0 = x$. If y is the maximum, then $y \dot{-} x = y - x$, and so $x + (y \dot{-} x) = x + (y - x) = y$. □

cmp:rec:exa: prop:min-pr

Proposition rec.13. *The minimum of x and y, $\min(x, y)$, is primitive recursive.*

Proof. Prove Proposition rec.13. □

Problem rec.3. Show that

$$f(x, y) = 2^{(2^{\cdot^{\cdot^{\cdot^{2^x}}}})} \Big\} y \text{ 2's}$$

is primitive recursive.

Problem rec.4. Show that integer division $d(x, y) = \lfloor x/y \rfloor$ (i.e., division, where you disregard everything after the decimal point) is primitive recursive. When $y = 0$, we stipulate $d(x, y) = 0$. Give an explicit definition of d using primitive recursion and composition.

Proposition rec.14. *The set of primitive recursive functions is closed under the following two operations:*

1. *Finite sums: if $f(\vec{x}, z)$ is primitive recursive, then so is the function*

$$g(\vec{x}, y) = \sum_{z=0}^{y} f(\vec{x}, z).$$

2. *Finite products: if $f(\vec{x}, z)$ is primitive recursive, then so is the function*

$$h(\vec{x}, y) = \prod_{z=0}^{y} f(\vec{x}, z).$$

Proof. For example, finite sums are defined recursively by the equations

$$\begin{aligned} g(\vec{x}, 0) &= f(\vec{x}, 0) \\ g(\vec{x}, y+1) &= g(\vec{x}, y) + f(\vec{x}, y+1). \end{aligned}$$

□

rec.8 Primitive Recursive Relations

cmp:rec:prr:sec

Definition rec.15. A relation $R(\vec{x})$ is said to be primitive recursive if its characteristic function,

$$\chi_R(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}) \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive.

In other words, when one speaks of a primitive recursive relation $R(\vec{x})$, one is referring to a relation of the form $\chi_R(\vec{x}) = 1$, where χ_R is a primitive recursive function which, on any input, returns either 1 or 0. For example, the relation IsZero(x), which holds if and only if $x = 0$, corresponds to the function χ_{IsZero}, defined using primitive recursion by

$$\chi_{\text{IsZero}}(0) = 1, \quad \chi_{\text{IsZero}}(x+1) = 0.$$

It should be clear that one can compose relations with other primitive recursive functions. So the following are also primitive recursive:

1. The equality relation, $x = y$, defined by IsZero$(|x - y|)$
2. The less-than relation, $x \leq y$, defined by IsZero$(x \mathbin{\dot{-}} y)$

Proposition rec.16. *The set of primitive recursive relations is closed under boolean operations, that is, if $P(\vec{x})$ and $Q(\vec{x})$ are primitive, so are*

1. $\neg R(\vec{x})$

2. $P(\vec{x}) \wedge Q(\vec{x})$

3. $P(\vec{x}) \vee Q(\vec{x})$

4. $P(\vec{x}) \to Q(\vec{x})$

Proof. Suppose $P(\vec{x})$ and $Q(\vec{x})$ are primitive recursive, i.e., their characteristic functions χ_P and χ_Q are. We have to show that the characteristic functions of $\neg R(\vec{x})$, etc., are also primitive recursive.

$$\chi_{\neg P}(\vec{x}) = \begin{cases} 0 & \text{if } \chi_P(\vec{x}) = 1 \\ 1 & \text{otherwise} \end{cases}$$

We can define $\chi_{\neg P}(\vec{x})$ as $1 \dot{-} \chi_P(\vec{x})$.

$$\chi_{P \wedge Q}(\vec{x}) = \begin{cases} 1 & \text{if } \chi_P(\vec{x}) = \chi_Q(\vec{x}) = 1 \\ 0 & \text{otherwise} \end{cases}$$

We can define $\chi_{P \wedge Q}(\vec{x})$ as $\chi_P(\vec{x}) \cdot \chi_Q(\vec{x})$ or as $\min(\chi_P(\vec{x}), \chi_Q(\vec{x}))$.

Similarly, $\chi_{P \vee Q}(\vec{x}) = \max(\chi_P(\vec{x}), \chi_Q(\vec{x}))$ and $\chi_{P \vee Q}(\vec{x}) = \max(1 \dot{-} \chi_P(\vec{x}), \chi_Q(\vec{x}))$. □

Proposition rec.17. *The set of primitive recursive relations is closed under bounded quantification, i.e., if $R(\vec{x}, z)$ is a primitive recursive relation, then so are the relations $(\forall z < y)\ R(\vec{x}, z)$ and $(\exists z < y)\ R(\vec{x}, z)$.*

($(\forall z < y)\ R(\vec{x}, z)$ holds of $\vec{x}$ and y if and only if $R(\vec{x}, z)$ holds for every z less than y, and similarly for $(\exists z < y)\ R(\vec{x}, z)$.)

Proof. By convention, we take $(\forall z < 0)\ R(\vec{x}, z)$ to be true (for the trivial reason that there are no z less than 0) and $(\exists z < 0)\ R(\vec{x}, z)$ to be false. A universal quantifier functions just like a finite product or iterated minimum, i.e., if $P(\vec{x}, y) \Leftrightarrow (\forall z < y)\ R(\vec{x}, z)$ then $\chi_P(\vec{x}, y)$ can be defined by

$$\begin{aligned} \chi_P(\vec{x}, 0) &= 1 \\ \chi_P(\vec{x}, y+1) &= \min(\chi_P(\vec{x}, y), \chi_R(\vec{x}, y+1))). \end{aligned}$$

Bounded existential quantification can similarly be defined using max. Alternatively, it can be defined from bounded universal quantification, using the equivalence $(\exists z < y)\ R(\vec{x}, z) \leftrightarrow \neg(\forall z < y)\ \neg R(\vec{x}, z)$. Note that, for example, a bounded quantifier of the form $(\exists x \leq y)\ \ldots x \ldots$ is equivalent to $(\exists x < y + 1)\ \ldots x \ldots$. □

Another useful primitive recursive function is the conditional function, $\mathrm{cond}(x, y, z)$, defined by

$$\mathrm{cond}(x, y, z) = \begin{cases} y & \text{if } x = 0 \\ z & \text{otherwise.} \end{cases}$$

This is defined recursively by

$$\mathrm{cond}(0, y, z) = y, \quad \mathrm{cond}(x + 1, y, z) = z.$$

One can use this to justify definitions of primitive recursive functions by cases from primitive recursive relations:

Proposition rec.18. *If $g_0(\vec{x}), \ldots, g_m(\vec{x})$ are functions, and $R_1(\vec{x}), \ldots, R_{m-1}(\vec{x})$ are primitive recursive relations, then the function f defined by*

$$f(\vec{x}) = \begin{cases} g_0(\vec{x}) & \text{if } R_0(\vec{x}) \\ g_1(\vec{x}) & \text{if } R_1(\vec{x}) \text{ and not } R_0(\vec{x}) \\ \vdots & \\ g_{m-1}(\vec{x}) & \text{if } R_{m-1}(\vec{x}) \text{ and none of the previous hold} \\ g_m(\vec{x}) & \text{otherwise} \end{cases}$$

is also primitive recursive.

Proof. When $m = 1$, this is just the function defined by

$$f(\vec{x}) = \mathrm{cond}(\chi_{\neg R_0}(\vec{x}), g_0(\vec{x}), g_1(\vec{x})).$$

For m greater than 1, one can just compose definitions of this form. □

rec.9 Bounded Minimization

explanation

It is often useful to define a function as the least number satisfying some property or relation P. If P is decidable, we can compute this function simply by trying out all the possible numbers, 0, 1, 2, ..., until we find the least one satisfying P. This kind of unbounded search takes us out of the realm of primitive recursive functions. However, if we're only interested in the least number *less than some independently given bound*, we stay primitive recursive. In other words, and a bit more generally, suppose we have a primitive recursive relation $R(x, z)$. Consider the function that maps x and y to the least $z < y$ such that $R(x, z)$. It, too, can be computed, by testing whether $R(x, 0)$, $R(x, 1)$, ..., $R(x, y - 1)$. But why is it primitive recursive?

cmp:rec:bmi:sec

Proposition rec.19. *If $R(\vec{x}, z)$ is primitive recursive, so is the function $m_R(\vec{x}, y)$ which returns the least z less than y such that $R(\vec{x}, z)$ holds, if there is one, and y otherwise. We will write the function m_R as*

$$(\min z < y)\, R(\vec{x}, z),$$

Proof. Note than there can be no $z < 0$ such that $R(\vec{x}, z)$ since there is no $z < 0$ at all. So $m_R(\vec{x}, 0) = 0$.

In case the bound is of the form $y + 1$ we have three cases: (a) There is a $z < y$ such that $R(\vec{x}, z)$, in which case $m_R(\vec{x}, z) = m_R(\vec{x}, y)$. (b) There is no

such $z < y$ but $R(\vec{x}, y)$ holds, then $m_R(\vec{x}, y+1) = y$. (c) There is no $z < y+1$ such that $R(\vec{x}, z)$, then $m_R(\vec{z}, y+1) = y+1$. Note that there is a $z < y$ such that $R(\vec{x}, z)$ iff $m_R(\vec{x}, y) \neq y$. So,

$$m_R(\vec{x}, 0) = 0$$
$$m_R(\vec{x}, y+1) = \begin{cases} m_R(\vec{z}, y) & \text{if } m_R(\vec{x}, y) \neq y \\ y & \text{if } m_R(\vec{x}, y) = y \text{ and } R(\vec{x}, y) \\ y+1 & \text{otherwise.} \end{cases}$$

□

Problem rec.5. Suppose $R(\vec{x}, z)$ is primitive recursive. Define the function $m'_R(\vec{x}, y)$ which returns the least z less than y such that $R(\vec{x}, z)$ holds, if there is one, and 0 otherwise, by primitive recursion from χ_R.

rec.10 Primes

cmp:rec:pri: sec

Bounded quantification and bounded minimization provide us with a good deal of machinery to show that natural functions and relations are primitive recursive. For example, consider the relation relation "x divides y", written $x \mid y$. The relation $x \mid y$ holds if division of y by x is possible without remainder, i.e., if y is an integer multiple of x. (If it doesn't hold, i.e., the remainder when dividing x by y is > 0, we write $x \nmid y$.) In other words, $x \mid y$ iff for some z, $x \cdot z = y$. Obviously, any such z, if it exists, must be $\leq y$. So, we have that $x \mid y$ iff for some $z \leq y$, $x \cdot z = y$. We can define the relation $x \mid y$ by bounded existential quantification from $=$ and multiplication by

$$x \mid y \Leftrightarrow (\exists z \leq y)\ (x \cdot z) = y.$$

We've thus shown that $x \mid y$ is primitive recursive.

A natural number x is *prime* if it is neither 0 nor 1 and is only divisible by 1 and itself. In other words, prime numbers are such that, whenever $y \mid x$, either $y = 1$ or $y = x$. To test if x is prime, we only have to check if $y \mid x$ for all $y \leq x$, since if $y > x$, then automatically $y \nmid x$. So, the relation $\mathrm{Prime}(x)$, which holds iff x is prime, can be defined by

$$\mathrm{Prime}(x) \Leftrightarrow x \geq 2 \wedge (\forall y \leq x)\ (y \mid x \rightarrow y = 1 \vee y = x)$$

and is thus primitive recursive.

The primes are 2, 3, 5, 7, 11, etc. Consider the function $p(x)$ which returns the xth prime in that sequence, i.e., $p(0) = 2$, $p(1) = 3$, $p(2) = 5$, etc. (For convenience we will often write $p(x)$ as p_x ($p_0 = 2$, $p_1 = 3$, etc.)

If we had a function nextPrime(x), which returns the first prime number larger than x, p can be easily defined using primitive recursion:

$$p(0) = 2$$
$$p(x+1) = \mathrm{nextPrime}(p(x))$$

Since nextPrime(x) is the least y such that $y > x$ and y is prime, it can be easily computed by unbounded search. But it can also be defined by bounded minimization, thanks to a result due to Euclid: there is always a prime number between x and $x! + 1$.

$$\text{nextPrime(x)} = (\min y \leq x! + 1)\,(y > x \wedge \text{Prime}(y)).$$

This shows, that nextPrime(x) and hence $p(x)$ are (not just computable but) primitive recursive.

(If you're curious, here's a quick proof of Euclid's theorem. Suppose p_n is the largest prime $\leq x$ and consider the product $p = p_0 \cdot p_1 \cdot \cdots \cdot p_n$ of all primes $\leq x$. Either $p + 1$ is prime or there is a prime between x and $p + 1$. Why? Suppose $p + 1$ is not prime. Then some prime number $q \mid p + 1$ where $q < p + 1$. None of the primes $\leq x$ divide $p + 1$. (By definition of p, each of the primes $p_i \leq x$ divides p, i.e., with remainder 0. So, each of the primes $p_i \leq x$ divides $p + 1$ with remainder 1, and so $p_i \nmid p + 1$.) Hence, q is a prime $> x$ and $< p + 1$. And $p \leq x!$, so there is a prime $> x$ and $\leq x! + 1$.)

Problem rec.6. Define integer division $d(x, y)$ using bounded minimization.

rec.11 Sequences

cmp:rec:seq:sec

The set of primitive recursive functions is remarkably robust. But we will be able to do even more once we have developed a adequate means of handling *sequences*. We will identify finite sequences of natural numbers with natural numbers in the following way: the sequence $\langle a_0, a_1, a_2, \ldots, a_k \rangle$ corresponds to the number

$$p_0^{a_0+1} \cdot p_1^{a_1+1} \cdot p_2^{a_2+1} \cdot \cdots \cdot p_k^{a_k+1}.$$

We add one to the exponents to guarantee that, for example, the sequences $\langle 2, 7, 3 \rangle$ and $\langle 2, 7, 3, 0, 0 \rangle$ have distinct numeric codes. We can take both 0 and 1 to code the empty sequence; for concreteness, let Λ denote 0.

The reason that this coding of sequences works is the so-called Fundamental Theorem of Arithmetic: every natural number $n \geq 2$ can be written in one and only one way in the form

$$n = p_0^{a_0} \cdot p_1^{a_1} \cdot \cdots \cdot p_k^{a_k}$$

with $a_k \geq 1$. This guarantees that the mapping $\langle\rangle(a_0, \ldots, a_k) = \langle a_0, \ldots, a_k \rangle$ is injective: different sequences are mapped to different numbers; to each number only at most one sequence corresponds.

We'll now show that the operations of determining the length of a sequence, determining its ith element, appending an element to a sequence, and concatenating two sequences, are all primitive recursive.

Proposition rec.20. *The function* len(s)*, which returns the length of the sequence s, is primitive recursive.*

Proof. Let $R(i, s)$ be the relation defined by

$$R(i, s) \text{ iff } p_i \mid s \wedge p_{i+1} \nmid s.$$

R is clearly primitive recursive. Whenever s is the code of a non-empty sequence, i.e.,

$$s = p_0^{a_0+1} \cdot \dots \cdot p_k^{a_k+1},$$

$R(i, s)$ holds if p_i is the largest prime such that $p_i \mid s$, i.e., $i = k$. The length of s thus is $i + 1$ iff p_i is the largest prime that divides s, so we can let

$$\text{len}(s) = \begin{cases} 0 & \text{if } s = 0 \text{ or } s = 1 \\ 1 + (\min i < s)\, R(i, s) & \text{otherwise} \end{cases}$$

We can use bounded minimization, since there is only one i that satisfies $R(s, i)$ when s is a code of a sequence, and if i exists it is less than s itself. □

Proposition rec.21. *The function* $\text{append}(s, a)$*, which returns the result of appending* a *to the sequence* s*, is primitive recursive.*

Proof. append can be defined by:

$$\text{append}(s, a) = \begin{cases} 2^{a+1} & \text{if } s = 0 \text{ or } s = 1 \\ s \cdot p_{\text{len}(s)}^{a+1} & \text{otherwise.} \end{cases}$$

□

Proposition rec.22. *The function* $\text{element}(s, i)$*, which returns the* i*th element of* s *(where the initial element is called the* 0*th), or* 0 *if* i *is greater than or equal to the length of* s*, is primitive recursive.*

Proof. Note that a is the ith element of s iff p_i^{a+1} is the largest power of p_i that divides s, i.e., $p_i^{a+1} \mid s$ but $p_i^{a+2} \nmid s$. So:

$$\text{element}(s, i) = \begin{cases} 0 & \text{if } i \geq \text{len}(s) \\ (\min a < s)\, (p_i^{a+2} \nmid s) & \text{otherwise.} \end{cases}$$

□

Instead of using the official names for the functions defined above, we introduce a more compact notation. We will use $(s)_i$ instead of $\text{element}(s, i)$, and $\langle s_0, \dots, s_k \rangle$ to abbreviate

$$\text{append}(\text{append}(\dots \text{append}(\Lambda, s_0) \dots), s_k).$$

Note that if s has length k, the elements of s are $(s)_0, \dots, (s)_{k-1}$.

Proposition rec.23. *The function* $\text{concat}(s, t)$*, which concatenates two sequences, is primitive recursive.*

Proof. We want a function concat with the property that

$$\text{concat}(\langle a_0, \ldots, a_k\rangle, \langle b_0, \ldots, b_l\rangle) = \langle a_0, \ldots, a_k, b_0, \ldots, b_l\rangle.$$

We'll use a "helper" function $\text{hconcat}(s, t, n)$ which concatenates the first n symbols of t to s. This function can be defined by primitive recursion as follows:

$$\begin{aligned}\text{hconcat}(s, t, 0) &= s\\ \text{hconcat}(s, t, n+1) &= \text{append}(\text{hconcat}(s, t, n), (t)_n)\end{aligned}$$

Then we can define concat by

$$\text{concat}(s, t) = \text{hconcat}(s, t, \text{len}(t)).$$

□

We will write $s \frown t$ instead of $\text{concat}(s, t)$.

It will be useful for us to be able to bound the numeric code of a sequence in terms of its length and its largest element. Suppose s is a sequence of length k, each element of which is less than equal to some number x. Then s has at most k prime factors, each at most p_{k-1}, and each raised to at most $x + 1$ in the prime factorization of s. In other words, if we define

$$\text{sequenceBound}(x, k) = p_{k-1}^{k\cdot(x+1)},$$

then the numeric code of the sequence s described above is at most $\text{sequenceBound}(x, k)$.

Having such a bound on sequences gives us a way of defining new functions using bounded search. For example, we can define concat using bounded search. All we need to do is write down a primitive recursive *specification* of the object (number of the concatenated sequence) we are looking for, and a bound on how far to look. The following works:

$$\begin{aligned}\text{concat}(s, t) = (\min v < \text{sequenceBound}(s + t, \text{len}(s) + \text{len}(t)))&\\ (\text{len}(v) = \text{len}(s) + \text{len}(t) \wedge&\\ (\forall i < \text{len}(s))\ ((v)_i = (s)_i) \wedge&\\ (\forall j < \text{len}(t))\ ((v)_{\text{len}(s)+j} = (t)_j))&\end{aligned}$$

Problem rec.7. Show that there is a primitive recursive function $\text{sconcat}(s)$ with the property that

$$\text{sconcat}(\langle s_0, \ldots, s_k\rangle) = s_0 \frown \ldots \frown s_k.$$

Problem rec.8. Show that there is a primitive recursive function $\text{tail}(s)$ with the property that

$$\begin{aligned}\text{tail}(\Lambda) &= 0 \text{ and}\\ \text{tail}(\langle s_0, \ldots, s_k\rangle) &= \langle s_1, \ldots, s_k\rangle.\end{aligned}$$

cmp:rec:seq: prop:subseq

Proposition rec.24. *The function* subseq(s, i, n) *which returns the subsequence of s of length n beginning at the ith element, is primitive recursive.*

Proof. Exercise. □

Problem rec.9. Prove Proposition rec.24.

rec.12 Trees

cmp:rec:tre: sec

Sometimes it is useful to represent trees as natural numbers, just like we can represent sequences by numbers and properties of and operations on them by primitive recursive relations and functions on their codes. We'll use sequences and their codes to do this. A tree can be either a single node (possibly with a label) or else a node (possibly with a label) connected to a number of subtrees. The node is called the *root* of the tree, and the subtrees it is connected to its *immediate subtrees*.

We code trees recursively as a sequence $\langle k, d_1, \ldots, d_k \rangle$, where k is the number of immediate subtrees and d_1, …, d_k the codes of the immediate subtrees. If the nodes have labels, they can be included after the immediate subtrees. So a tree consisting just of a single node with label l would be coded by $\langle 0, l \rangle$, and a tree consisting of a root (labelled l_1) connected to two single nodes (labelled l_2, l_3) would be coded by $\langle 2, \langle 0, l_2 \rangle, \langle 0, l_3 \rangle, l_1 \rangle$.

cmp:rec:tre: prop:subtreeseq

Proposition rec.25. *The function* SubtreeSeq(t)*, which returns the code of a sequence the elements of which are the codes of all subtrees of the tree with code t, is primitive recursive.*

Proof. First note that $\text{ISubtrees}(t) = \text{subseq}(t, 1, (t)_0)$ is primitive recursive and returns the codes of the immediate subtrees of a tree t. Now we can define a helper function $\text{hSubtreeSeq}(t, n)$ which computes the sequence of all subtrees which are n nodes remove from the root. The sequence of subtrees of t which is 0 nodes removed from the root—in other words, begins at the root of t—is the sequence consisting just of t. To obtain a sequence of all level $n+1$ subtrees of t, we concatenate the level n subtrees with a sequence consisting of all immediate subtrees of the level n subtrees. To get a list of all these, note that if $f(x)$ is a primitive recursive function returning codes of sequences, then $g_f(s, k) = f((s)_0) \frown \ldots \frown f((s)_k)$ is also primivive recursive:

$$\begin{aligned} g(s, 0) &= f((s)_0) \\ g(s, k+1) &= g(s, k) \frown f((s)_{k+1}) \end{aligned}$$

For instance, if s is a sequence of trees, then $h(s) = g_{\text{ISubtrees}}(s, \text{len}(s))$ gives the sequence of the immediate subtrees of the elements of s. We can use it to define hSubtreeSeq by

$$\begin{aligned} \text{hSubtreeSeq}(t, 0) &= \langle t \rangle \\ \text{hSubtreeSeq}(t, n+1) &= \text{hSubtreeSeq}(t, n) \frown h(\text{hSubtree}(t, n)). \end{aligned}$$

The maximum level of subtrees in a tree coded by t, i.e., the maximum distance between the root and a leaf node, is bounded by the code t. So a sequence of codes of all subtrees of the tree coded by t is given by $\text{hSubtreeSeq}(t, t)$. □

Problem rec.10. The definition of hSubtreeSeq in the proof of Proposition rec.25 in general includes repetitions. Give an alternative definition which guarantees that the code of a subtree occurs only once in the resulting list.

rec.13 Other Recursions

cmp:rec:ore:sec

Using pairing and sequencing, we can justify more exotic (and useful) forms of primitive recursion. For example, it is often useful to define two functions simultaneously, such as in the following definition:

$$\begin{aligned} h_0(\vec{x}, 0) &= f_0(\vec{x}) \\ h_1(\vec{x}, 0) &= f_1(\vec{x}) \\ h_0(\vec{x}, y+1) &= g_0(\vec{x}, y, h_0(\vec{x}, y), h_1(\vec{x}, y)) \\ h_1(\vec{x}, y+1) &= g_1(\vec{x}, y, h_0(\vec{x}, y), h_1(\vec{x}, y)) \end{aligned}$$

This is an instance of *simultaneous recursion*. Another useful way of defining functions is to give the value of $h(\vec{x}, y+1)$ in terms of *all* the values $h(\vec{x}, 0)$, $\ldots$, $h(\vec{x}, y)$, as in the following definition:

$$\begin{aligned} h(\vec{x}, 0) &= f(\vec{x}) \\ h(\vec{x}, y+1) &= g(\vec{x}, y, \langle h(\vec{x}, 0), \ldots, h(\vec{x}, y)\rangle). \end{aligned}$$

The following schema captures this idea more succinctly:

$$h(\vec{x}, y) = g(\vec{x}, y, \langle h(\vec{x}, 0), \ldots, h(\vec{x}, y-1)\rangle)$$

with the understanding that the last argument to g is just the empty sequence when y is 0. In either formulation, the idea is that in computing the "successor step," the function h can make use of the entire sequence of values computed so far. This is known as a *course-of-values* recursion. For a particular example, it can be used to justify the following type of definition:

$$h(\vec{x}, y) = \begin{cases} g(\vec{x}, y, h(\vec{x}, k(\vec{x}, y))) & \text{if } k(\vec{x}, y) < y \\ f(\vec{x}) & \text{otherwise} \end{cases}$$

In other words, the value of h at y can be computed in terms of the value of h at *any* previous value, given by k.

You should think about how to obtain these functions using ordinary primitive recursion. One final version of primitive recursion is more flexible in that one is allowed to change the *parameters* (side values) along the way:

$$\begin{aligned} h(\vec{x}, y) &= f(\vec{x}) \\ h(\vec{x}, y+1) &= g(\vec{x}, y, h(k(\vec{x}), y)) \end{aligned}$$

This, too, can be simulated with ordinary primitive recursion. (Doing so is tricky. For a hint, try unwinding the computation by hand.)

rec.14 Non-Primitive Recursive Functions

cmp:rec:npr:sec

The primitive recursive functions do not exhaust the intuitively computable functions. It should be intuitively clear that we can make a list of all the unary primitive recursive functions, f_0, f_1, f_2, ... such that we can effectively compute the value of f_x on input y; in other words, the function $g(x, y)$, defined by

$$g(x, y) = f_x(y)$$

is computable. But then so is the function

$$\begin{aligned} h(x) &= g(x, x) + 1 \\ &= f_x(x) + 1. \end{aligned}$$

For each primitive recursive function f_i, the value of h and f_i differ at i. So h is computable, but not primitive recursive; and one can say the same about g. This is an "effective" version of Cantor's diagonalization argument.

One can provide more explicit examples of computable functions that are not primitive recursive. For example, let the notation $g^n(x)$ denote $g(g(\ldots g(x)))$, with n g's in all; and define a sequence $g_0, g_1, \ldots$ of functions by

$$\begin{aligned} g_0(x) &= x + 1 \\ g_{n+1}(x) &= g_n^x(x) \end{aligned}$$

You can confirm that each function g_n is primitive recursive. Each successive function grows much faster than the one before; $g_1(x)$ is equal to $2x$, $g_2(x)$ is equal to $2^x \cdot x$, and $g_3(x)$ grows roughly like an exponential stack of x 2's. Ackermann's function is essentially the function $G(x) = g_x(x)$, and one can show that this grows faster than any primitive recursive function.

Let us return to the issue of enumerating the primitive recursive functions. Remember that we have assigned symbolic notations to each primitive recursive function; so it suffices to enumerate notations. We can assign a natural number $\#(F)$ to each notation F, recursively, as follows:

$$\begin{aligned} \#(0) &= \langle 0 \rangle \\ \#(S) &= \langle 1 \rangle \\ \#(P_i^n) &= \langle 2, n, i \rangle \\ \#(\mathrm{Comp}_{k,l}[H, G_0, \ldots, G_{k-1}]) &= \langle 3, k, l, \#(H), \#(G_0), \ldots, \#(G_{k-1}) \rangle \\ \#(\mathrm{Rec}_l[G, H]) &= \langle 4, l, \#(G), \#(H) \rangle \end{aligned}$$

Here we are using the fact that every sequence of numbers can be viewed as a natural number, using the codes from the last section. The upshot is that every code is assigned a natural number. Of course, some sequences (and hence some numbers) do not correspond to notations; but we can let f_i be the unary primitive recursive function with notation coded as i, if i codes such a notation; and the constant 0 function otherwise. The net result is that we have an explicit way of enumerating the unary primitive recursive functions.

(In fact, some functions, like the constant zero function, will appear more than once on the list. This is not just an artifact of our coding, but also a result of the fact that the constant zero function has more than one notation. We will later see that one can not computably avoid these repetitions; for example, there is no computable function that decides whether or not a given notation represents the constant zero function.)

We can now take the function $g(x, y)$ to be given by $f_x(y)$, where f_x refers to the enumeration we have just described. How do we know that $g(x, y)$ is computable? Intuitively, this is clear: to compute $g(x, y)$, first "unpack" x, and see if it a notation for a unary function; if it is, compute the value of that function on input y.

digression

You may already be convinced that (with some work!) one can write a program (say, in Java or C++) that does this; and now we can appeal to the Church-Turing thesis, which says that anything that, intuitively, is computable can be computed by a Turing machine.

Of course, a more direct way to show that $g(x, y)$ is computable is to describe a Turing machine that computes it, explicitly. This would, in particular, avoid the Church-Turing thesis and appeals to intuition. But, as noted above, working with Turing machines directly is unpleasant. Soon we will have built up enough machinery to show that $g(x, y)$ is computable, appealing to a model of computation that can be *simulated* on a Turing machine: namely, the recursive functions.

rec.15 Partial Recursive Functions

cmp:rec:par:sec

To motivate the definition of the recursive functions, note that our proof that there are computable functions that are not primitive recursive actually establishes much more. The argument was simple: all we used was the fact was that it is possible to enumerate functions $f_0, f_1, \ldots$ such that, as a function of x and y, $f_x(y)$ is computable. So the argument applies to *any class of functions that can be enumerated in such a way.* This puts us in a bind: we would like to describe the computable functions explicitly; but any explicit description of a collection of computable functions cannot be exhaustive!

The way out is to allow *partial* functions to come into play. We will see that it *is* possible to enumerate the partial computable functions. In fact, we already pretty much know that this is the case, since it is possible to enumerate Turing machines in a systematic way. We will come back to our diagonal argument later, and explore why it does not go through when partial functions are included.

The question is now this: what do we need to add to the primitive recursive functions to obtain all the partial recursive functions? We need to do two things:

1. Modify our definition of the primitive recursive functions to allow for partial functions as well.

2. *Add* something to the definition, so that some new partial functions are included.

The first is easy. As before, we will start with zero, successor, and projections, and close under composition and primitive recursion. The only difference is that we have to modify the definitions of composition and primitive recursion to allow for the possibility that some of the terms in the definition are not defined. If f and g are partial functions, we will write $f(x)\downarrow$ to mean that f is defined at x, i.e., x is in the domain of f; and $f(x)\uparrow$ to mean the opposite, i.e., that f is not defined at x. We will use $f(x) \simeq g(x)$ to mean that either $f(x)$ and $g(x)$ are both undefined, or they are both defined and equal. We will use these notations for more complicated terms as well. We will adopt the convention that if h and $g_0, \ldots, g_k$ all are partial functions, then

$$h(g_0(\vec{x}), \ldots, g_k(\vec{x}))$$

is defined if and only if each g_i is defined at $\vec{x}$, and h is defined at $g_0(\vec{x})$, $\ldots$, $g_k(\vec{x})$. With this understanding, the definitions of composition and primitive recursion for partial functions is just as above, except that we have to replace "$=$" by "$\simeq$".

What we will add to the definition of the primitive recursive functions to obtain partial functions is the *unbounded search operator*. If $f(x, \vec{z})$ is any partial function on the natural numbers, define $\mu x\ f(x, \vec{z})$ to be

> the least x such that $f(0, \vec{z}), f(1, \vec{z}), \ldots, f(x, \vec{z})$ are all defined, and $f(x, \vec{z}) = 0$, if such an x exists

with the understanding that $\mu x\ f(x, \vec{z})$ is undefined otherwise. This defines $\mu x\ f(x, \vec{z})$ uniquely.

Note that our definition makes no reference to Turing machines, or algorithms, or any specific computational model. But like composition and primitive recursion, there is an operational, computational intuition behind unbounded search. When it comes to the computability of a partial function, arguments where the function is undefined correspond to inputs for which the computation does not halt. The procedure for computing $\mu x\ f(x, \vec{z})$ will amount to this: compute $f(0, \vec{z}), f(1, \vec{z}), f(2, \vec{z})$ until a value of 0 is returned. If any of the intermediate computations do not halt, however, neither does the computation of $\mu x\ f(x, \vec{z})$. explanation

If $R(x, \vec{z})$ is any relation, $\mu x\ R(x, \vec{z})$ is defined to be $\mu x\ (1 \dot{-} \chi_R(x, \vec{z}))$. In other words, $\mu x\ R(x, \vec{z})$ returns the least value of x such that $R(x, \vec{z})$ holds. So, if $f(x, \vec{z})$ is a total function, $\mu x\ f(x, \vec{z})$ is the same as $\mu x\ (f(x, \vec{z}) = 0)$. But note that our original definition is more general, since it allows for the possibility that $f(x, \vec{z})$ is not everywhere defined (whereas, in contrast, the characteristic function of a relation is always total).

Definition rec.26. The set of *partial recursive functions* is the smallest set of partial functions from the natural numbers to the natural numbers (of various arities) containing zero, successor, and projections, and closed under composition, primitive recursion, and unbounded search.

Of course, some of the partial recursive functions will happen to be total, i.e., defined for every argument.

Definition rec.27. The set of *recursive functions* is the set of partial recursive functions that are total. cmp:rec:par: defn:recursive-fn

A recursive function is sometimes called "total recursive" to emphasize that it is defined everywhere.

rec.16 The Normal Form Theorem

cmp:rec:nft: sec

Theorem rec.28 (Kleene's Normal Form Theorem)**.** *There is a primitive recursive relation $T(e, x, s)$ and a primitive recursive function $U(s)$, with the following property: if f is any partial recursive function, then for some e,* cmp:rec:nft: thm:kleene-nf

$$f(x) \simeq U(\mu s\, T(e, x, s))$$

for every x.

explanation

The proof of the normal form theorem is involved, but the basic idea is simple. Every partial recursive function has an *index* e, intuitively, a number coding its program or definition. If $f(x) \downarrow$, the computation can be recorded systematically and coded by some number s, and that s codes the computation of f on input x can be checked primitive recursively using only x and the definition e. This means that T is primitive recursive. Given the full record of the computation s, the "upshot" of s is the value of $f(x)$, and it can be obtained from s primitive recursively as well.

The normal form theorem shows that only a single unbounded search is required for the definition of any partial recursive function. We can use the numbers e as "names" of partial recursive functions, and write φ_e for the function f defined by the equation in the theorem. Note that any partial recursive function can have more than one index—in fact, every partial recursive function has infinitely many indices.

rec.17 The Halting Problem

cmp:rec:hlt: sec

The *halting problem* in general is the problem of deciding, given the specification e (e.g., program) of a computable function and a number n, whether the computation of the function on input n halts, i.e., produces a result. Famously, Alan Turing proved that this problem itself cannot be solved by a computable function, i.e., the function

$$h(e, n) = \begin{cases} 1 & \text{if computation } e \text{ halts on input } n \\ 0 & \text{otherwise,} \end{cases}$$

is not computable.

In the context of partial recursive functions, the role of the specification of a program may be played by the index e given in Kleene's normal form theorem. If f is a partial recursive function, any e for which the equation in the normal form theorem holds, is an index of f. Given a number e, the normal form theorem states that

$$\varphi_e(x) \simeq U(\mu s\, T(e, x, s))$$

is partial recursive, and for every partial recursive $f\colon \mathbb{N} \to \mathbb{N}$, there is an $e \in \mathbb{N}$ such that $\varphi_e(x) \simeq f(x)$ for all $x \in \mathbb{N}$. In fact, for each such f there is not just one, but infinitely many such e. The *halting function* h is defined by

$$h(e, x) = \begin{cases} 1 & \text{if } \varphi_e(x) \downarrow \\ 0 & \text{otherwise.} \end{cases}$$

Note that $h(e, x) = 0$ if $\varphi_e(x) \uparrow$, but also when e is not the index of a partial recursive function at all.

cmp:rec:hlt:thm:halting-problem

Theorem rec.29. *The halting function h is not partial recursive.*

Proof. If h were partial recursive, we could define

$$d(y) = \begin{cases} 1 & \text{if } h(y, y) = 0 \\ \mu x\ x \neq x & \text{otherwise.} \end{cases}$$

From this definition it follows that

1. $d(y) \downarrow$ iff $\varphi_y(y) \uparrow$ or y is not the index of a partial recursive function.
2. $d(y) \uparrow$ iff $\varphi_y(y) \downarrow$.

If h were partial recursive, then d would be partial recursive as well. Thus, by the Kleene normal form theorem, it has an index e_d. Consider the value of $h(e_d, e_d)$. There are two possible cases, 0 and 1.

1. If $h(e_d, e_d) = 1$ then $\varphi_{e_d}(e_d) \downarrow$. But $\varphi_{e_d} \simeq d$, and $d(e_d)$ is defined iff $h(e_d, e_d) = 0$. So $h(e_d, e_d) \neq 1$.
2. If $h(e_d, e_d) = 0$ then either e_d is not the index of a partial recursive function, or it is and $\varphi_{e_d}(e_d) \uparrow$. But again, $\varphi_{e_d} \simeq d$, and $d(e_d)$ is undefined iff $\varphi_{e_d}(e_d) \downarrow$.

The upshot is that e_d cannot, after all, be the index of a partial recursive function. But if h were partial recursive, d would be too, and so our definition of e_d as an index of it would be admissible. We must conclude that h cannot be partial recursive. □

rec.18 General Recursive Functions

There is another way to obtain a set of total functions. Say a total function $f(x, \vec{z})$ is *regular* if for every sequence of natural numbers $\vec{z}$, there is an x such that $f(x, \vec{z}) = 0$. In other words, the regular functions are exactly those functions to which one can apply unbounded search, and end up with a total function. One can, conservatively, restrict unbounded search to regular functions: cmp:rec:gen:sec

Definition rec.30. The set of *general recursive functions* is the smallest set of functions from the natural numbers to the natural numbers (of various arities) containing zero, successor, and projections, and closed under composition, primitive recursion, and unbounded search applied to *regular* functions. cmp:rec:gen:defn:general-recursive

Clearly every general recursive function is total. The difference between Definition rec.30 and Definition rec.27 is that in the latter one is allowed to use partial recursive functions along the way; the only requirement is that the function you end up with at the end is total. So the word "general," a historic relic, is a misnomer; on the surface, Definition rec.30 is *less* general than Definition rec.27. But, fortunately, the difference is illusory; though the definitions are different, the set of general recursive functions and the set of recursive functions are one and the same.

Photo Credits

Bibliography