Basic Computability Theory
Lecture Notes COMP 330 Autumn 2014

Prakash Panangaden

5th November 2015

These notes are a basic introduction to the *abstract* part of basic computability theory. It is assumed that the reader understands what an algorithm is and has seen many different models of computation including Turing machines, **while** programs and perhaps others. You do not have to be familiar with all of them, of course. It is sufficient to be proficient in at least one programming language and to be aware that Turing machines, **while** programs, λ-calculus, RAM machines and all programming languages are essentially equivalent, so it does not matter which formalism is used to express an algorithm. I expect the readers to be familiar with the idea that there are problems that cannot be solved algorithmically: for example, the halting problem.

Here we will study the functions computed by algorithms. A **computable** function is just a function for which there is an algorithm to compute it. An essential aspect of computability is that to produce a finite piece of information requires only a finite piece of input. Thus, for example, a Turing machine may have an infinite tape, but the input is only on a finite piece of it.

1 Partial Functions

One of the basic facts of life is that a lot of algorithms do not terminate on all their inputs; indeed, many never terminate. In order to define the functions computed by algorithms we need to work with *partial* functions. When I write "function" I will mean "partial function"; if I want to talk about what we usually call "functions," I will say "total function."

We typically use the natural numbers $\mathbf{N} = \{0, 1, 2, \ldots\}$ as the basic data type. We will write $f : \mathbf{N} \rightharpoonup \mathbf{N}$ to indicate that the partial function f takes a natural number as an argument and *may* produce a natural number as a result. This means that there are values of n for which $f(n)$ may not be defined. If we write $f(n) = m$ we mean that f is defined at n and produces the value m. We write $f(n)\uparrow$ to indicate that $f(n)$ is undefined, and $f(n)\downarrow$ to indicate that $f(n)$ is defined.

The function f is an abstract mathematical entity: if we pass to a partial function an argument where it is not defined there is simply no result; it is not as if the function "signals" that it is undefined. It is more sensible to ask "what do we see when we run the algorithm for a partial computable function where the function is not defined?" In this case the algorithm will never return and we will be waiting forever; at any finite stage we will not know for sure whether it is going to terminate or not. The algorithm does *not* print on the screen "Hey, I'm not terminating!"
Definition 1. The **domain** of a partial function f, is the set of values for which f is defined. The **range** of a partial function f is the set of values that f can return.

$$\mathsf{dom}(f) = \{n | f(n)\downarrow\}, \quad \mathsf{ran}(f) = \{n | \exists m. f(m) = n\}.$$

If we have two partial functions f and g, we write $f = g$ to mean that

$$\mathsf{dom}(f) = \mathsf{dom}(g) \text{ and } \forall n \in \mathsf{dom}(f),\ f(n) = g(n).$$

Definition 2. We say that g **extends** f if $\mathsf{dom}(f) \subseteq \mathsf{dom}(g)$ and $\forall n \in \mathsf{dom}(f)\ f(n) = g(n)$.

The partial function g can be defined in places where f is not defined. The *empty* function has domain $\emptyset$; it is nowhere defined. Clearly this is a computable function.

2 Remarks on Algorithms and Data types

The use of natural numbers is not important: I could have used strings, lists or whatever data type of *finite objects* that you like. The *real numbers* are not finite entities and computability on the reals is a story that I will not tell here. I will not normally spell out algorithms in some formal notation like C++ or Turing machines. I will use informal – but convincing – descriptions.

It is important to understand what does and does not count as an acceptable algorithm. Here is one example that I see students use in the homework quite often: "I will check for every possible string if..." Of course this is not possible if you are working with an infinite set of strings. Another danger is the following: "I will run the algorithm with input x and if it fails to halts I will ..." This is also not an algorithmic step, because we cannot always know *in finite time* whether an algorithm halts or not.

We can always code descriptions as numbers. First note that we can (and usually do) write programs as strings of ascii characters. Every ascii character can be viewed as a number so every string can be viewed as a (very large) number. Thus, we can think of a program in Java (or whatever language you use) as a natural number. We can think of Turing machines as being given descriptions in some formal language as a string and, in the same way, we can view this string as a number. I will often say, "the code number of an algorithm A" meaning the number obtained in this way without being specific about the coding scheme. If M is a Turing machine I will often write $\langle M\rangle$ to mean the code number for M, and for an algorithm A, in some algorithmic notation, like pseudo-code, I will write $\langle A\rangle$ for the code number of A.

We can always view a pair of numbers as a single number. For example, given the pair $\langle n, m\rangle$ we can represent it as the single number $x = 2^n 3^m$ (this time I *do* mean arithmetic exponentiation!), and we can recover the original pair by factoring the number x. Clearly, we can represent arbitrary finite sequences of numbers in a similar way: a lot of information can be packed into a single number!

3 Computable and Computably Enumerable Sets

We now come to the all-important central concepts of the subject. Given a set $X \subseteq \mathbf{N}$, a subset of the natural numbers, we define its characteristic function by the following formula:

$$1_X(n) = \begin{cases} 1 & \text{if } n \in X \\ 0 & \text{if } n \notin X \end{cases}$$

Note that the characteristic function is *always* a total function.

Definition 3. A set $x \subseteq \mathbf{N}$ is **computable** or **decidable** if its (total) characteristic function is computable.

A computable set is one for which we have a decision procedure, *i.e.* one which *always* terminates with a definite answer. For example, the set of prime numbers is computable: we definitely have an algorithm that can take a number as input and *decide* whether it is prime *or not*. This algorithm will always terminate.

If we write $\langle M,\ x\rangle$ to be the encoding of a pair of numbers where $\langle M\rangle$ is the code number of a Turing machine and x is the code of an input string we know that the set below is not computable

$$\mathsf{A_{TM}} \stackrel{\text{def}}{=} \{\langle M,\ x\rangle | M \text{ accepts } x\}.$$

The following set is also not computable

$$\mathsf{H_{TM}} \stackrel{\text{def}}{=} \{\langle M,\ x\rangle | M \text{ halts on } x\}.$$

The next proposition illustrates what can be done with these ideas.
Proposition 4. An *infinite* set X, of natural numbers is computable if and only if it is the range of some **total non-decreasing** computable function f.

Proof. It is one of the assumptions that X is infinite and hence that the range of f is infinite. Suppose we have an f as described in the proposition and A is the algorithm that computes it. We give a decision procedure for X as follows. We want to know if a given x is in X. Run A successively on $0, 1, 2, \ldots$ and look at the outputs. If the output is x we stop and say "Yes, x is in X," if the output is less than x we go on with the next input, if the output is more than x we stop and say "No, x is not in X." This algorithm has to terminate (please convince yourself that this is true) and it decides the question.

Suppose that we know that X is computable and that B is an algorithm that decides the question "is x in X?" We define f as follows. Given n the input for f we run B successively on $0, 1, 2, \ldots$, every time we find a number in X we store it in a list and count how many numbers we have seen. When we hit the nth number we return this as the output. Convince yourself that this always terminates. Thus, f is computable and non-decreasing and its range is exactly X. ∎

The computable sets are the first step in a hierarchy of increasingly complicated sets. The next step is the class of computably enumerable sets; these are affectionately called the CE sets.

Definition 5. A set $X \subseteq \mathbf{N}$ is called **computably enumerable** (CE) if there is an algorithm A that lists all the members of X, and only the members of X, in some arbitrary order.

In this case we do not expect A to halt, except in the rare cases where we are enumerating a finite set, because it has to list all the elements of X. Each individual entry in the list is produced *at some finite stage* and we can inspect the list as it is coming out. It is very important to understand this definition carefully. A set is CE if there is an algorithm with the following properties: if you give it an element in the set the algorithm is *guaranteed to terminate and say that the element is in the set*, if you give it an element not in the set it may terminate and say so or *it may loop forever without giving any answer*. If we wait long enough we will get a "yes" answer for every element in the set but the elements come out in some arbitrary order so we can never be sure if a particular element that we are waiting for is not in the set or whether it will appear later.

Theorem 6. A set $X \subseteq \mathbf{N}$ is CE if and only if any one of the following equivalent conditions hold.

(i) X is the domain of a computable function

(ii) the **semi-characteristic** function of X

$$\mathcal{S}_X(n) = \begin{cases} 1 & \text{if } n \in X \\ \text{undefined} & \text{if } n \notin X \end{cases}$$

is computable.

(iii) X is the range of a computable function

Proof. Clearly (ii) immediately implies (i). We show that the definition of CE implies (ii). Suppose that we have an enumerating algorithm A for X. To compute $\mathcal{S}_X(n)$ we run A and inspect the list as it comes out checking for n. If n does appear on the list we output 1; of course if n does not appear, we wait forever, but this is fine, we only want to construct a *semi*-characteristic function.

Now we show that (i) (and hence (ii)) implies that X is CE. Suppose that B is an algorithm to compute f and $\mathsf{dom}(f) = X$. This looks hard, we cannot say "run B on $0, 1, 2, \ldots$ and output any numbers for which it terminates."

The algorithm B may fail to terminate on the very first number. However, there is a technique called "dovetailing" which allows one to do just this apparently impossible task. We first run B on 0 for one step and store the state of the computation. Then we run B on 0 for one (more) step and on 1 for one step; then we run B on 0 for two more steps and on 1 for two more steps and on 2 for two steps. In every phase we run the computation longer and we include more arguments in our simulation. As soon as B terminates on any input we print this input and continue. Of course, as soon as one of the sub-computations terminates we remove it from the list of computations that are active. If $n \in \mathsf{dom}(f)$, then this procedure will eventually run B on n long enough to determine that it has terminated and n will get enumerated. Thus X is CE.

If, in the dovetailing procedure, we print the outputs with which B terminates instead of the inputs, we see that (iii) implies that X is CE. To show that X is CE implies (iii) we proceed a follows. We know that the semi-characteristic function is computable by some algorithm C. We define a partial function g as follows:

$$g(n) = \begin{cases} n & \text{if } C \text{ terminates on } n \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Clearly g is computable and its range is X.

∎

The basic closure properties of CE sets are given in the next few theorems.
Theorem 7. The union and intersection of two (hence any finite number of) CE sets is CE.

Proof. Suppose that X and Y are two CE sets with enumerating algorithms A and B respectively. We run A and B is parallel and as soon as an item appears on the list we enumerate it. It is only slightly more complicated to prove the analogous result for intersection but this is a homework question so I will leave it to you. ∎

The following simple theorem is called Post's theorem.
Theorem 8 (Post)**.** (a) If X is computable it is also CE.

(b) If X is CE and its complement $\overline{X}$ is also CE then X is computable.

Proof. The first part is trivial; but that does not make it unimportant.

For the second part, we assume that A enumerates X and that B enumerates $\overline{X}$. In order to show that X is computable we need a decision procedure for X. We run both A and B in parallel and as soon as one of them enumerates n we know whether n is in the set X or not. This has to terminate because n is in one of the sets and the enumerator for that set has to produce the answer in some finite time. ∎

The following theorem is easy but *very* important. We work with pairs of numbers $\langle n,\ m\rangle$. We know that we can define a *total* computable function that converts pairs of naturals into individual naturals and that there are computable functions in the reverse direction as well. We write $\pi_1(\langle n,\ m\rangle) = n$ and $\pi_2(\langle n,\ m\rangle) = m$ for these functions; they are called the projection functions. In fact (exercise for you, I will show you the answer in later notes) you can make the pair function and its inverses bijective. We will just assume that we have enriched our language with a pair constructor and the projections.

Theorem 9. A set $X \subseteq \mathbf{N}$ is CE if and only if there is a **computable** set Y of pairs of natural numbers such that

$$\forall x, x \in X \Leftrightarrow \exists y \in \mathbf{N}(\langle x,\ y\rangle \in Y).$$

Proof. If Y is computable (or even just CE) we can enumerate its elements and use π_1 to extract the first components. So if such a Y exists then X is certainly CE. If X is CE, with enumerating algorithm A, we construct Y as follows:

$$Y = \{\langle x,\ n\rangle | A \text{ enumerates } x \text{ within } n \text{ steps}\}.$$

Clearly Y is decidable because we are putting a time bound on the running of the algorithm. Clearly, if x is in X then A enumerates x after some number of steps so

$$\forall x, x \in X \Leftrightarrow \exists n \in \mathbf{N}(\langle x,\ n\rangle \in Y).$$

∎