Part I

Computability

Chapter 1

Recursive Functions

1.1 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.

1.2 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 1.1. If f is a k-place function and $g_0, \dots, g_{k-1}$ are l-place functions on the natural numbers, the *composition* of f with $g_0, \dots, g_{k-1}$ is the l-place function h defined by

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

Definition 1.2. 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 1.3. 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, …, g_{k-1} are l-place primitive recursive functions, then the composition of f with g_0, …, 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, …, 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 1.1. Multiplication satisfies the recursive equations

$$0 \cdot y = y$$
$$(x+1) \cdot y = (x \cdot y) + x$$

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.

1.3 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.

1.4 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 $\mathrm{succ}(\mathrm{succ}(\dots \mathrm{succ}(\mathrm{zero}(x))))$.
2. The identity function: $\mathrm{id}(x) = x$, 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, $\mathrm{pred}(x)$, defined by
$$\mathrm{pred}(0) = 0, \quad \mathrm{pred}(x + 1) = x$$
8. Truncated subtraction, $x \dot{-} y$, defined by
$$x \dot{-} 0 = x, \quad x \dot{-} (y + 1) = \mathrm{pred}(x \dot{-} y)$$
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 $\mathrm{const}_n(x)$ 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 $\mathrm{const}_n(x)$ and multiplication as $f(x) = \mathrm{const}_2(x) \cdot P^1_0(x)$. 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 $\mathrm{pred}(x)$ in the definition of $\mathrm{pred}(x + 1)$. But we can define $\mathrm{pred}'(x, y)$ by

$$\mathrm{pred}'(0, y) = \mathrm{zero}(y) = 0$$
$$\mathrm{pred}'(x + 1, y) = P^3_0(x, \mathrm{pred}'(x, y), y) = x$$

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

Problem 1.2. Show that

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

is primitive recursive.

Problem 1.3. 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.

1.5 Primitive Recursive Relations

cmp:rec:prr:sec

Definition 1.4. 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, $\mathrm{cond}(x, y, z)$, 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:

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

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

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

1.6 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 1.5. *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 1.4. 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.

1.7 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

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

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:

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

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 $x! + 1$.

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

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 1.5. Define integer division $d(x, y)$ using bounded minimization.

1.8 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. $\text{len}(s)$, which returns the length of the sequence s: Let $R(i, s)$ be the relation defined by

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

R is primitive recursive. Now let

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

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

2. $\text{append}(s, a)$, which returns the result of appending a to the sequence s:

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

3. $\text{element}(s, i)$, 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:

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

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

$$\text{append}(\text{append}(\ldots \text{append}(\emptyset, s_0) \ldots), s_k).$$

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

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

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

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, suppose we want to define the function $\text{concat}(s, t)$, which concatenates two sequences. One first option is to define 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:

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

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:

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

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

Problem 1.6. Show that there is a primitive recursive function $\mathrm{sconcat}(s)$ with the property that

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

1.9 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:

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

The following schema captures this idea more succinctly:

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

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:

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

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

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

and then we define the following:

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

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.

1.10 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, … 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.

1.11 Partial Recursive Functions

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! cmp:rec:par:sec

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

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

is defined if and only if each g_i is defined at $\vec{x}$, and h is defined at $g_0(\vec{x})$, …, $g_k(\vec{x})$. 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 1.6. 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 1.7.** 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.

1.12 The Normal Form Theorem

cmp:rec:nft:sec

Theorem 1.8 (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.

1.13 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 1.9. *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. $\square$

1.14 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 1.10. 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 1.10 and Definition 1.7 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 1.10 is *less* general than Definition 1.7. 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.

Chapter 2

The Lambda Calculus

2.1 Introduction

cmp:lam:int:sec

The lambda calculus was originally designed by Alonzo Church in the early 1930s as a basis for constructive logic, and *not* as a model of the computable functions. But soon after the Turing computable functions, the recursive functions, and the general recursive functions were shown to be equivalent, lambda computability was added to the list. The fact that this initially came as a small surprise makes the characterization all the more interesting.

Lambda notation is a convenient way of referring to a function directly by a symbolic expression which defines it, instead of defining a name for it. Instead of saying "let f be the function defined by $f(x) = x + 3$," one can say, "let f be the function $\lambda x.\,(x + 3)$." In other words, $\lambda x.\,(x + 3)$ is just a *name* for the function that adds three to its argument. In this expression, x is a dummy variable, or a placeholder: the same function can just as well be denoted by $\lambda y.\,(y+3)$. The notation works even with other parameters around. For example, suppose $g(x, y)$ is a function of two variables, and k is a natural number. Then $\lambda x.\, g(x, k)$ is the function which maps any x to $g(x, k)$.

This way of defining a function from a symbolic expression is known as *lambda abstraction*. The flip side of lambda abstraction is *application*: assuming one has a function f (say, defined on the natural numbers), one can *apply* it to any value, like 2. In conventional notation, of course, we write $f(2)$ for the result.

What happens when you combine lambda abstraction with application? Then the resulting expression can be simplified, by "plugging" the applicand in for the abstracted variable. For example,

$$(\lambda x.\,(x + 3))(2)$$

can be simplified to $2 + 3$.

Up to this point, we have done nothing but introduce new notations for conventional notions. The lambda calculus, however, represents a more radical departure from the set-theoretic viewpoint. In this framework:

1. Everything denotes a function.
2. Functions can be defined using lambda abstraction.
3. Anything can be applied to anything else.

For example, if F is a term in the lambda calculus, $F(F)$ is always assumed to be meaningful. This liberal framework is known as the *untyped* lambda calculus, where "untyped" means "no restriction on what can be applied to what."

digression There is also a *typed* lambda calculus, which is an important variation on the untyped version. Although in many ways the typed lambda calculus is similar to the untyped one, it is much easier to reconcile with a classical set-theoretic framework, and has some very different properties.

Research on the lambda calculus has proved to be central in theoretical computer science, and in the design of programming languages. LISP, designed by John McCarthy in the 1950s, is an early example of a language that was influenced by these ideas.

2.2 The Syntax of the Lambda Calculus

One starts with a sequence of variables x, y, z, ... and some constant symbols a, b, c, ... The set of terms is defined inductively, as follows: cmp:lam:syn:sec

1. Each variable is a term.
2. Each constant is a term.
3. If M and N are terms, so is (MN).
4. If M is a term and x is a variable, then $(\lambda x.\, M)$ is a term.

The system without any constants at all is called the *pure* lambda calculus.

We will follow a few notational conventions:

1. When parentheses are left out, application takes place from left to right. For example, if M, N, P, and Q are terms, then $MNPQ$ abbreviates $(((MN)P)Q)$.
2. Again, when parentheses are left out, lambda abstraction is to be given the widest scope possible. From example, $\lambda x.\, MNP$ is read $\lambda x.\, (MNP)$.
3. A lambda can be used to abstract multiple variables. For example, $\lambda xyz.\, M$ is short for $\lambda x.\, \lambda y.\, \lambda z.\, M$.

For example,

$$\lambda xy.\, xxyx\lambda z.\, xz$$

abbreviates

$$\lambda x.\, \lambda y.\, ((((xx)y)x)\lambda z.\, (xz)).$$

You should memorize these conventions. They will drive you crazy at first, but you will get used to them, and after a while they will drive you less crazy than having to deal with a morass of parentheses.

Two terms that differ only in the names of the bound variables are called α-equivalent; for example, $\lambda x.\, x$ and $\lambda y.\, y$. It will be convenient to think of these as being the "same" term; in other words, when we say that M and N are the same, we also mean "up to renamings of the bound variables." Variables that are in the scope of a λ are called "bound", while others are called "free." There are no free variables in the previous example; but in

$$(\lambda z.\, yz)x$$

y and x are free, and z is bound.

2.3 Reduction of Lambda Terms

cmp:lam:red:sec What can one do with lambda terms? Simplify them. If M and N are any lambda terms and x is any variable, we can use $M[N/x]$ to denote the result of substituting N for x in M, after renaming any bound variables of M that would interfere with the free variables of N after the substitution. For example,

$$(\lambda w.\, xxw)[yyz/x] = \lambda w.\, (yyz)(yyz)w.$$

Alternative notations for substitution are $[N/x]M$, $M[N/x]$, and also $M[x/N]$. digression Beware!

Intuitively, $(\lambda x.\, M)N$ and $M[N/x]$ have the same meaning; the act of replacing the first term by the second is called β-conversion. More generally, if it is possible convert a term P to P' by β-conversion of some subterm, one says *P β-reduces to P' in one step.* If P can be converted to P' with any number of one-step reductions (possibly none), then P *β-reduces* to P'. A term that cannot be β-reduced any further is called *β-irreducible*, or β-normal. I will say "reduces" instead of "β-reduces," etc., when the context is clear.

Let us consider some examples.

1. We have

$$\begin{aligned}(\lambda x.\, xxy)\lambda z.\, z &\rhd_1 (\lambda z.\, z)(\lambda z.\, z)y\\ &\rhd_1 (\lambda z.\, z)y\\ &\rhd_1 y\end{aligned}$$

2. "Simplifying" a term can make it more complex:

$$\begin{aligned}(\lambda x.\, xxy)(\lambda x.\, xxy) &\rhd_1 (\lambda x.\, xxy)(\lambda x.\, xxy)y\\ &\rhd_1 (\lambda x.\, xxy)(\lambda x.\, xxy)yy\\ &\rhd_1 \dots\end{aligned}$$

3. It can also leave a term unchanged:

$$(\lambda x.\, xx)(\lambda x.\, xx) \triangleright_1 (\lambda x.\, xx)(\lambda x.\, xx)$$

4. Also, some terms can be reduced in more than one way; for example,

$$(\lambda x.\, (\lambda y.\, yx)z)v \triangleright_1 (\lambda y.\, yv)z$$

by contracting the outermost application; and

$$(\lambda x.\, (\lambda y.\, yx)z)v \triangleright_1 (\lambda x.\, zx)v$$

by contracting the innermost one. Note, in this case, however, that both terms further reduce to the same term, zv.

The final outcome in the last example is not a coincidence, but rather illustrates a deep and important property of the lambda calculus, known as the "Church-Rosser property."

2.4 The Church-Rosser Property

cmp:lam:cr:sec

Theorem 2.1. *Let M, N_1, and N_2 be terms, such that $M \triangleright N_1$ and $M \triangleright N_2$. Then there is a term P such that $N_1 \triangleright P$ and $N_2 \triangleright P$.* *cmp:lam:cr:thm:church-rosser*

Corollary 2.2. *Suppose M can be reduced to normal form. Then this normal form is unique.*

Proof. If $M \triangleright N_1$ and $M \triangleright N_2$, by the previous theorem there is a term P such that N_1 and N_2 both reduce to P. If N_1 and N_2 are both in normal form, this can only happen if $N_1 = P = N_2$. □

Finally, we will say that two terms M and N are *β-equivalent*, or just *equivalent*, if they reduce to a common term; in other words, if there is some P such that $M \triangleright P$ and $N \triangleright P$. This is written $M \equiv N$. Using Theorem 2.1, you can check that $\equiv$ is an equivalence relation, with the additional property that for every M and N, if $M \triangleright N$ or $N \triangleright M$, then $M \equiv N$. (In fact, one can show that $\equiv$ is the *smallest* equivalence relation having this property.)

2.5 Representability by Lambda Terms

How can the lambda calculus serve as a model of computation? At first, it is not even clear how to make sense of this statement. To talk about computability on the natural numbers, we need to find a suitable representation for such numbers. Here is one that works surprisingly well. cmp:lam:rep:sec

Definition 2.3. For each natural number n, define the *numeral* $\overline{n}$ to be the lambda term $\lambda x.\, \lambda y.\, (x(x(x(\dots x(y)))))$, where there are n x's in all.

The terms $\overline{n}$ are "iterators": on input f, $\overline{n}$ returns the function mapping y to $f^n(y)$. Note that each numeral is normal. We can now say what it means for a lambda term to "compute" a function on the natural numbers.

Definition 2.4. Let $f(x_0, \dots, x_{n-1})$ be an n-ary partial function from $\mathbb{N}$ to $\mathbb{N}$. We say a lambda term X *represents* f if for every sequence of natural numbers $m_0, \dots, m_{n-1}$,

$$X\overline{m_0}\overline{m_1}\dots\overline{m_{n-1}} \triangleright \overline{f(m_0, m_1, \dots, m_{n-1})}$$

if $f(m_0, \dots, m_{n-1})$ is defined, and $X\overline{m_0}\overline{m_1}\dots\overline{m_{n-1}}$ has no normal form otherwise.

cmp:lam:rep: thm:lambda-rep

Theorem 2.5. *A function f is a partial computable function if and only if it is represented by a lambda term.*

This theorem is somewhat striking. As a model of computation, the lambda calculus is a rather simple calculus; the only operations are lambda abstraction and application! From these meager resources, however, it is possible to implement any computational procedure. explanation

2.6 Lambda Representable Functions are Computable

cmp:lam:cmp: sec

cmp:lam:cmp: thm:lambda-computable

Theorem 2.6. *If a partial function f is represented by a lambda term, it is computable.*

Proof. Suppose a function f, is represented by a lambda term X. Let us describe an informal procedure to compute f. On input $m_0, \dots, m_{n-1}$, write down the term $X\overline{m}_0 \dots \overline{m}_{n-1}$. Build a tree, first writing down all the one-step reductions of the original term; below that, write all the one-step reductions of those (i.e., the two-step reductions of the original term); and keep going. If you ever reach a numeral, return that as the answer; otherwise, the function is undefined.

An appeal to Church's thesis tells us that this function is computable. A better way to prove the theorem would be to give a recursive description of this search procedure. For example, one could define a sequence primitive recursive functions and relations, "IsASubterm," "Substitute," "ReducesToInOneStep," "ReductionSequence," "Numeral," etc. The partial recursive procedure for computing $f(m_0, \dots, m_{n-1})$ is then to search for a sequence of one-step reductions starting with $X\overline{m_0}\dots\overline{m_{n-1}}$ and ending with a numeral, and return the number corresponding to that numeral. The details are long and tedious but otherwise routine. □

2.7 Computable Functions are Lambda Representable

cmp:lam:lrp:sec

Theorem 2.7. *Every computable partial function if representable by a lambda term.* cmp:lam:lrp:thm:computable-lambda

Proof. Wwe need to show that every partial computable function f is represented by a lambda term $\overline{f}$. By Kleene's normal form theorem, it suffices to show that every primitive recursive function is represented by a lambda term, and then that the functions so represented are closed under suitable compositions and unbounded search. To show that every primitive recursive function is represented by a lambda term, it suffices to show that the initial functions are represented, and that the partial functions that are represented by lambda terms are closed under composition, primitive recursion, and unbounded search. □

We will use a more conventional notation to make the rest of the proof more readable. For example, we will write $M(x, y, z)$ instead of $Mxyz$. While this is suggestive, you should remember that terms in the untyped lambda calculus do not have associated arities; so, for the same term M, it makes just as much sense to write $M(x, y)$ and $M(x, y, z, w)$. But using this notation indicates that we are treating M as a function of three variables, and helps make the intentions behind the definitions clearer. In a similar way, we will say "define M by $M(x, y, z) = \dots$" instead of "define M by $M = \lambda x. \lambda y. \lambda z. \dots$."

2.8 The Basic Primitive Recursive Functions are Lambda Representable

cmp:lam:bas:sec

Lemma 2.8. *The functions* 0, S, *and* P^n_i *are lambda representable.*

Proof. Zero, $\overline{0}$, is just $\lambda x. \lambda y. y$.

The successor function $\overline{S}$, is defined by $\overline{S}(u) = \lambda x. \lambda y. x(uxy)$. You should think about why this works; for each numeral $\overline{n}$, thought of as an iterator, and each function f, $S(\overline{n}, f)$ is a function that, on input y, applies f n times starting with y, and then applies it once more.

There is nothing to say about projections: $\overline{P^n_i}(x_0, \dots, x_{n-1}) = x_i$. In other words, by our conventions, $\overline{P^n_i}$ is the lambda term $\lambda x_0. \dots \lambda x_{n-1}. x_i$. □

2.9 Lambda Representable Functions Closed under Composition

cmp:lam:com:sec

Lemma 2.9. *The lambda representable functions are closed under composition.*

Proof. Suppose f is defined by composition from h, g_0, …, g_{k-1}. Assuming h, g_0, …, g_{k-1} are represented by $\overline{h}$, $\overline{g_0}$, …, $\overline{g_{k-1}}$, respectively, we need to find a term $\overline{f}$ representing f. But we can simply define $\overline{f}$ by

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

In other words, the language of the lambda calculus is well suited to represent composition. □

2.10 Lambda Representable Functions Closed under Primitive Recursion

cmp:lam:pr:sec When it comes to primitive recursion, we finally need to do some work. We will have to proceed in stages. As before, on the assumption that we already have terms $\overline{g}$ and $\overline{h}$ representing functions g and h, respectively, we want a term $\overline{f}$ representing the function f defined by

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

So, in general, given lambda terms G' and H', it suffices to find a term F such that

$$\begin{aligned} F(\overline{0}, \vec{z}) &\equiv G'(\vec{z}) \\ F(\overline{n+1}, \vec{z}) &\equiv H'(\overline{n}, F(\overline{n}, \vec{z}), \vec{z}) \end{aligned}$$

for every natural number n; the fact that G' and H' represent g and h means that whenever we plug in numerals $\vec{m}$ for $\vec{z}$, $F(\overline{n+1}, \vec{m})$ will normalize to the right answer.

But for this, it suffices to find a term F satisfying

$$\begin{aligned} F(\overline{0}) &\equiv G \\ F(\overline{n+1}) &\equiv H(\overline{n}, F(\overline{n})) \end{aligned}$$

for every natural number n, where

$$\begin{aligned} G &= \lambda \vec{z}.\, G'(\vec{z}) \text{ and} \\ H(u, v) &= \lambda \vec{z}.\, H'(u, v(u, \vec{z}), \vec{z}). \end{aligned}$$

In other words, with lambda trickery, we can avoid having to worry about the extra parameters $\vec{z}$—they just get absorbed in the lambda notation.

Before we define the term F, we need a mechanism for handling ordered pairs. This is provided by the next lemma.

Lemma 2.10. *There is a lambda term D such that for each pair of lambda terms M and N, $D(M, N)(\overline{0}) \triangleright M$ and $D(M, N)(\overline{1}) \triangleright N$.*

Proof. First, define the lambda term K by

$$K(y) = \lambda x.\, y.$$

In other words, K is the term $\lambda y.\, \lambda x.\, y$. Looking at it differently, for every M, $K(M)$ is a constant function that returns M on any input.

Now define $D(x, y, z)$ by $D(x, y, z) = z(K(y))x$. Then we have

$$\begin{aligned} D(M, N, \overline{0}) &\triangleright \overline{0}(K(N))M \triangleright M \text{ and} \\ D(M, N, \overline{1}) &\triangleright \overline{1}(K(N))M \triangleright K(N)M \triangleright N, \end{aligned}$$

as required. □

The idea is that $D(M, N)$ represents the pair $\langle M, N\rangle$, and if P is assumed to represent such a pair, $P(\overline{0})$ and $P(\overline{1})$ represent the left and right projections, $(P)_0$ and $(P)_1$. We will use the latter notations.

Lemma 2.11. *The lambda representable functions are closed under primitive recursion.*

Proof. We need to show that given any terms, G and H, we can find a term F such that

$$\begin{aligned} F(\overline{0}) &\equiv G \\ F(\overline{n+1}) &\equiv H(\overline{n}, F(\overline{n})) \end{aligned}$$

for every natural number n. The idea is roughly to compute sequences of *pairs*

$$\langle \overline{0}, F(\overline{0})\rangle, \langle \overline{1}, F(\overline{1})\rangle, \ldots,$$

using numerals as iterators. Notice that the first pair is just $\langle \overline{0}, G\rangle$. Given a pair $\langle \overline{n}, F(\overline{n})\rangle$, the next pair, $\langle \overline{n+1}, F(\overline{n+1})\rangle$ is supposed to be equivalent to $\langle \overline{n+1}, H(\overline{n}, F(\overline{n}))\rangle$. We will design a lambda term T that makes this one-step transition.

The details are as follows. Define $T(u)$ by

$$T(u) = \langle S((u)_0), H((u)_0, (u)_1)\rangle.$$

Now it is easy to verify that for any number n,

$$T(\langle \overline{n}, M\rangle) \triangleright \langle \overline{n+1}, H(\overline{n}, M)\rangle.$$

As suggested above, given G and H, define $F(u)$ by

$$F(u) = (u(T, \langle \overline{0}, G\rangle))_1.$$

In other words, on input $\overline{n}$, F iterates T n times on $\langle \overline{0}, G\rangle$, and then returns the second component. To start with, we have

1. $\overline{0}(T, \langle \overline{0}, G\rangle) \equiv \langle \overline{0}, G\rangle$

2. $F(\overline{0}) \equiv G$

By induction on n, we can show that for each natural number one has the following:

1. $\overline{n+1}(T, \langle \overline{0}, G \rangle) \equiv \langle \overline{n+1}, F(\overline{n+1}) \rangle$
2. $F(\overline{n+1}) \equiv H(\overline{n}, F(\overline{n}))$

For the second clause, we have

$$\begin{aligned} F(\overline{n+1}) &\triangleright (\overline{n+1}(T, \langle \overline{0}, G \rangle))_1 \\ &\equiv (T(\overline{n}(T, \langle \overline{0}, G \rangle)))_1 \\ &\equiv (T(\langle \overline{n}, F(\overline{n}) \rangle))_1 \\ &\equiv (\langle \overline{n+1}, H(\overline{n}, F(\overline{n})) \rangle)_1 \\ &\equiv H(\overline{n}, F(\overline{n})). \end{aligned}$$

Here we have used the induction hypothesis on the second-to-last line. For the first clause, we have

$$\begin{aligned} \overline{n+1}(T, \langle \overline{0}, G \rangle) &\equiv T(\overline{n}(T, \langle \overline{0}, G \rangle)) \\ &\equiv T(\langle \overline{n}, F(\overline{n}) \rangle) \\ &\equiv \langle \overline{n+1}, H(\overline{n}, F(\overline{n})) \rangle \\ &\equiv \langle \overline{n+1}, F(\overline{n+1}) \rangle. \end{aligned}$$

Here we have used the second clause in the last line. So we have shown $F(\overline{0}) \equiv G$ and, for every n, $F(\overline{n+1}) \equiv H(\overline{n}, F(\overline{n}))$, which is exactly what we needed. □

2.11 Fixed-Point Combinators

cmp:lam:fix:sec Suppose you have a lambda term g, and you want another term k with the property that k is β-equivalent to gk. Define terms

$$\operatorname{diag}(x) = xx$$

and

$$l(x) = g(\operatorname{diag}(x))$$

using our notational conventions; in other words, l is the term $\lambda x.\, g(xx)$. Let k be the term ll. Then we have

$$\begin{aligned} k &= (\lambda x.\, g(xx))(\lambda x.\, g(xx)) \\ &\triangleright g((\lambda x.\, g(xx))(\lambda x.\, g(xx))) \\ &= gk. \end{aligned}$$

If one takes

$$Y = \lambda g.\, ((\lambda x.\, g(xx))(\lambda x.\, g(xx)))$$

then Yg and $g(Yg)$ reduce to a common term; so $Yg \equiv_\beta g(Yg)$. This is known as "Curry's combinator." If instead one takes

$$Y = (\lambda xg.\, g(xxg))(\lambda xg.\, g(xxg))$$

then in fact Yg reduces to $g(Yg)$, which is a stronger statement. This latter version of Y is known as "Turing's combinator."

2.12 Lambda Representable Functions Closed under Minimization

cmp:lam:min:sec

Lemma 2.12. *Suppose $f(x, y)$ is primitive recursive. Let g be defined by*

$$g(x) \simeq \mu y\; f(x, y).$$

Then g is represented by a lambda term.

Proof. The idea is roughly as follows. Given x, we will use the fixed-point lambda term Y to define a function $h_x(n)$ which searches for a y starting at n; then $g(x)$ is just $h_x(0)$. The function h_x can be expressed as the solution of a fixed-point equation:

$$h_x(n) \simeq \begin{cases} n & \text{if } f(x, n) = 0 \\ h_x(n+1) & \text{otherwise.} \end{cases}$$

Here are the details. Since f is primitive recursive, it is represented by some term F. Remember that we also have a lambda term D, such that $D(M, N, \bar{0}) \triangleright M$ and $D(M, N, \bar{1}) \triangleright N$. Fixing x for the moment, to represent h_x we want to find a term H (depending on x) satisfying

$$H(\overline{n}) \equiv D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})).$$

We can do this using the fixed-point term Y. First, let U be the term

$$\lambda h.\, \lambda z.\, D(z, (h(Sz)), F(x, z)),$$

and then let H be the term YU. Notice that the only free variable in H is x. Let us show that H satisfies the equation above.

By the definition of Y, we have

$$H = YU \equiv U(YU) = U(H).$$

In particular, for each natural number n, we have

$$\begin{aligned} H(\overline{n}) &\equiv U(H, \overline{n}) \\ &\rhd D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})), \end{aligned}$$

as required. Notice that if you substitute a numeral $\overline{m}$ for x in the last line, the expression reduces to $\overline{n}$ if $F(\overline{m}, \overline{n})$ reduces to $\overline{0}$, and it reduces to $H(S(\overline{n}))$ if $F(\overline{m}, \overline{n})$ reduces to any other numeral.

To finish off the proof, let G be $\lambda x.\, H(\overline{0})$. Then G represents g; in other words, for every m, $G(\overline{m})$ reduces to reduces to $\overline{g(m)}$, if $g(m)$ is defined, and has no normal form otherwise. □

Chapter 3

Computability Theory

3.1 Introduction

cmp:thy:int:sec

The branch of logic known as *Computability Theory* deals with issues having to do with the computability, or relative computability, of functions and sets. It is a evidence of Kleene's influence that the subject used to be known as *Recursion Theory*, and today, both names are commonly used.

Let us call a function $f\colon \mathbb{N} \nrightarrow \mathbb{N}$ *partial computable* if it can be computed in some model of computation. If f is total we will simply say that f is *computable*. A relation R with computable characteristic function χ_R is also called computable. 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.

One can explore the subject without having to refer to a specific model of computation. To do this, one shows that there is a universal partial computable function, $\mathrm{Un}(k, x)$. This allows us to enumerate the partial computable functions. We will adopt the notation φ_k to denote the k-th unary partial computable function, defined by $\varphi_k(x) \simeq \mathrm{Un}(k, x)$. (Kleene used $\{k\}$ for this purpose, but this notation has not been used as much recently.) Slightly more generally, we can uniformly enumerate the partial computable functions of arbitrary arities, and we will use φ^n_k to denote the k-th n-ary partial recursive function.

Recall that if $f(\vec{x}, y)$ is a total or partial function, then $\mu y\ f(\vec{x}, y)$ is the function of $\vec{x}$ that returns the least y such that $f(\vec{x}, y) = 0$, assuming that all of $f(\vec{x}, 0), \dots, f(\vec{x}, y-1)$ are defined; if there is no such y, $\mu y\ f(\vec{x}, y)$ is undefined. If $R(\vec{x}, y)$ is a relation, $\mu y\ R(\vec{x}, y)$ is defined to be the least y such that $R(\vec{x}, y)$ is true; in other words, the least y such that *one minus* the characteristic function of R is equal to zero at $\vec{x}, y$.

To show that a function is computable, there are two ways one can proceed:

1. Rigorously: describe a Turing machine or partial recursive function ex-

plicitly, and show that it computes the function you have in mind;

2. Informally: describe an algorithm that computes it, and appeal to Church's thesis.

There is no fine line between the two; a detailed description of an algorithm should provide enough information so that it is relatively clear how one could, in principle, design the right Turing machine or sequence of partial recursive definitions. Fully rigorous definitions are unlikely to be informative, and we will try to find a happy medium between these two approaches; in short, we will try to find intuitive yet rigorous proofs that the precise definitions could be obtained.

3.2 Coding Computations

In every model of computation, it is possible to do the following: cmp:thy:cod:sec

1. Describe the *definitions* of computable functions in a systematic way. For instance, you can think of Turing machine specifications, recursive definitions, or programs in a programming language as providing these definitions.

2. Describe the complete record of the computation of a function given by some definition for a given input. For instance, a Turing machine computation can be described by the sequence of configurations (state of the machine, contents of the tape) for each step of computation.

3. Test whether a putative record of a computation is in fact the record of how a computable function with a given definition would be computed for a given input.

4. Extract from such a description of the complete record of a computation the value of the function for a given input. For instance, the contents of the tape in the very last step of a halting Turing machine computation is the value.

Using coding, it is possible to assign to each description of a computable function a numerical *index* in such a way that the instructions can be recovered from the index in a computable way. Similarly, the complete record of a computation can be coded by a single number as well. The resulting arithmetical relation "s codes the record of computation of the function with index e for input x" and the function "output of computation sequence with code s" are then computable; in fact, they are primitive recursive.

This fundamental fact is very powerful, and allows us to prove a number of striking and important results about computability, independently of the model of computation chosen.

3.3 The Normal Form Theorem

cmp:thy:nfm:sec

cmp:thy:nfm:thm:normal-form **Theorem 3.1** (Kleene's Normal Form Theorem)**.** *There are a primitive recursive relation $T(k, x, s)$ and a primitive recursive function $U(s)$, with the following property: if f is any partial computable function, then for some k,*

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

for every x.

Proof Sketch. For any model of computation one can rigorously define a description of the computable function f and code such description using a natural number k. One can also rigorously define a notion of "computation sequence" which records the process of computing the function with index k for input x. These computation sequences can likewise be coded as numbers s. This can be done in such a way that (a) it is decidable whether a number s codes the computation sequence of the function with index k on input x and (b) what the end result of the computation sequence coded by s is. In fact, the relation in (a) and the function in (b) are primitive recursive. □

In order to give a rigorous proof of the Normal Form Theorem, we would have to fix a model of computation and carry out the coding of descriptions of computable functions and of computation sequences in detail, and verify that the relation T and function U are primitive recursive. For most applications, it suffices that T and U are computable and that U is total. explanation

It is probably best to remember the proof of the normal form theorem in slogan form: $\mu s\, T(k, x, s)$ searches for a computation sequence of the function with index k on input x, and U returns the output of the computation sequence if one can be found.

T and U can be used to define the enumeration φ_0, φ_1, φ_2, ... From now on, we will assume that we have fixed a suitable choice of T and U, and take the equation

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

to be the *definition* of φ_e.

Here is another useful fact:

Theorem 3.2. *Every partial computable function has infinitely many indices.*

Again, this is intuitively clear. Given any (description of) a computable function, one can come up with a different description which computes the same function (input-output pair) but does so, e.g., by first doing something that has no effect on the computation (say, test if $0 = 0$, or count to 5, etc.). The index of the altered description will always be different from the original index. Both are indices of the same function, just computed slightly differently.

3.4 The s-m-n Theorem

explanation The next theorem is known as the "s-m-n theorem," for a reason that will be clear in a moment. The hard part is understanding just what the theorem says; once you understand the statement, it will seem fairly obvious. cmp:thy:smn:sec

Theorem 3.3. *For each pair of natural numbers n and m, there is a primitive recursive function s^m_n such that for every sequence x, a_0, ..., a_{m-1}, y_0 ,..., y_{n-1}, we have* cmp:thy:smn:thm:s-m-n

$$\varphi^n_{s^m_n(x,a_0,\dots,a_{m-1})}(y_0,\dots,y_{n-1}) \simeq \varphi^{m+n}_x(a_0,\dots,a_{m-1},y_0,\dots,y_{n-1}).$$

explanation It is helpful to think of s^m_n as acting on *programs.* That is, s^m_n takes a program, x, for an $(m+n)$-ary function, as well as fixed inputs a_0, ..., a_{m-1}; and it returns a program, $s^m_n(x,a_0,\dots,a_{m-1})$, for the n-ary function of the remaining arguments. It you think of x as the description of a Turing machine, then $s^m_n(x,a_0,\dots,a_{m-1})$ is the Turing machine that, on input y_0, ..., y_{n-1}, prepends a_0, ..., a_{m-1} to the input string, and runs x. Each s^m_n is then just a primitive recursive function that finds a code for the appropriate Turing machine.

3.5 The Universal Partial Computable Function

cmp:thy:uni:sec

Theorem 3.4. *There is a universal partial computable function* $\mathrm{Un}(k,x)$. *In other words, there is a function* $\mathrm{Un}(k,x)$ *such that:* cmp:thy:uni:thm:univ-comp

1. $\mathrm{Un}(k,x)$ *is partial computable.*

2. *If $f(x)$ is any partial computable function, then there is a natural number k such that $f(x) \simeq \mathrm{Un}(k,x)$ for every x.*

Proof. Let $\mathrm{Un}(k,x) \simeq U(\mu s\, T(k,x,s))$ in Kleene's normal form theorem. □

explanation This is just a precise way of saying that we have an effective enumeration of the partial computable functions; the idea is that if we write f_k for the function defined by $f_k(x) = \mathrm{Un}(k,x)$, then the sequence f_0, f_1, f_2, ... includes all the partial computable functions, with the property that $f_k(x)$ can be computed "uniformly" in k and x. For simplicity, we am using a binary function that is universal for unary functions, but by coding sequences of numbers we can easily generalize this to more arguments. For example, note that if $f(x,y,z)$ is a 3-place partial recursive function, then the function $g(x) \simeq f((x)_0,(x)_1,(x)_2)$ is a unary recursive function.

3.6 No Universal Computable Function

cmp:thy:nou:sec

cmp:thy:nou:thm:no-univ **Theorem 3.5.** *There is no universal computable function. In other words, the universal function* $\mathrm{Un}'(k,x) = \varphi_k(x)$ *is not computable.*

Proof. This theorem says that there is no *total* computable function that is universal for the total computable functions. The proof is a simple diagonalization: if $\mathrm{Un}'(k,x)$ were total and computable, then

$$d(x) = \mathrm{Un}'(x,x) + 1$$

would also be total and computable. However, for every k, $d(k)$ is not equal to $\mathrm{Un}'(k,k)$. □

Theorem Theorem 3.4 above shows that we can get around this diagonalization argument, but only at the expense of allowing partial functions. It is worth trying to understand what goes wrong with the diagonalization argument, when we try to apply it in the partial case. In particular, the function $h(x) = \mathrm{Un}(x,x) + 1$ *is* partial recursive. Suppose h is the k-th function in the enumeration; what can we say about $h(k)$? explanation

3.7 The Halting Problem

cmp:thy:hlt:sec

Since, in our construction, $\mathrm{Un}(k,x)$ is defined if and only if the computation of the function coded by k produces a value for input x, it is natural to ask if we can decide whether this is the case. And in fact, it is not. For the Turing machine model of computation, this means that whether a given Turing machine halts on a given input is computationally undecidable. The following theorem is therefore known as the "undecidability of the halting problem." I will provide two proofs below. The first continues the thread of our previous discussion, while the second is more direct.

cmp:thy:hlt:thm:halting-problem **Theorem 3.6.** *Let*

$$h(k,x) = \begin{cases} 1 & \text{if } \mathrm{Un}(k,x) \text{ is defined} \\ 0 & \text{otherwise.} \end{cases}$$

Then h is not computable.

Proof. If h were computable, we would have a universal computable function, as follows. Suppose h is computable, and define

$$\mathrm{Un}'(k,x) = \begin{cases} fnUn(k,x) & \text{if } h(k,x) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

But now $\mathrm{Un}'(k,x)$ is a total function, and is computable if h is. For instance, we could define g using primitive recursion, by

$$g(0,k,x) \simeq 0$$
$$g(y+1,k,x) \simeq \mathrm{Un}(k,x);$$

then

$$\mathrm{Un}'(k,x) \simeq g(h(k,x),k,x).$$

And since $\mathrm{Un}'(k,x)$ agrees with $\mathrm{Un}(k,x)$ wherever the latter is defined, Un' is universal for those partial computable functions that happen to be total. But this contradicts Theorem 3.5. □

Proof. Suppose $h(k,x)$ were computable. Define the function g by

$$g(x) = \begin{cases} 0 & \text{if } h(x,x) = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function g is partial computable; for example, one can define it as $\mu y\, h(x,x) = 0$. So, for some k, $g(x) \simeq \mathrm{Un}(k,x)$ for every x. Is g defined at k? If it is, then, by the definition of g, $h(k,k) = 0$. By the definition of f, this means that $\mathrm{Un}(k,k)$ is undefined; but by our assumption that $g(k) \simeq \mathrm{Un}(k,x)$ for every x, this means that $g(k)$ is undefined, a contradiction. On the other hand, if $g(k)$ is undefined, then $h(k,k) \neq 0$, and so $h(k,k) = 1$. But this means that $\mathrm{Un}(k,k)$ is defined, i.e., that $g(k)$ is defined. □

explanation We can describe this argument in terms of Turing machines. Suppose there were a Turing machine H that took as input a description of a Turing machine K and an input x, and decided whether or not K halts on input x. Then we could build another Turing machine G which takes a single input x, calls H to decide if machine x halts on input x, and does the opposite. In other words, if H reports that x halts on input x, G goes into an infinite loop, and if H reports that x doesn't halt on input x, then G just halts. Does G halt on input G? The argument above shows that it does if and only if it doesn't—a contradiction. So our supposition that there is a such Turing machine H, is false.

3.8 Comparison with Russell's Paradox

It is instructive to compare and contrast the arguments in this section with Russell's paradox: cmp:thy:rus:sec

1. Russell's paradox: let $S = \{x : x \notin x\}$. Then $x \in S$ if and only if $x \notin S$, a contradiction.

   Conclusion: There is no such set S. Assuming the existence of a "set of all sets" is inconsistent with the other axioms of set theory.

2. A modification of Russell's paradox: let F be the "function" from the set of all functions to $\{0,1\}$, defined by

$$F(f) = \begin{cases} 1 & \text{if } f \text{ is in the domain of } f\text{, and } f(f) = 0 \\ 0 & \text{otherwise} \end{cases}$$

   A similar argument shows that $F(F) = 0$ if and only if $F(F) = 1$, a contradiction.

   Conclusion: F is not a function. The "set of all functions" is too big to be the domain of a function.

3. The diagonalization argument: let $f_0, f_1, \ldots$ be the enumeration of the partial computable functions, and let $G\colon \mathbb{N} \to \{0,1\}$ be defined by

$$G(x) = \begin{cases} 1 & \text{if } f_x(x) \downarrow = 0 \\ 0 & \text{otherwise} \end{cases}$$

   If G is computable, then it is the function f_k for some k. But then $G(k) = 1$ if and only if $G(k) = 0$, a contradiction.

   Conclusion: G is not computable. Note that according to the axioms of set theory, G is still a function; there is no paradox here, just a clarification.

That talk of partial functions, computable functions, partial computable functions, and so on can be confusing. The set of all partial functions from $\mathbb{N}$ to $\mathbb{N}$ is a big collection of objects. Some of them are total, some of them are computable, some are both total and computable, and some are neither. Keep in mind that when we say "function," by default, we mean a total function. Thus we have:

1. computable functions

2. partial computable functions that are not total

3. functions that are not computable

4. partial functions that are neither total nor computable

To sort this out, it might help to draw a big square representing all the partial functions from $\mathbb{N}$ to $\mathbb{N}$, and then mark off two overlapping regions, corresponding to the total functions and the computable partial functions, respectively. It is a good exercise to see if you can describe an object in each of the resulting regions in the diagram.

3.9 Computable Sets

We can extend the notion of computability from computable functions to computable sets: cmp:thy:cps:sec

Definition 3.7. Let S be a set of natural numbers. Then S is *computable* iff its characteristic function is. In other words, S is computable iff the function

$$\chi_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

is computable. Similarly, a relation $R(x_0, \dots, x_{k-1})$ is computable if and only if its characteristic function is.

explanation Computable sets are also called *decidable*.

Notice that we now have a number of notions of computability: for partial functions, for functions, and for sets. Do not get them confused! The Turing machine computing a partial function returns the output of the function, for input values at which the function is defined; the Turing machine computing a set returns either 1 or 0, after deciding whether or not the input value is in the set or not.

3.10 Computably Enumerable Sets

cmp:thy:ces:sec

Definition 3.8. A set is *computably enumerable* if it is empty or the range of a computable function.

Historical Remarks Computably enumarable sets are also called *recursively enumerable* instead. This is the original terminology, and today both are commonly used, as well as the abbreviations "c.e." and "r.e."

explanation You should think about what the definition means, and why the terminology is appropriate. The idea is that if S is the range of the computable function f, then

$$S = \{f(0), f(1), f(2), \dots\},$$

and so f can be seen as "enumerating" the elements of S. Note that according to the definition, f need not be an increasing function, i.e., the enumeration need not be in increasing order. In fact, f need not even be injective, so that the constant function $f(x) = 0$ enumerates the set $\{0\}$.

Any computable set is computably enumerable. To see this, suppose S is computable. If S is empty, then by definition it is computably enumerable. Otherwise, let a be any element of S. Define f by

$$f(x) = \begin{cases} x & \text{if } \chi_S(x) = 1 \\ a & \text{otherwise.} \end{cases}$$

Then f is a computable function, and S is the range of f.

3.11 Equivalent Defininitions of Computably Enumerable Sets

cmp:thy:eqc:sec The following gives a number of important equivalent statements of what it means to be computably enumerable.

cmp:thy:eqc:thm:ce-equiv **Theorem 3.9.** *Let S be a set of natural numbers. Then the following are equivalent:*

1. *S is computably enumerable.*
2. *S is the range of a* partial *computable function.*
3. *S is empty or the range of a primitive recursive function.*
4. *S is the* domain *of a partial computable function.*

The first three clauses say that we can equivalently take any non-empty computably enumerable set to be enumerated by either a computable function, a partial computable function, or a primitive recursive function. The fourth clause tells us that if S is computably enumerable, then for some index e, explanation

$$S = \{x : \varphi_e(x) \downarrow\}.$$

In other words, S is the set of inputs on for which the computation of φ_e halts. For that reason, computably enumerable sets are sometimes called *semi-decidable*: if a number is in the set, you eventually get a "yes," but if it isn't, you never get a "no"!

Proof. Since every primitive recursive function is computable and every computable function is partial computable, (3) implies (1) and (1) implies (2). (Note that if S is empty, S is the range of the partial computable function that is nowhere defined.) If we show that (2) implies (3), we will have shown the first three clauses equivalent.

So, suppose S is the range of the partial computable function φ_e. If S is empty, we are done. Otherwise, let a be any element of S. By Kleene's normal form theorem, we can write

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

In particular, $\varphi_e(x) \downarrow$ and $= y$ if and only if there is an s such that $T(e, x, s)$ and $U(s) = y$. Define $f(z)$ by

$$f(z) = \begin{cases} U((z)_1) & \text{if } T(e, (z)_0, (z)_1) \\ a & \text{otherwise.} \end{cases}$$

Then f is primitive recursive, because T and U are. Expressed in terms of Turing machines, if z codes a pair $\langle (z)_0, (z)_1 \rangle$ such that $(z)_1$ is a halting computation of machine e on input $(z)_0$, then f returns the output of the computation; otherwise, it returns a.We need to show that S is the range of f, i.e., for any natural number y, $y \in S$ if and only if it is in the range of f. In the forwards direction, suppose $y \in S$. Then y is in the range of φ_e, so for some x and s, $T(e, x, s)$ and $U(s) = y$; but then $y = f(\langle x, s \rangle)$. Conversely, suppose y is in the range of f. Then either $y = a$, or for some z, $T(e, (z)_0, (z)_1)$ and $U((z)_1) = y$. Since, in the latter case, $\varphi_e(x) \downarrow= y$, either way, y is in S.

(The notation $\varphi_e(x) \downarrow= y$ means "$\varphi_e(x)$ is defined and equal to y." We could just as well use $\varphi_e(x) = y$, but the extra arrow is sometimes helpful in reminding us that we are dealing with a partial function.)

To finish up the proof of Theorem 3.9, it suffices to show that (1) and (4) are equivalent. First, let us show that (1) implies (4). Suppose S is the range of a computable function f, i.e.,

$$S = \{y : \text{for some } x, f(x) = y\}.$$

Let

$$g(y) = \mu x\; f(x) = y.$$

Then g is a partial computable function, and $g(y)$ is defined if and only if for some x, $f(x) = y$. In other words, the domain of g is the range of f. Expressed in terms of Turing machines: given a Turing machine F that enumerates the elements of S, let G be the Turing machine that semi-decides S by searching through the outputs of F to see if a given element is in the set.

Finally, to show (4) implies (1), suppose that S is the domain of the partial computable function φ_e, i.e.,

$$S = \{x : \varphi_e(x) \downarrow\}.$$

If S is empty, we are done; otherwise, let a be any element of S. Define f by

$$f(z) = \begin{cases} (z)_0 & \text{if } T(e, (z)_0, (z)_1) \\ a & \text{otherwise.} \end{cases}$$

Then, as above, a number x is in the range of f if and only if $\varphi_e(x) \downarrow$, i.e., if and only if $x \in S$. Expressed in terms of Turing machines: given a machine M_e that semi-decides S, enumerate the elements of S by running through all possible Turing machine computations, and returning the inputs that correspond to halting computations. □

The fourth clause of Theorem 3.9 provides us with a convenient way of enumerating the computably enumerable sets: for each e, let W_e denote the domain of φ_e. Then if A is any computably enumerable set, $A = W_e$, for some e.

The following provides yet another characterization of the computably enumerable sets.

Theorem 3.10. *A set S is computably enumerable if and only if there is a computable relation $R(x, y)$ such that* cmp:thy:eqc:thm:exists-char

$$S = \{x : \exists y\, R(x, y)\}.$$

Proof. In the forward direction, suppose S is computably enumerable. Then for some e, $S = W_e$. For this value of e we can write S as

$$S = \{x : \exists y\, T(e, x, y)\}.$$

In the reverse direction, suppose $S = \{x : \exists y\, R(x, y)\}$. Define f by

$$f(x) \simeq \mu y\ AtomRx, y.$$

Then f is partial computable, and S is the domain of f. □

3.12 Computably Enumerable Sets are Closed under Union and Intersection

cmp:thy:clo:sec The following theorem gives some closure properties on the set of computably enumerable sets.

Theorem 3.11. *Suppose A and B are computably enumerable. Then so are $A \cap B$ and $A \cup B$.*

Proof. Theorem 3.9 allows us to use various characterizations of the computably enumerable sets. By way of illustration, we will provide a few different proofs.

For the first proof, suppose A is enumerated by a computable function f, and B is enumerated by a computable function g. Let

$$h(x) = \mu y\ (f(y) = x \lor g(y) = x) \text{ and}$$
$$j(x) = \mu y\ (f((y)_0) = x \land g((y)_1) = x).$$

Then $A \cup B$ is the domain of h, and $A \cap B$ is the domain of j.

Here is what is going on, in computational terms: given procedures that enumerate A and B, we can semi-decide if an element x is in $A \cup B$ by looking for x in either enumeration; and we can semi-decide if an element x is in $A \cap B$ for looking for x in both enumerations at the same time. explanation

For the second proof, suppose again that A is enumerated by f and B is enumerated by g. Let

$$k(x) = \begin{cases} f(x/2) & \text{if } x \text{ is even} \\ g((x-1)/2) & \text{if } x \text{ is odd.} \end{cases}$$

Then k enumerates $A \cup B$; the idea is that k just alternates between the enumerations offered by f and g. Enumerating $A \cap B$ is tricker. If $A \cap B$ is empty, it is trivially computably enumerable. Otherwise, let c be any element of $A \cap B$, and define l by

$$l(x) = \begin{cases} f((x)_0) & \text{if } f((x)_0) = g((x)_1) \\ c & \text{otherwise.} \end{cases}$$

In computational terms, l runs through pairs of elements in the enumerations of f and g, and outputs every match it finds; otherwise, it just stalls by outputting c.

For the last proof, suppose A is the *domain* of the partial function $m(x)$ and B is the domain of the partial function $n(x)$. Then $A \cap B$ is the domain of the partial function $m(x) + n(x)$.

explanation In computational terms, if A is the set of values for which m halts and B is the set of values for which n halts, $A \cap B$ is the set of values for which both procedures halt.

Expressing $A \cup B$ as a set of halting values is more difficult, because one has to simulate m and n in parallel. Let d be an index for m and let e be an index for n; in other words, $m = \varphi_d$ and $n = \varphi_e$. Then $A \cup B$ is the domain of the function

$$p(x) = \mu y\ (T(d, x, y) \vee T(e, x, y)).$$

explanation In computational terms, on input x, p searches for either a halting computation for m or a halting computation for n, and halts if it finds either one. □

3.13 Computably Enumerable Sets not Closed under Complement

Suppose A is computably enumerable. Is the complement of A, $\overline{A} = \mathbb{N} \setminus A$, necessarily computably enumerable as well? The following theorem and corollary show that the answer is "no." cmp:thy:cmp:sec

Theorem 3.12. *Let A be any set of natural numbers. Then A is computable if and only if both A and $\overline{A}$ are computably enumerable.* cmp:thy:cmp:thm:ce-comp

Proof. The forwards direction is easy: if A is computable, then $\overline{A}$ is computable as well ($\chi_A = 1 \dot{-} \chi_{\overline{A}}$), and so both are computably enumerable.

In the other direction, suppose A and $\overline{A}$ are both computably enumerable. Let A be the domain of φ_d, and let $\overline{A}$ be the domain of φ_e. Define h by

$$h(x) = \mu s\ (T(d, x, s) \vee T(e, x, s)).$$

In other words, on input x, h searches for either a halting computation of φ_d or a halting computation of φ_e. Now, if $x \in A$, it will succeed in the first case, and if $x \in \overline{A}$, it will succeed in the second case. So, h is a total computable function. But now we have that for every x, $x \in A$ if and only if $T(e, x, h(x))$, i.e., if φ_e is the one that is defined. Since $T(e, x, h(x))$ is a computable relation, A is computable. □

explanation It is easier to understand what is going on in informal computational terms: to decide A, on input x search for halting computations of φ_e and φ_f. One of them is bound to halt; if it is φ_e, then x is in A, and otherwise, x is in $\overline{A}$.

Corollary 3.13. *$\overline{K_0}$ is not computably enumerable.* cmp:thy:cmp:cor:comp-k

Proof. We know that K_0 is computably enumerable, but not computable. If $\overline{K_0}$ were computably enumerable, then K_0 would be computable by Theorem 3.12. □

3.14 Reducibility

cmp:thy:red:sec We now know that there is at least one set, K_0, that is computably enumerable but not computable. It should be clear that there are others. The method of reducibility provides a powerful method of showing that other sets have these properties, without constantly having to return to first principles. explanation

Generally speaking, a "reduction" of a set A to a set B is a method of transforming answers to whether or not elements are in B into answers as to whether or not elements are in A. We will focus on a notion called "many-one reducibility," but there are many other notions of reducibility available, with varying properties. Notions of reducibility are also central to the study of computational complexity, where efficiency issues have to be considered as well. For example, a set is said to be "NP-complete" if it is in NP and every NP problem can be reduced to it, using a notion of reduction that is similar to the one described below, only with the added requirement that the reduction can be computed in polynomial time.

We have already used this notion implicitly. Define the set K by

$$K = \{x : \varphi_x(x) \downarrow\},$$

i.e., $K = \{x : x \in W_x\}$. Our proof that the halting problem in unsolvable, Theorem 3.6, shows most directly that K is not computable. Recall that K_0 is the set

$$K_0 = \{\langle e, x\rangle : \varphi_e(x) \downarrow\}.$$

i.e. $K_0 = \{\langle x, e\rangle : x \in W_e\}$. It is easy to extend any proof of the uncomputability of K to the uncomputability of K_0: if K_0 were computable, we could decide whether or not an element x is in K simply by asking whether or not the pair $\langle x, x\rangle$ is in K_0. The function f which maps x to $\langle x, x\rangle$ is an example of a *reduction* of K to K_0.

Definition 3.14. Let A and B be sets. Then A is said to be *many-one reducible* to B, written $A \leq_m B$, if there is a computable function f such that for every natural number x,

$$x \in A \quad \text{if and only if} \quad f(x) \in B.$$

If A is many-one reducible to B and vice-versa, then A and B are said to be *many-one equivalent*, written $A \equiv_m B$.

If the function f in the definition above happens to be injective, A is said to be *one-one reducible* to B. Most of the reductions described below meet this stronger requirement, but we will not use this fact.

digression

It is true, but by no means obvious, that one-one reducibility really is a stronger requirement than many-one reducibility. In other words, there are infinite sets A and B such that A is many-one reducible to B but not one-one reducible to B.

3.15 Properties of Reducibility

The intuition behind writing $A \leq_m B$ is that A is "no harder than" B. The following two propositions support this intuition. cmp:thy:ppr:sec

Proposition 3.15. *If $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$.* *cmp:thy:ppr:prop:trans-red*

Proof. Composing a reduction of A to B with a reduction of B to C yields a reduction of A to C. (You should check the details!) □

Proposition 3.16. *Let A and B be any sets, and suppose A is many-one reducible to B.* *cmp:thy:ppr:prop:reduce*

1. *If B is computably enumerable, so is A.*
2. *If B is computable, so is A.*

Proof. Let f be a many-one reduction from A to B. For the first claim, just check that if B is the domain of a partial function g, then A is the domain of $g \circ f$:

$$\begin{aligned} x \in A & \text{iff } f(x) \in B \\ & \text{iff } g(f(x)) \downarrow . \end{aligned}$$

For the second claim, remember that if B is computable then B and $\overline{B}$ are computably enumerable. It is not hard to check that f is also a many-one reduction of $\overline{A}$ to $\overline{B}$, so, by the first part of this proof, A and $\overline{A}$ are computably enumerable. So A is computable as well. (Alternatively, you can check that $\chi_A = \chi_B \circ f$; so if χ_B is computable, then so is χ_A.) □

digression A more general notion of reducibility called *Turing reducibility* is useful in other contexts, especially for proving undecidability results. Note that by Corollary 3.13, the complement of K_0 is not reducible to K_0, since it is not computably enumerable. But, intuitively, if you knew the answers to questions about K_0, you would know the answer to questions about its complement as well. A set A is said to be Turing reducible to B if one can determine answers to questions in A using a computable procedure that can ask questions about B. This is more liberal than many-one reducibility, in which (1) you are only allowed to ask one question about B, and (2) a "yes" answer has to translate to a "yes" answer to the question about A, and similarly for "no." It is still the case that if A is Turing reducible to B and B is computable then A is computable as well (though, as we have seen, the analogous statement does not hold for computable enumerability).

You should think about the various notions of reducibility we have discussed, and understand the distinctions between them. We will, however, only deal with many-one reducibility in this chapter. Incidentally, both types of reducibility discussed in the last paragraph have analogues in computational complexity, with the added requirement that the Turing machines run in polynomial time: the complexity version of many-one reducibility is known as *Karp reducibility*, while the complexity version of Turing reducibility is known as *Cook reducibility*.

3.16 Complete Computably Enumerable Sets

cmp:thy:cce:sec

Definition 3.17. A set A is a *complete computably enumerable set* (under many-one reducibility) if

1. A is computably enumerable, and
2. for any other computably enumerable set B, $B \leq_m A$.

In other words, complete computably enumerable sets are the "hardest" computably enumerable sets possible; they allow one to answer questions about *any* computably enumerable set.

Theorem 3.18. *K, K_0, and K_1 are all complete computably enumerable sets.*

Proof. To see that K_0 is complete, let B be any computably enumerable set. Then for some index e,

$$B = W_e = \{x : \varphi_e(x) \downarrow\}.$$

Let f be the function $f(x) = \langle e, x\rangle$. Then for every natural number x, $x \in B$ if and only if $f(x) \in K_0$. In other words, f reduces B to K_0.

To see that K_1 is complete, note that in the proof of Proposition 3.19 we reduced K_0 to it. So, by Proposition 3.15, any computably enumerable set can be reduced to K_1 as well.

K can be reduced to K_0 in much the same way. □

Problem 3.1. Give a reduction of K to K_0.

So, it turns out that all the examples of computably enumerable sets that we have considered so far are either computable, or complete. This should seem strange! Are there any examples of computably enumerable sets that are neither computable nor complete? The answer is yes, but it wasn't until the middle of the 1950s that this was established by Friedberg and Muchnik, independently. digression

3.17 An Example of Reducibility

Let us consider an application of Proposition 3.16. cmp:thy:k1:sec

Proposition 3.19. *Let* cmp:thy:k1:prop:k1

$$K_1 = \{e : \varphi_e(0) \downarrow\}.$$

Then K_1 is computably enumerable but not computable.

Proof. Since $K_1 = \{e : \exists s\, T(e, 0, s)\}$, K_1 is computably enumerable by Theorem 3.10.

To show that K_1 is not computable, let us show that K_0 is reducible to it.

explanation This is a little bit tricky, since using K_1 we can only ask questions about computations that start with a particular input, 0. Suppose you have a smart friend who can answer questions of this type (friends like this are known as "oracles"). Then suppose someone comes up to you and asks you whether or not $\langle e, x\rangle$ is in K_0, that is, whether or not machine e halts on input x. One thing you can do is build another machine, e_x, that, for *any* input, ignores that input and instead runs e on input x. Then clearly the question as to whether machine e halts on input x is equivalent to the question as to whether machine e_x halts on input 0 (or any other input). So, then you ask your friend whether this new machine, e_x, halts on input 0; your friend's answer to the modified question provides the answer to the original one. This provides the desired reduction of K_0 to K_1.

Using the universal partial computable function, let f be the 3-ary function defined by

$$f(x, y, z) \simeq \varphi_x(y).$$

Note that f ignores its third input entirely. Pick an index e such that $f = \varphi^3_e$; so we have

$$\varphi^3_e(x, y, z) \simeq \varphi_x(y).$$

By the s-m-n theorem, there is a function $s(e, x, y)$ such that, for every z,

$$\begin{aligned} \varphi_{s(e,x,y)}(z) &\simeq \varphi^3_e(x, y, z) \\ &\simeq \varphi_x(y). \end{aligned}$$

explanation In terms of the informal argument above, $s(e, x, y)$ is an index for the machine that, for any input z, ignores that input and computes $\varphi_x(y)$.

In particular, we have

$$\varphi_{s(e,x,y)}(0) \downarrow \quad \text{if and only if} \quad \varphi_x(y) \downarrow .$$

In other words, $\langle x, y\rangle \in K_0$ if and only if $s(e, x, y) \in K_1$. So the function g defined by

$$g(w) = s(e, (w)_0, (w)_1)$$

is a reduction of K_0 to K_1. □

3.18 Totality is Undecidable

cmp:thy:tot:sec

Let us consider one more example of using the s-m-n theorem to show that something is noncomputable. Let Tot be the set of indices of total computable functions, i.e.

$$\text{Tot} = \{x : \text{for every } y,\ \varphi_x(y)\downarrow\}.$$

cmp:thy:tot:prop:total

Proposition 3.20. Tot *is not computable.*

Proof. To see that Tot is not computable, it suffices to show that K is reducible to it. Let $h(x, y)$ be defined by

$$h(x, y) \simeq \begin{cases} 0 & \text{if } x \in K \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that $h(x, y)$ does not depend on y at all. It should not be hard to see that h is partial computable: on input x, y, the we compute h by first simulating the function φ_x on input x; if this computation halts, $h(x, y)$ outputs 0 and halts. So $h(x, y)$ is just $Z(\mu s\, T(x, x, s))$, where Z is the constant zero function.

Using the s-m-n theorem, there is a primitive recursive function $k(x)$ such that for every x and y,

$$\varphi_{k(x)}(y) = \begin{cases} 0 & \text{if } x \in K \\ \text{undefined} & \text{otherwise} \end{cases}$$

So $\varphi_{k(x)}$ is total if $x \in K$, and undefined otherwise. Thus, k is a reduction of K to Tot. □

It turns out that Tot is not even computably enumerable—its complexity lies further up on the "arithmetic hierarchy." But we will not worry about this strengthening here. digression

3.19 Rice's Theorem

cmp:thy:rce:sec

If you think about it, you will see that the specifics of Tot do not play into the proof of Proposition 3.20. We designed $h(x, y)$ to act like the constant function $j(y) = 0$ exactly when x is in K; but we could just as well have made it act like any other partial computable function under those circumstances. This observation lets us state a more general theorem, which says, roughly, that no nontrivial property of computable functions is decidable.

Keep in mind that $\varphi_0, \varphi_1, \varphi_2, \ldots$ is our standard enumeration of the partial computable functions.

Theorem 3.21 (Rice's Theorem)**.** *Let C be any set of partial computable functions, and let $A = \{n : \varphi_n \in C\}$. If A is computable, then either C is $\emptyset$ or C is the set of all the partial computable functions.*

An *index set* is a set A with the property that if n and m are indices which "compute" the same function, then either both n and m are in A, or neither is. It is not hard to see that the set A in the theorem has this property. Conversely, if A is an index set and C is the set of functions computed by these indices, then $A = \{n : \varphi_n \in C\}$.

explanation

With this terminology, Rice's theorem is equivalent to saying that no nontrivial index set is decidable. To understand what the theorem says, it is helpful to emphasize the distinction between *programs* (say, in your favorite programming language) and the functions they compute. There are certainly questions about programs (indices), which are syntactic objects, that are computable: does this program have more than 150 symbols? Does it have more than 22 lines? Does it have a "while" statement? Does the string "hello world" every appear in the argument to a "print" statement? Rice's theorem says that no nontrivial question about the program's *behavior* is computable. This includes questions like these: does the program halt on input 0? Does it ever halt? Does it ever output an even number?

Proof of Rice's theorem. Suppose C is neither $\emptyset$ nor the set of all the partial computable functions, and let A be the set of indices of functions in C. We will show that if A were computable, we could solve the halting problem; so A is not computable.

Without loss of generality, we can assume that the function f which is nowhere defined is not in C (otherwise, switch C and its complement in the argument below). Let g be any function in C. The idea is that if we could decide A, we could tell the difference between indices computing f, and indices computing g; and then we could use that capability to solve the halting problem.

Here's how. Using the universal computation predicate, we can define a function

$$h(x, y) \simeq \begin{cases} \text{undefined} & \text{if } \varphi_x(x) \uparrow \\ g(y) & \text{otherwise.} \end{cases}$$

To compute h, first we try to compute $\varphi_x(x)$; if that computation halts, we go on to compute $g(y)$; and if *that* computation halts, we return the output. More formally, we can write

$$h(x, y) \simeq P^2_0(g(y), \mathrm{Un}(x, x)).$$

where $P^2_0(z_0, z_1) = z_0$ is the 2-place projection function returning the 0-th argument, which is computable.

Then h is a composition of partial computable functions, and the right side is defined and equal to $g(y)$ just when $\mathrm{Un}(x, x)$ and $g(y)$ are both defined.

Notice that for a fixed x, if $\varphi_x(x)$ is undefined, then $h(x, y)$ is undefined for every y; and if $\varphi_x(x)$ is defined, then $h(x, y) \simeq g(y)$. So, for any fixed value of x, either $h(x, y)$ acts just like f or it acts just like g, and deciding whether or not $\varphi_x(x)$ is defined amounts to deciding which of these two cases holds. But this amounts to deciding whether or not $h_x(y) \simeq h(x,y)$ is in C or not, and if A were computable, we could do just that.

More formally, since h is partial computable, it is equal to the function φ_k for some index k. By the s-m-n theorem there is a primitive recursive function s such that for each x, $\varphi_{s(k,x)}(y) = h_x(y)$. Now we have that for each x, if $\varphi_x(x)\downarrow$, then $\varphi_{s(k,x)}$ is the same function as g, and so $s(k,x)$ is in A. On the other hand, if $\varphi_x(x)\uparrow$, then $\varphi_{s(k,x)}$ is the same function as f, and so $s(k,x)$ is not in A. In other words we have that for every x, $x \in K$ if and only if $s(k,x) \in A$. If A were computable, K would be also, which is a contradiction. So A is not computable. □

Rice's theorem is very powerful. The following immediate corollary shows some sample applications.

Corollary 3.22. *The following sets are undecidable.*

1. $\{x : 17 \text{ is in the range of } \varphi_x\}$
2. $\{x : \varphi_x \text{ is constant}\}$
3. $\{x : \varphi_x \text{ is total}\}$
4. $\{x : \text{whenever } y < y',\ \varphi_x(y)\downarrow, \text{ and if } \varphi_x(y')\downarrow, \text{ then } \varphi_x(y) < \varphi_x(y')\}$

Proof. These are all nontrivial index sets. □

3.20 The Fixed-Point Theorem

cmp:thy:fix:sec Let's consider the halting problem again. As temporary notation, let us write $\ulcorner\varphi_x(y)\urcorner$ for $\langle x,y\rangle$; think of this as representing a "name" for the value $\varphi_x(y)$. With this notation, we can reword one of our proofs that the halting problem is undecidable.

Question: is there a computable function h, with the following property? For every x and y,

$$h(\ulcorner\varphi_x(y)\urcorner) = \begin{cases} 1 & \text{if } \varphi_x(y)\downarrow \\ 0 & \text{otherwise.} \end{cases}$$

Answer: No; otherwise, the partial function

$$g(x) \simeq \begin{cases} 0 & \text{if } h(\ulcorner\varphi_x(x)\urcorner) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

would be computable, and so have some index e. But then we have

$$\varphi_e(e) \simeq \begin{cases} 0 & \text{if } h(\ulcorner\varphi_e(e)\urcorner) = 0 \\ \text{undefined} & \text{otherwise,} \end{cases}$$

in which case $\varphi_e(e)$ is defined if and only if it isn't, a contradiction.

Now, take a look at the equation with φ_e. There is an instance of self-reference there, in a sense: we have arranged for the value of $\varphi_e(e)$ to depend on $\ulcorner\varphi_e(e)\urcorner$, in a certain way. The fixed-point theorem says that we *can* do this, in general—not just for the sake of proving contradictions.

Lemma 3.23 gives two equivalent ways of stating the fixed-point theorem. Logically speaking, the fact that the statements are equivalent follows from the fact that they are both true; but what we really mean is that each one follows straightforwardly from the other, so that they can be taken as alternative statements of the same theorem.

Lemma 3.23. *The following statements are equivalent:* cmp:thy:fix:lem:fixed-equiv

1. *For every partial computable function $g(x, y)$, there is an index e such that for every y,*
$$\varphi_e(y) \simeq g(e, y).$$

2. *For every computable function $f(x)$, there is an index e such that for every y,*
$$\varphi_e(y) \simeq \varphi_{f(e)}(y).$$

Proof. (1) $\Rightarrow$ (2): Given f, define g by $g(x, y) \simeq \mathrm{Un}(f(x), y)$. Use (1) to get an index e such that for every y,

$$\begin{aligned}\varphi_e(y) &= \mathrm{Un}(f(e), y)\\ &= \varphi_{f(e)}(y).\end{aligned}$$

(2) $\Rightarrow$ (1): Given g, use the s-m-n theorem to get f such that for every x and y, $\varphi_{f(x)}(y) \simeq g(x, y)$. Use (2) to get an index e such that

$$\begin{aligned}\varphi_e(y) &= \varphi_{f(e)}(y)\\ &= g(e, y).\end{aligned}$$

This concludes the proof. □

explanation Before showing that statement (1) is true (and hence (2) as well), consider how bizarre it is. Think of e as being a computer program; statement (1) says that given any partial computable $g(x, y)$, you can find a computer program e that computes $g_e(y) \simeq g(e, y)$. In other words, you can find a computer program that computes a function that references the program itself.

Theorem 3.24. *The two statements in Lemma 3.23 are true. Specifically, for every partial computable function $g(x, y)$, there is an index e such that for every y,*

$$\varphi_e(y) \simeq g(e, y).$$

Proof. The ingredients are already implicit in the discussion of the halting problem above. Let $\mathrm{diag}(x)$ be a computable function which for each x returns an index for the function $f_x(y) \simeq \varphi_x(x, y)$, i.e.

$$\varphi_{\mathrm{diag}(x)}(y) \simeq \varphi_x(x, y).$$

Think of diag as a function that transforms a program for a 2-ary function into a program for a 1-ary function, obtained by fixing the original program as its first argument. The function diag can be defined formally as follows: first define s by

$$s(x, y) \simeq \mathrm{Un}^2(x, x, y),$$

where Un^2 is a 3-ary function that is universal for partial computable 2-ary functions. Then, by the s-m-n theorem, we can find a primitive recursive function diag satisfying

$$\varphi_{\mathrm{diag}(x)}(y) \simeq s(x, y).$$

Now, define the function l by

$$l(x, y) \simeq g(\mathrm{diag}(x), y).$$

and let $\ulcorner l \urcorner$ be an index for l. Finally, let $e = \mathrm{diag}(\ulcorner l \urcorner)$. Then for every y, we have

$$\begin{aligned}
\varphi_e(y) &\simeq \varphi_{\mathrm{diag}(\ulcorner l \urcorner)}(y) \\
&\simeq \varphi_{\ulcorner l \urcorner}(\ulcorner l \urcorner, y) \\
&\simeq l(\ulcorner l \urcorner, y) \\
&\simeq g(\mathrm{diag}(\ulcorner l \urcorner), y) \\
&\simeq g(e, y),
\end{aligned}$$

as required. □

What's going on? Suppose you are given the task of writing a computer program that prints itself out. Suppose further, however, that you are working with a programming language with a rich and bizarre library of string functions. In particular, suppose your programming language has a function diag which works as follows: given an input string s, diag locates each instance of the symbol 'x' occuring in s, and replaces it by a quoted version of the original string. For example, given the string explanation

```
hello x world
```

as input, the function returns

```
hello 'hello x world' world
```

as output. In that case, it is easy to write the desired program; you can check that

```
print(diag('print(diag(x))'))
```

does the trick. For more common programming languages like C++ and Java, the same idea (with a more involved implementation) still works.

We are only a couple of steps away from the proof of the fixed-point theorem. Suppose a variant of the print function $\mathrm{print}(x, y)$ accepts a string x and another numeric argument y, and prints the string x repeatedly, y times. Then the "program"

```
getinput(y); print(diag('getinput(y); print(diag(x), y)'), y)
```

prints itself out y times, on input y. Replacing the getinput—print—diag skeleton by an arbitrary funtion $g(x, y)$ yields

```
g(diag('g(diag(x), y)'), y)
```

which is a program that, on input y, runs g on the program itself and y. Thinking of “quoting” with “using an index for,” we have the proof above.

For now, it is o.k. if you want to think of the proof as formal trickery, or black magic. But you should be able to reconstruct the details of the argument given above. When we prove the incompleteness theorems (and the related “fixed-point theorem”) we will discuss other ways of understanding why it works.

digression The same idea can be used to get a “fixed point” combinator. Suppose you have a lambda term g, and you want another term k with the property that k is β-equivalent to gk. Define terms

$$\mathrm{diag}(x) = xx$$

and

$$l(x) = g(\mathrm{diag}(x))$$

using our notational conventions; in other words, l is the term $\lambda x.\, g(xx)$. Let k be the term ll. Then we have

$$\begin{aligned} k &= (\lambda x.\, g(xx))(\lambda x.\, g(xx)) \\ &\triangleright g((\lambda x.\, g(xx))(\lambda x.\, g(xx))) \\ &= gk. \end{aligned}$$

If one takes

$$Y = \lambda g.\, ((\lambda x.\, g(xx))(\lambda x.\, g(xx)))$$

then Yg and $g(Yg)$ reduce to a common term; so $Yg \equiv_\beta g(Yg)$. This is known as “Curry's combinator.” If instead one takes

$$Y = (\lambda xg.\, g(xxg))(\lambda xg.\, g(xxg))$$

then in fact Yg reduces to $g(Yg)$, which is a stronger statement. This latter version of Y is known as “Turing's combinator.”

3.21 Applying the Fixed-Point Theorem

The fixed-point theorem essentially lets us define partial computable functions in terms of their indices. For example, we can find an index e such that for every y, cmp:thy:apf:sec

$$\varphi_e(y) = e + y.$$

As another example, one can use the proof of the fixed-point theorem to design a program in Java or C++ that prints itself out.

Remember that if for each e, we let W_e be the domain of φ_e, then the sequence W_0, W_1, W_2, ... enumerates the computably enumerable sets. Some of these sets are computable. One can ask if there is an algorithm which takes as input a value x, and, if W_x happens to be computable, returns an index for its characteristic function. The answer is "no," there is no such algorithm:

Theorem 3.25. *There is no partial computable function f with the following property: whenever W_e is computable, then $f(e)$ is defined and $\varphi_{f(e)}$ is its characteristic function.*

Proof. Let f be any computable function; we will construct an e such that W_e is computable, but $\varphi_{f(e)}$ is not its characteristic function. Using the fixed point theorem, we can find an index e such that

$$\varphi_e(y) \simeq \begin{cases} 0 & \text{if } y = 0 \text{ and } \varphi_{f(e)}(0) \downarrow = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

That is, e is obtained by applying the fixed-point theorem to the function defined by

$$g(x, y) \simeq \begin{cases} 0 & \text{if } y = 0 \text{ and } \varphi_{f(x)}(0) \downarrow = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Informally, we can see that g is partial computable, as follows: on input x and y, the algorithm first checks to see if y is equal to 0. If it is, the algorithm computes $f(x)$, and then uses the universal machine to compute $\varphi_{f(x)}(0)$. If this last computation halts and returns 0, the algorithm returns 0; otherwise, the algorithm doesn't halt.

But now notice that if $\varphi_{f(e)}(0)$ is defined and equal to 0, then $\varphi_e(y)$ is defined exactly when y is equal to 0, so $W_e = \{0\}$. If $\varphi_{f(e)}(0)$ is not defined, or is defined but not equal to 0, then $W_e = \emptyset$. Either way, $\varphi_{f(e)}$ is not the characteristic function of W_e, since it gives the wrong answer on input 0. □

3.22 Defining Functions using Self-Reference

cmp:thy:slf:sec

It is generally useful to be able to define functions in terms of themselves. For example, given computable functions k, l, and m, the fixed-point lemma tells us that there is a partial computable function f satisfying the following equation for every y:

$$f(y) \simeq \begin{cases} k(y) & \text{if } l(y) = 0 \\ f(m(y)) & \text{otherwise.} \end{cases}$$

Again, more specifically, f is obtained by letting

$$g(x,y) \simeq \begin{cases} k(y) & \text{if } l(y) = 0 \\ \varphi_x(m(y)) & \text{otherwise} \end{cases}$$

and then using the fixed-point lemma to find an index e such that $\varphi_e(y) = g(e,y)$.

For a concrete example, the "greatest common divisor" function $\gcd(u,v)$ can be defined by

$$\gcd(u,v) \simeq \begin{cases} v & \text{if } 0 = 0 \\ \gcd(\mathrm{mod}(v,u),u) & \text{otherwise} \end{cases}$$

where $\mathrm{mod}(v,u)$ denotes the remainder of dividing v by u. An appeal to the fixed-point lemma shows that gcd is partial computable. (In fact, this can be put in the format above, letting y code the pair $\langle u,v\rangle$.) A subsequent induction on u then shows that, in fact, gcd is total.

Of course, one can cook up self-referential definitions that are much fancier than the examples just discussed. Most programming languages support definitions of functions in terms of themselves, one way or another. Note that this is a little bit less dramatic than being able to define a function in terms of an *index* for an algorithm computing the functions, which is what, in full generality, the fixed-point theorem lets you do.

3.23 Minimization with Lambda Terms

When it comes to the lambda calculus, we've shown the following: cmp:thy:mla:sec

1. Every primitive recursive function is represented by a lambda term.
2. There is a lambda term Y such that for any lambda term G, $YG \triangleright G(YG)$.

To show that every partial computable function is represented by some lambda term, we only need to show the following.

Lemma 3.26. *Suppose $f(x,y)$ is primitive recursive. Let g be defined by*

$$g(x) \simeq \mu y\; f(x,y) = 0.$$

Then g is represented by a lambda term.

Proof. The idea is roughly as follows. Given x, we will use the fixed-point lambda term Y to define a function $h_x(n)$ which searches for a y starting at n; then $g(x)$ is just $h_x(0)$. The function h_x can be expressed as the solution of a fixed-point equation:

$$h_x(n) \simeq \begin{cases} n & \text{if } f(x,n) = 0 \\ h_x(n+1) & \text{otherwise.} \end{cases}$$

Here are the details. Since f is primitive recursive, it is represented by some term F. Remember that we also have a lambda term D such that $D(M, N, \overline{0}) \triangleright M$ and $D(M, N, \overline{1}) \triangleright N$. Fixing x for the moment, to represent h_x we want to find a term H (depending on x) satisfying

$$H(\overline{n}) \equiv D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})).$$

We can do this using the fixed-point term Y. First, let U be the term

$$\lambda h.\, \lambda z.\, D(z, (h(Sz)), F(x, z)),$$

and then let H be the term YU. Notice that the only free variable in H is x. Let us show that H satisfies the equation above.

By the definition of Y, we have

$$H = YU \equiv U(YU) = U(H).$$

In particular, for each natural number n, we have

$$\begin{aligned} H(\overline{n}) &\equiv U(H, \overline{n}) \\ &\triangleright D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})), \end{aligned}$$

as required. Notice that if you substitute a numeral $\overline{m}$ for x in the last line, the expression reduces to $\overline{n}$ if $F(\overline{m}, \overline{n})$ reduces to $\overline{0}$, and it reduces to $H(S(\overline{n}))$ if $F(\overline{m}, \overline{n})$ reduces to any other numeral.

To finish off the proof, let G be $\lambda x.\, H(\overline{0})$. Then G represents g; in other words, for every m, $G(\overline{m})$ reduces to reduces to $\overline{g(m)}$, if $g(m)$ is defined, and has no normal form otherwise. $\square$

Photo Credits

Bibliography