Chapter udf

Computability Theory

Material in this chapter should be reviewed and expanded. In paticular, there are no exercises yet.

thy.1 Introduction

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_k^n 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 $f(\vec{x}, 0), \ldots, f(\vec{x}, y-1)$ 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 explicitly, 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.

thy.2 Coding Computations

In every model of computation, it is possible to do the following:

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.

thy.3 The Normal Form Theorem

Theorem thy.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.

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 thy.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.

thy.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.

Theorem thy.3. *For each pair of natural numbers n and m, there is a primitive recursive function s_n^m such that for every sequence x, a_0, ..., a_{m-1}, y_0 ,..., y_{n-1}, we have*

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

explanation   It is helpful to think of s_n^m as acting on *programs*. That is, s_n^m 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, $s_n^m(x, a_0, \ldots, a_{m-1})$, for the n-ary function of the remaining arguments. It you think of x as the description of a Turing machine, then $s_n^m(x, a_0, \ldots, a_{m-1})$ 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_n^m is then just a primitive recursive function that finds a code for the appropriate Turing machine.

thy.5 The Universal Partial Computable Function

Theorem thy.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:*

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. $\square$

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

computability-theory rev: 445393f (2018-08-14) by OLP / CC–BY

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.

thy.6   No Universal Computable Function

Theorem thy.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 thy.4](#) above shows that we can get around this diago-
nalization 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 $h(k)$?

thy.7   The Halting Problem

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.

Theorem thy.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 thy.5. $\square$

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. $\square$

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.

thy.8 Comparison with Russell's Paradox

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

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, ... 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.

thy.9 Computable Sets

We can extend the notion of computability from computable functions to computable sets:

Definition thy.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 $R(x_0, \ldots, x_{k-1})$ 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.

thy.10 Computably Enumerable Sets

Definition thy.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

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

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.

thy.11   Equivalent Defininitions of Computably Enumerable Sets

The following gives a number of important equivalent statements of what it means to be computably enumerable.

Theorem thy.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,

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

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 thy.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. $\qquad \square$

The fourth clause of Theorem thy.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 thy.10. *A set S is computably enumerable if and only if there is a* *computable relation $R(x, y)$ such that*

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

computability-theory rev: 445393f (2018-08-14) by OLP / CC–BY

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

$$f(x) \simeq \mu y \, Atom R x, y.$$

Then f is partial computable, and S is the domain of f. $\qquad\square$

thy.12 Computably Enumerable Sets are Closed under Union and Intersection

The following theorem gives some closure properties on the set of computably enumerable sets.

Theorem thy.11. *Suppose A and B are computably enumerable. Then so are $A \cap B$ and $A \cup B$.*

Proof. Theorem thy.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

$$h(x) = \mu y \, (f(y) = x \vee g(y) = x) \text{ and}$$
$$j(x) = \mu y \, (f((y)_0) = x \wedge g((y)_1) = x).$$

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 [explanation] 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.

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. $\square$

thy.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 thy.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 ($\chi_A = 1 \mathbin{\dot{-}} \chi_{\overline{A}}$), 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. $\square$

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 thy.13. $\overline{K_0}$ *is not computably enumerable.*

cmp:thy:cmp: cor:comp-k

12   *computability-theory* rev: 445393f (2018-08-14) by OLP / CC–BY

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 thy.12. $\qquad\square$

thy.14 Reducibility

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.

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 thy.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 thy.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.

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.

thy.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.

Proposition thy.15. *If $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$.*

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!) $\square$

Proposition thy.16. *Let A and B be any sets, and suppose A is many-one reducible to B.*

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 $g \circ f$:

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

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.) $\square$

A more general notion of reducibility called *Turing reducibility* is useful in other contexts, especially for proving undecidability results. Note that by Corollary thy.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).

computability-theory rev: 445393f (2018-08-14) by OLP / CC–BY

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*.

thy.16 Complete Computably Enumerable Sets

Definition thy.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 thy.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 thy.19 we reduced K_0 to it. So, by Proposition thy.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. $\square$

Problem thy.1. Give a reduction of K to K_0.

So, it turns out that all the examples of computably enumerable sets that digression
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.

thy.17 An Example of Reducibility

Let us consider an application of Proposition thy.16.

Proposition thy.19. *Let*

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

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 thy.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 $f = \varphi_e^3$; so we have

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

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

$$\varphi_{s(e,x,y)}(z) \simeq \varphi_e^3(x, y, z)$$
$$\simeq \varphi_x(y).$$

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. $\square$

thy.18   Totality is Undecidable

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.

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

Proposition thy.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. $\qquad\square$

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

thy.19   Rice's Theorem

If you think about it, you will see that the specifics of Tot do not play into the proof of Proposition thy.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 thy.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 $A = \{n : \varphi_n \in C\}$.

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

$$h(x, y) \simeq P_0^2(g(y), \text{Un}(x, x)).$$

where $P_0^2(z_0, z_1) = z_0$ 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 $\text{Un}(x, x)$ 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. $\qquad\square$

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

Corollary thy.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. $\qquad\square$

thy.20 The Fixed-Point Theorem

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 thy.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 thy.23. *The following statements are equivalent:*

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

(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
$$\varphi_e(y) = \varphi_{f(e)}(y)$$
$$= g(e, y).$$

This concludes the proof. $\square$

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 thy.24. *The two statements in Lemma thy.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.
$$\varphi_{\mathrm{diag}(x)}(y) \simeq \varphi_x(x, y).$$

computability-theory rev: 445393f (2018-08-14) by OLP / CC–BY

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 explanation 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

```
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.

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

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

and

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

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

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

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."

thy.21 Applying the Fixed-Point Theorem

The fixed-point theorem essentially lets us define partial computable func-
tions in terms of their indices. For example, we can find an index e such that
for every y,

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

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 $W_0, W_1, W_2, \ldots$ 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 thy.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. $\qquad \square$

thy.22 Defining Functions using Self-Reference

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.

thy.23 Minimization with Lambda Terms

When it comes to the lambda calculus, we've shown the following:

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 thy.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:

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

computability-theory rev: 445393f (2018-08-14) by OLP / CC–BY

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. $\qquad\square$

Photo Credits

Bibliography