Chapter udf

Recursive Functions

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 $\langle n_1, \dots, n_k \rangle$ 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

explanation Suppose we specify that a certain function l from $\mathbb{N}$ to $\mathbb{N}$ satisfies the following two clauses: cmp:rec:pre:sec

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

It is pretty clear that there is only one function, l, that meets these two criteria. This is an instance of a *definition by primitive recursion.* We can define even more fundamental functions like addition and multiplication by

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

and

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

Exponentiation can also be defined recursively, by

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

We can also compose functions to build more complex ones; for example,

$$\begin{aligned} k(x) &= x^x + (x+3) \cdot x \\ &= f(h(x,x), g(f(x,3),x)). \end{aligned}$$

Let $\mathrm{zero}(x)$ be the function that always returns 0, regardless of what x is, and let $\mathrm{succ}(x) = x+1$ be the successor function. The set of *primitive recursive functions* is the set of functions from $\mathbb{N}^n$ to $\mathbb{N}$ that you get if you start with zero and succ by iterating the two operations above, primitive recursion and composition. The idea is that primitive recursive functions are defined in a straightforward and explicit way, so that it is intuitively clear that each one can be computed using finite means.

Definition rec.1. If f is a k-place function and $g_0, \ldots, g_{k-1}$ are l-place functions on the natural numbers, the *composition* of f with $g_0, \ldots, g_{k-1}$ is the l-place function h defined by

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

Definition rec.2. If f is a k-place function and g is a $(k+2)$-place function, then the function defined by *primitive recursion from f and g* is the $(k+1)$-place function h defined by the equations

$$\begin{aligned} h(0, z_0, \ldots, z_{k-1}) &= f(z_0, \ldots, z_{k-1}) \\ h(x+1, z_0, \ldots, z_{k-1}) &= g(x, h(x, z_0, \ldots, z_{k-1}), z_0, \ldots, z_{k-1}) \end{aligned}$$

In addition to zero and succ, we will include among primitive recursive functions the projection functions,

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

for each natural number n and $i < n$. These are not terribly exciting in themselves: P_i^n is simply the k-place function that always returns its ith argument. But the allow us to define new functions by disregarding arguments or switching arguments, as we'll see later.

In the end, we have the following:

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_i^n is primitive recursive.
4. If f is a k-place primitive recursive function and $g_0, \ldots, g_{k-1}$ are l-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.

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

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

Our definition of composition may seem too rigid, since $g_0, \ldots, g_{k-1}$ are all required to have the same arity l. (Remember that the *arity* of a function is the number of arguments; an l-place function has arity l.) 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.

For another example, let us again consider addition. This is described recursively by the following two equations:

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

In other words, addition is the function add defined recursively by the equations

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

But even this is not a strict primitive recursive definition; we need to put it in the form

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

for some 1-place primitive recursive function f and some 3-place primitive recursive function g. We can take f to be P^1_0, and we can define g using composition,

$$g(y, w, x) = \mathrm{succ}(P^3_1(y, w, x)).$$

The function g, being the composition of basic primitive recursive functions, is primitive recursive; and hence so is h. (Note that, strictly speaking, we have defined the function $g(y, x)$ meeting the recursive specification of $x + y$; in other words, the variables are in a different order. Luckily, addition is commutative, so here the difference is not important; otherwise, we could define the function g' by

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

using composition.

explanation One advantage to having the precise 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 l-place functions g_0, …, g_{k-1}, and we have assigned notations H, G_0, …, G_{k-1} to the latter functions. Then, using a new symbol $\mathrm{Comp}_{k,l}$, we can denote the function f by $\mathrm{Comp}_{k,l}[H, G_0, \dots, G_{k-1}]$. For the functions defined by primitive recursion, we can use analogous notations of the form $\mathrm{Rec}_k[G, H]$, where k denotes that arity of the function being defined. With this setup, we can denote the addition function by

$$\mathrm{Rec}_2[P^1_0, \mathrm{Comp}_{1,3}[\mathrm{succ}, P^3_1]].$$

Having these notations sometimes proves useful.

Problem rec.1. Multiplication satisfies the recursive equations

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

Give the explicit precise definition of the function $\mathrm{mult}(x, y) = x \cdot y$, assuming that $\mathrm{add}(x, y) = x + y$ is already defined. Give the complete notation for mult.

rec.3 Primitive Recursive Functions are Computable

cmp:rec:cmp:sec

Suppose a function h is defined by primitive recursion

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

and suppose the functions f and g are computable. Then $h(0, \vec{z})$ can obviously be computed, since it is just $f(\vec{z})$ which we assume is computable. $h(1, \vec{z})$ can then also be computed, since $1 = 0 + 1$ and so $h(1, \vec{z})$ is just

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

We can go on in this way and compute

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

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

Thus, primitive recursion 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^n_i are computable, and composition and primitive recursion yield computable functions from computable functions, his means that every primitive recursive function is computable.

rec.4 Examples of Primitive Recursive Functions

Here are some examples of primitive recursive functions: cmp:rec:exa:sec

1. Constants: for each natural number n, the function that always returns n primitive recursive function, since it is equal to $\operatorname{succ}(\operatorname{succ}(\dots\operatorname{succ}(\operatorname{zero}(x))))$.
2. The identity function: $\operatorname{id}(x) = x$, i.e. P^1_0
3. Addition, $x + y$
4. Multiplication, $x \cdot y$
5. Exponentiation, x^y (with 0^0 defined to be 1)
6. Factorial, $x! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot x$
7. The predecessor function, $\operatorname{pred}(x)$, defined by
$$\operatorname{pred}(0) = 0, \quad \operatorname{pred}(x+1) = x$$
8. Truncated subtraction, $x \dot{-} y$, defined by
$$x \dot{-} 0 = x, \quad x \dot{-} (y+1) = \operatorname{pred}(x \dot{-} y)$$
9. Maximum, $\max(x, y)$, defined by
$$\max(x, y) = x + (y \dot{-} x)$$
10. Minimum, $\min(x, y)$
11. Distance between x and y, $|x - y|$

explanation In our definitions, we'll often use constants n. This is ok because the constant function $\operatorname{const}_n(x)$ is primitive recursive (defined from zero and succ). So if, e.g., we want to define the function $f(x) = 2 \cdot x$ can obtain it by composition from $\operatorname{const}_n(x)$ and multiplication as $f(x) = \operatorname{const}_2(x) \cdot P^1_0(x)$. We'll make use of this trick from now on.

You'll also have noticed that the definition of pred does not, strictly speaking, fit into the pattern of definition by primitive recursion, since that pattern requires an extra argument. It is also odd in that it does not actually $\operatorname{pred}(x)$ in the definition of $\operatorname{pred}(x+1)$. But we can define $\operatorname{pred}'(x, y)$ by

$$\begin{aligned} \operatorname{pred}'(0, y) &= \operatorname{zero}(y) = 0 \\ \operatorname{pred}'(x+1, y) &= P^3_0(x, \operatorname{pred}'(x, y), y) = x \end{aligned}$$

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

Problem rec.2. Show that

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

is primitive recursive.

Problem rec.3. Show that $d(x,y) = \lfloor x/y \rfloor$ (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 definifion of d using primitive recursion and composition. You will have detour through an axiliary function—you cannot use recursion on the arguments x or y themselves.

The set of primitive recursive functions is further closed under the following two operations:

1. Finite sums: if $f(x,\vec{z})$ is primitive recursive, then so is the function
$$g(y,\vec{z}) = \sum_{x=0}^{y} f(x,\vec{z}).$$
2. Finite products: if $f(x,\vec{z})$ is primitive recursive, then so is the function
$$h(y,\vec{z}) = \prod_{x=0}^{y} f(x,\vec{z}).$$

For example, finite sums are defined recursively by the equations

$$g(0,\vec{z}) = f(0,\vec{z}), \quad g(y+1,\vec{z}) = g(y,\vec{z}) + f(y+1,\vec{z}).$$

We can also define boolean operations, where 1 stands for true, and 0 for false:

1. Negation, $\mathrm{not}(x) = 1 \dot{-} x$
2. Conjunction, $\mathrm{and}(x,y) = x \cdot y$

Other classical boolean operations like $\mathrm{or}(x,y)$ and $\mathrm{ifthen}(x,y)$ can be defined from these in the usual way.

rec.5 Primitive Recursive Relations

cmp:rec:prr:sec

Definition rec.4. 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 $\mathrm{IsZero}(x)$, which holds if and only if $x = 0$, corresponds to the function χ_{IsZero}, defined using primitive recursion by

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

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

Furthermore, the set of primitive recursive relations is closed under boolean operations:

1. Negation, $\neg P$
2. Conjunction, $P \wedge Q$
3. Disjunction, $P \vee Q$
4. If …then, $P \rightarrow Q$

are all primitive recursive, if P and Q are. For suppose $\chi_P(\vec{z})$ an $\chi_Q(\vec{z})$ are primitive recursive. Then the relation $R(\vec{z})$ that holds iff both $P(\vec{z})$ and $Q(\vec{z})$ hold has the characteristic function $\chi_R(\vec{z}) = \mathrm{and}(\chi_P(\vec{z}), \chi_Q(\vec{z}))$.

One can also define relations using bounded quantification:

1. Bounded universal quantification: if $R(x, \vec{z})$ is a primitive recursive relation, then so is the relation

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

   which holds if and only if $R(x, \vec{z})$ holds for every x less than y.

2. Bounded existential quantification: if $R(x, \vec{z})$ is a primitive recursive relation, then so is

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

By convention, we take $(\forall x < 0)\ R(x, \vec{z})$ to be true (for the trivial reason that there *are* no x less than 0) and $(\exists x < 0)\ R(x, \vec{z})$ to be false. A universal quantifier functions just like a finite product; it can also be defined directly by

$$g(0, \vec{z}) = 1, \quad g(y+1, \vec{z}) = \mathrm{and}(g(y, \vec{z}), \chi_R(y, \vec{z})).$$

Bounded existential quantification can similarly be defined using or. Alternatively, it can be defined from bounded universal quantification, using the equivalence, $(\exists x < y)\ \varphi(x) \leftrightarrow \neg(\forall x < y)\ \neg\varphi(x)$. 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:

1. The conditional function, $\text{cond}(x, y, z)$, defined by

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

This is defined recursively by

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

One can use this to justify:

1. Definition by cases: if $g_0(\vec{x}), \dots, g_m(\vec{x})$ are functions, and $R_1(\vec{x}), \dots, R_{m-1}(\vec{x})$ are 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.

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

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

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

rec.6 Bounded Minimization

cmp:rec:bmi:sec 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 indipendently 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 y and z to the least $x < y$ such that $R(x, z)$. It, too, can be computed, by testing whether $R(0, z)$, $R(1, z)$, ..., $R(y - 1, z)$. But why is it primitive recursive? explanation

Proposition rec.5. *If $R(x, \vec{z})$ is primitive recursive, so is the function $m_R(y, \vec{z})$ which returns the least x less than y such that $R(x, \vec{z})$ holds, if there is one, and* 0 *otherwise. We will write the function m_R as*

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

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

In case the bound is $y + 1$ we have three cases: (a) There is an $x < y$ such that $R(x, \vec{z})$, in which case $m_R(y + 1, \vec{z}) = m_R(y, \vec{z})$. (b) There is no such x but $R(y, \vec{z})$ holds, then $m_R(y + 1, \vec{z}) = y$. (c) There is no $x < y + 1$ such that $R(x, \vec{z})$, then $m_R(y + 1, \vec{z}) = 0$. So,

$$m_R(0, \vec{z}) = 0$$

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

□

explanation

The choice of "0 otherwise" is somewhat arbitrary. It is in fact even easier to recursively define the function m'_R which returns the least x less than y such that $R(x, \vec{z})$ holds, and $y + 1$ otherwise. When we use min, however, we will always know that the least x such that $R(x, \vec{z})$ exists and is less than y. Thus, in practice, we will not have to worry about the possibility that if $(\min x < y)\, R(x, \vec{z}) = 0$ we do not know if that value indicates that $R(0, \vec{z})$ or that for no $x < y$, $R(x, \vec{z})$. As with bounded quantification, $(\min x \le y) \ldots$ can be understood as $(\min x < y + 1) \ldots$.

Problem rec.4. Suppose $R(x, \vec{z})$ is primitive recursive. Define the function $m'_R(y, \vec{z})$ which returns the least x less than y such that $R(x, \vec{z})$ holds, if there is one, and $y + 1$ otherwise, by primitive recursion from χ_R.

rec.7 Primes

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 $x \mid y$. $x \mid y$ holds if division of x by y 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 $\le y$. So, we have that $x \mid y$ iff for some $z \le y$, $x \cdot z = y$. We can define the relation $x \mid y$ by bounded existential quantification from $=$ and multiplication by cmp:rec:pri:sec

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

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

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

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:

$$\begin{aligned} p(0) &= 2 \\ p(x+1) &= \mathrm{nextPrime}(p(x)) \end{aligned}$$

Since $\mathrm{nextPrime}(x)$ 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 $x! + 1$.

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

This shows, that $\mathrm{nextPrime}(x)$ 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 $p = p_0 \cdot p_1 \cdot \dots \cdot p_n$ 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.5. Define integer division $d(x, y)$ using bounded minimization.

rec.8 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 an adequate means of handling *sequences*. We will identify finite sequences of natural numbers with natural numbers in the following way: the sequence $\langle a_0, a_1, a_2, \dots, a_k \rangle$ corresponds to the number

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

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 $\emptyset$ denote 0.

Let us define the following functions:

1. len(s), which returns the length of the sequence s: Let $R(i, s)$ be the relation defined by

$$R(i, s) \text{ iff } p_i \mid s \wedge (\forall j < s)\,(j > i \rightarrow p_j \not\mid s)$$

R is primitive recursive. Now let

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

Note that we need to bound the search on i; clearly s provides an acceptable bound.

2. append(s, a), which returns the result of appending a to the sequence s:

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

3. element(s, i), which returns the ith element of s (where the initial element is called the 0th), or 0 if i is greater than or equal to the length of s:

$$\mathrm{element}(s, i) = \begin{cases} 0 & \text{if } i \geq \mathrm{len}(s) \\ \min j < s\ (p_i^{j+2} \not\mid s) - 1 & \text{otherwise} \end{cases}$$

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

$$\mathrm{append}(\mathrm{append}(\dots \mathrm{append}(\emptyset, s_0) \dots), s_k).$$

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

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

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

then the numeric code of the sequence s described above is at most sequenceBound(x, k).

Having such a bound on sequences gives us a way of defining new functions using bounded search. For example, suppose we want to define the function concat(s, t), which concatenates two sequences. One first option is to define a "helper" function hconcat(s, t, n) which concatenates the first n symbols of t to s. This function can be defined by primitive recursion, as follows:

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

Then we can define concat by

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

But using bounded search, we can be lazy. All we need to do is write down a primitive recursive *specification* of the object (number) we are looking for, and a bound on how far to look. The following works:

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

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

Problem rec.6. Show that there is a primitive recursive function $\mathrm{sconcat}(s)$ with the property that

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

rec.9 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:

$$\begin{aligned} f_0(0, \vec{z}) &= k_0(\vec{z}) \\ f_1(0, \vec{z}) &= k_1(\vec{z}) \\ f_0(x + 1, \vec{z}) &= h_0(x, f_0(x, \vec{z}), f_1(x, \vec{z}), \vec{z}) \\ f_1(x + 1, \vec{z}) &= h_1(x, f_0(x, \vec{z}), f_1(x, \vec{z}), \vec{z}) \end{aligned}$$

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

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

The following schema captures this idea more succinctly:

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

with the understanding that the second argument to h is just the empty sequence when x is 0. In either formulation, the idea is that in computing the "successor step," the function f 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:

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

In other words, the value of f at x can be computed in terms of the value of f 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:

$$\begin{aligned} f(0, \vec{z}) &= g(\vec{z}) \\ f(x + 1, \vec{z}) &= h(x, f(x, k(\vec{z})), \vec{z}) \end{aligned}$$

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

Finally, notice that we can always extend our "universe" by defining additional objects in terms of the natural numbers, and defining primitive recursive functions that operate on them. For example, we can take an integer to be given by a pair $\langle m, n \rangle$ of natural numbers, which, intuitively, represents the integer $m - n$. In other words, we say

$$\text{Integer}(x) \Leftrightarrow \text{length}(x) = 2$$

and then we define the following:

1. $\text{iequal}(x, y)$
2. $\text{iplus}(x, y)$
3. $\text{iminus}(x, y)$
4. $\text{itimes}(x, y)$

Similarly, we can define a rational number to be a pair $\langle x, y \rangle$ of integers with $y \neq 0$, representing the value x/y. And we can define qequal, qplus, qminus, qtimes, qdivides, and so on.

rec.10 Non-Primitive Recursive Functions

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, $f_0, f_1, f_2, \ldots$ such that we can effectively compute the value of f_x on input y; in other words, the function $g(x, y)$, defined by cmp:rec:npr:sec

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

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 a 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 I am 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.11 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, …, g_k all are partial functions, then

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

is defined if and only if each g_i is defined at $\vec{x}$, and h is defined at $g_0(\vec{x})$, …, $g_k(\vec{x})$. 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 $f(0, \vec{z}), f(1, \vec{z}), \dots, f(x, \vec{z})$ 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.6. 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.

cmp:rec:par: defn:recursive-fn **Definition rec.7.** The set of *recursive functions* is the set of partial recursive functions that are total.

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

rec.12 The Normal Form Theorem

cmp:rec:nft:sec

Theorem rec.8 (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.13 The Halting Problem

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 cmp:rec:hlt:sec

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

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.9. *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.14 General Recursive Functions

cmp:rec:gen: sec

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:

Definition rec.10. 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.10 and Definition rec.7 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.10 is *less* general than Definition rec.7. 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