Recursion Theory and Undecidability

Lecture Notes for CSCI 303 by David Kempe

March 26, 2008

In the second half of the class, we will explore limits on computation. These are questions of the form "What types of things can be computed at all?" "What types of things can be computed *efficiently*?" "How fast can problem XYZ be solved?" We already saw some positive examples in class: problems we could solve, and solve efficiently. For instance, we saw that we can sort n numbers in time $O(n \log n)$. Or compute the edit distance of two strings in time $O(nm)$. But we might wonder if that's the best possible. Maybe, there are still faster algorithms out there. Could we *prove* that these problems cannot be solved faster?

In order to answer these questions, we first need to be precise about what exactly we mean by "a problem" and by "to compute". In the case of an algorithm, we can go with "we know it when we see it". We can look at an algorithm, and agree that it only uses "reasonable" types of statements. When we try to prove lower bounds, or prove that a problem cannot be solved at all, we need to be more precise: after all, an algorithm could contain the statement "Miraculously and instantaneously guess the solution to the problem." We would probably all agree that that's cheating. But how about something like "Instantaneously find all the prime factors of the number n"? Would that be legal?

This writeup focuses only on the question of what problems can be solved *at all*. Running time lower bounds and the theory of NP-completeness are covered in the textbook, and we will cover them in class next. The material covered here is necessarily only a small excerpt of what's known in the field. For more information on these topics, check out Part II of the book by Sipser [6], the book by Hopcroft, Motwani, and Ullman [3], or the book by Kfoury, Moll, and Arbib [4].

1 Problems

Let's first start, though, by defining what types of problems we are looking at. For starters, it would be nice not to have to talk about graphs, and arrays, and strings, and all those. It turns out that all of them can be encoded as strings (take your favorite format for writing down the input in a file). And strings in turn can be read off as a binary number. So we really lose nothing (except notational convenience) by focusing only on strings or numbers, as we see fit.

The simplest type of problem is one in which we only want a Yes/No answer to a question about the input. We've already seen several of those in class. In that case, we can identify the problem with the set of strings/numbers for which the *correct answer* is "Yes". We call those the *"Yes" instances*, and usually denote them by upper-case letters such as X, Y. Thus, a problem X is simply a set. An algorithm solving (or *deciding*, as it is called) X gets as input a string x, and should output "Yes" (or 1) if $x \in X$, and "No" (or 0) if $x \notin X$. Notice that the set X will almost always contain infinitely many elements. For instance, X could be the set of all bipartite graphs (remember the "Wrestler" problem). Given a string x describing a graph, an algorithm would then have to output "Yes" if the graph described by it is bipartite.

This definition is what we will usually refer to as a *problem*. However, we will sometimes also be interested in computing a function. There, both the input and output are numbers. Again, since these numbers can be interpreted as anything, this does not restrict the types of computations possible. For instance, the input string x could describe a graph G with edge weights, and the output $y = f(x)$ could be interpreted as a set of edges to include in a minimum spanning tree. The reason we focus on numbers exclusively is that it makes several formal definitions a little less cumbersome to state.

There is one slight addition to the notion of a function, since we want to explicitly allow ourselves to model the function that is computed by a program that ends up in an infinite loop somewhere. So in addition to natural numbers, we allow the *output* of a function f to take on the special value $\perp$, which we interpret as "undefined". A function which is defined on all inputs (i.e., never takes on the value $\perp$) is also called *total*.

2 Computation and Programs

The next thing we need to do is define what exactly we mean by "computing". In the positive cases, i.e., when we can give examples of *being able to compute something*, we can look at the specification of the algorithm, and decide if it looks acceptable as an algorithm. But to prove *impossibility results*, we need to be precise about what we allow, and what we don't.

2.1 Turing Machines

The classical model for computation is the *Turing Machine*, named after Alan Turing, who proposed it in his seminal paper [7]. Books such as [3, 6] cover it in detail. The goal is to be as simple a model of "computing" as one can come up with, yet very powerful.

A Turing Machine consists of a *head* (processor) and a *tape* (memory). The tape is infinite and 1-dimensional, and consists of cells. Each cell contains a character $\sigma \in \Sigma$ from an *alphabet* Σ. That alphabet could be $\{0, 1, \square\}$, for instance, where $\square$ denotes an empty cell.

The head is always located on one of the cells of the tape. It has a finite set of *states* Q it can be in. A *transition function* t tells the head what to do: when being in state $q \in Q$, and reading a symbol $\sigma \in \Sigma$, the transition function specifies: (1) with what to overwrite the symbol on the tape, (2) whether to move right or left one cell on the tape, and (3) which new state q' to enter. The Turing Machine always starts in a predefined *start state* s, and as soon as it reaches a predefined *finish state* f, it terminates.

We can think of the transition function as the program, and the state as where in the program the execution currently resides. The input is written initially on the tape, and the output will be either added to the tape, or overwriting the input. The advantage of this model is that it is really simple, and the way it works is much easier to specify than, say, each command in a reasonable programming language. On the other hand, that makes actually *using* it much more cumbersome. For your amusement, you might want to design a Turing Machine that adds two numbers written in binary.

2.2 Other Models of Computation

There are many natural models of computation. *Cellular Automata* are beloved by physicists and others. They consist of an infinite 1-dimensional or 2-dimensional (or higher-dimensional) array of little automata. Each automaton has an identical *transition function*. The transition function determines the next state of the automaton, based on its previous state and the previous states of its neighbors. All the automata execute synchronously. Input is encoded in the initial state of the automata. It might be initially surprising that with sufficiently complex transition functions, one can get cellular automata to perform any computation that a "regular computer" can perform. One of the most well-known cellular automaton models is the "Game of Life". It consists of a very small and simple rule set, yet it has been shown that one can simulate computers with it.

To mathematicians, the most common model used to be that of *recursive functions* [5]. Essentially, starting from basic arithmetic and recursive nesting of functions, it allows one to build up more functions. Every function that can be expressed in this way is considered *recursive* or *computable*. These models were devised as models of how mathematicians compute, long before actual computers were built.

2.3 A "Natural" Programming Language

For our purposes, we will think of programs as very simplified C or Java programs, as suggested in [4]. Memory consists of an infinite array of integer numbers, which can be addressed as an array. Importantly, integers are *not* restricted to any number of bits, but rather can be arbitrarily large. In terms of statements, our language has assignments, basic arithmetic (0, +1, -1 is really enough, as all the rest can be built up from it). In addition, it has basic logic, "if" and "while" statements, and allows blocking parts of the program using "begin" and "end". Everything else, including string manipulations, can be built up from these. It is a bit cumbersome, but not very difficult.

The "meaning" of each of the commands is exactly what they mean in C or Java. We could define them formally here, but that would be more formalism than necessary.

2.4 The Church-Turing Thesis

While the different models described above differ in how intuitive they are, and also slightly in how efficiently they allow us to "program" in them, there has been a remarkable observation: *So far, all natural models of computation that have been proposed have been shown to allow computing exactly the same functions.* That is, if a function f can be computed using a Turing Machine, then it can also be computed using a standard programming language, and vice versa. The way this is proved is by showing how any one of these models can simulate any other.

The conviction that this will hold for *all* "natural" models of computation, even those not defined yet, was made explicitly by Church, Kleene and Turing, and is called by subsets of those names, most often *Church Turing Thesis*. Obviously, there is no way to verify this, as "natural" is not formally defined.

One consequence of this observation is that each program is also a string, for instance the string describing its program code in binary. Thus, it is also a number, and we can simply number all programs in this way. Of course, many numbers will correspond to programs that are not syntactically well formed. For all those programs, we will simply define that they are the infinite loop, i.e., the function f they compute is undefined everywhere $f(n) = \perp$ for all n. (We could make another arbitrary definition instead for such malformed programs.)

Formally, we will denote all programs by $P_1, P_2, \ldots$. For simpler notation, we will use $P_i(j) \uparrow$ to denote that the program number i will not terminate on input j (it runs into an infinite loop somewhere). Similarly, $P_i(j) \downarrow$ denotes that program i on input j does terminate. Similar to the case of function, we call a program *total* if it terminates on all inputs.

3 Non-Computable Functions

There is a fairly simple argument that there must be functions that cannot be computed. The reason is that even if we restrict our attention to functions which only take on values $\{0, 1\}$, there are uncountably many of them. Each function can be written as an infinite string of 0s and 1s, where the k^{th} digit is the value of the function for input k. If we reinterpret this infinite string as a binary number, and put a decimal dot before it, then this exactly captures all real numbers in $[0, 1]$. There are uncountably infinitely many of those. (This will also follow from our diagonalization argument below.) On the other hand, there are only countably infinitely many different programs. After all, each program is a *finite* string, so we can number them. Say, we first number programs by length, and within each length alphabetically. Or we simply look at the numbers we defined above. Either way, we can identify programs with natural numbers. Thus, there must be functions not computed by any programs. In fact, almost all functions *cannot* be computed, even with infinite memory and arbitrary amounts of time.

While the above argument tells us that there are many function which cannot be computed by any programs, it does not tell us if there are any functions which we really care about, and which cannot be computed. It turns out that there are. The most famous and basic one is the *Halting Problem*. Informally, it is as follows: given a program P and an input x to that program, we want to decide if the program P

will terminate when run on input x. Clearly, this is an important task, if we want to avoid running our program into an endless loop. It would be really nice if compilers could do that for us. Formally, we define the function ϕ as:

$$\phi(P, x) \quad := \quad \begin{cases} 1 & \text{if } P(x) \text{ terminates} \\ 0 & \text{otherwise} \end{cases}$$

(Notice that we use two inputs here: However, we can write out the pair as a string, and simply interpret it as a number again. As before, every type of input can be written as a single number with the correct encoding.)

Theorem 1 *There is no program P which computes the function ϕ.*

Proof. The proof is by contradiction. Suppose that we had a program Φ which always terminates, and correctly computes ϕ, i.e., $\Phi(P, x) = \phi(P, x)$ for all programs P and all inputs x. We could then use the function Φ as a subroutine in the following program, which we call P', and which gets as input another program P:

(a) Run $\Phi(P, P)$. (That is, treat P as both the program and the input.)

(b) If it returns 1, then deliberately enter an endless loop.

(c) Otherwise, return 1.

Clearly, if Φ is a valid program, then so is P', as all it adds is an "if" statement, an endless loop (`while (0 == 0) do;`), and a return statement.

Now let's see what Φ outputs when we feed it P' as input for its program P and for its input x. If we run P' on input P', it will either terminate or it will not.

In the first case ($P'(P') \downarrow$), we have that $\phi(P', P') = 1$. Because Φ correctly computes ϕ by assumption, and always terminates, we also get that $\Phi(P', P') = 1$ in that case. But that means that our program P' will deliberately enter the infinite loop, so it will not terminate. That clearly contradicts the assumption that P' terminated on input P'. So we cannot be in that case.

In the second case, P' does not terminate on input P', so $\phi(P', P') = 0$. Again, because Φ compute ϕ correctly, we also get that $\Phi(P', P') = 0$. But then, the definition of P' shows that it will terminate and return 1. So we have that if P' does not terminate on input P', then it actually terminates on the same input. Again, that is a contradiction.

Since this covers all cases, we can only conclude that the remaining assumption must have been incorrect, i.e., there cannot be any program Φ which always terminates and correctly computes ϕ. ■

A few more remarks are in order about this theorem.

Remark 2 *It does not rule out writing a program that would solve the halting problem for many types of programs and many inputs. In fact, that is quite easy. What it rules out is writing a program that will solve this problem correctly for* every input*.*

Remark 3 *The proof crucially relies on having infinite memory and arbitrarily long integers. This seems like a reasonable abstract model of computation. However, in practice, computers have only finite memory. Counting all memory in hard drives, RAM, clocks, graphics cards, and so forth, let's conservatively estimate that a "normal" computer has at most $1,000,000,000,000$ bits of memory available. Then, we could solve the halting problem with a super-duper computer as follows: the super-duper computer simulates our regular computer on the input. For each step of the simulation, the super-duper computer records the state of all the bits in the regular computer in a large table. There can be at most $k := 2^{1000000000000}$ different such states (since each bit can be 0 or 1). So after at most k steps, the regular computer must either have terminated, or repeated some state. In the former case, the super-duper computer can safely report that the program terminated. In the latter case, the regular computer will be entering a cycle, since from the same state, it will repeat exactly the same sequence. Thus, it "only" takes the regular computer k steps to decide the halting*

problem. Of course, this is utterly impractical, as the super-duper computer would need a table of size k, far exceeding any number of anything known to exist in the universe. But it does show that for impossibility of computation to kick in, one needs arbitrarily large numbers.

Remark 4 *We can think of what the proof does as follows. Draw a 2-dimensional table as follows:*

$\downarrow$ *Program* | *Input* $\rightarrow$	1	2	3	4	
1	$\downarrow$ ($\uparrow$)	$\downarrow$	$\uparrow$	$\downarrow$	$\cdots$
2	$\downarrow$	$\uparrow$ ($\downarrow$)	$\uparrow$	$\downarrow$	$\cdots$
3	$\uparrow$	$\downarrow$	$\uparrow$ ($\downarrow$)	$\uparrow$	$\cdots$
4	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$ ($\uparrow$)	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The rows of the table go over all programs, and the columns over all inputs. Our assumption was that we have a program Φ which, when given the row and column, correctly returns whether the value in that cell is $\uparrow$ or $\downarrow$.

We now define the function ψ as indicated in the parentheses. That is, for program i, on input i, we do the opposite of what program i does. That makes sure that ψ cannot be equal to P_i (because they do something different on input i). Because the infinite table contains all programs, ψ is different from all programs, and can therefore not itself be computed by any program.

In essence, this is exactly identical to Cantor's Diagonalization Argument, showing that there are real numbers that are not rational. We simply need to replace $\uparrow$ and $\downarrow$ by 0 and 1. By enumerating all rational numbers, and making sure that our newly constructed number differs from each number in at least one digit, we ensure that it cannot itself be a rational number.

The slight twist here is the computation issue. If we assumed that there is a program Φ which can correctly return the value of any table entry, then we can also use it to write the program P' we defined above, which exactly computes ψ. That is how we got a contradiction.

Notice that from both the proof given above and the diagonalization argument in the remark, we actually proved something slightly stronger. Namely, that the following seemingly easier function can already not be computed: "given a program P_i, does P_i terminate on input i?" As opposed to being able to decide for *all* inputs whether P_i would terminate on them, this would only require us to be able to decide if P_i would terminate when given its own description as input. But, as we see above, that's already enough to construct P', and hence get the contradiction.

The set of all programs terminating on their own code as input is referenced so often that it has a name: we write HALT $:= \{i \mid P_i(i) \downarrow\}$ for the set of all programs that terminate on their own code. Deciding this set is also often called the Halting Problem. (That is, people are not specific about which version is called that.)

4 Decidable and Enumerable Sets

We have seen above that there are sets (such as HALT), for which we cannot come up with any program which will always correctly output if a given input i is in the set. This motivates the following definition:

Definition 5 (recursive, decidable) *(a) A function f is* recursive *or* computable *if there is a program P computing it, i.e., $P(x) = f(x)$ for all inputs x.*

(b) A set S is recursive*, or* recursively decidable *(or simply* decidable*) if there is a program P correctly deciding its membership. That is, P is total (it always terminates), and $P(x) = 1$ if $x \in S$, while $P(x) = 0$ if $x \notin S$.*

This captures exactly all sets which can be decided with programs that always terminate. A somewhat weaker notion is obtained if we only require that the function terminates when the input x is in the set, but it may run forever when $x \notin S$. This notion is also very important, so we define it:

Definition 6 (recursively enumerable (r.e.)) *A set S is* recursively enumerable *(*r.e. *for short) if there is a program P such that $P(x) = 1$ if $x \in S$, and $P(x) \uparrow$ if $x \notin S$.*

The name recursively enumerable is justified as follows: if a set S has this property, we can start computation of P for more and more inputs x in parallel. As soon as a computation $P(x)$ succeeds, the new program outputs x. If we interleave these computations carefully, the result is that each element of S will eventually be printed ("enumerated"), while no other elements will be. In that sense, S can be enumerated by a program.

Notice that every recursive set is also r.e. There are two ways to see this.

(a) If we have a deciding program P for S, then we can define a new program P' as follows: execute P on input x, and if P returns 1, then also return 1. If P returns 0, then enter an infinite loop.

(b) In terms of programming intuition, we do the following: run a loop over all integers x. Run the program P on x. If it returns 1, then output x, otherwise don't. Either way, move on to the next loop iteration. Clearly, this will output exactly all of S eventually, so it proves that S can be enumerated.

As a first example of a set that is r.e., but not recursive, we can return to HALT. The enumerating program for HALT is very simple: given input i, simply simulate $P_i(i)$. If it terminates, then output 1, otherwise, we are automatically stuck in an infinite loop. All that P needs to work is an ability to interpret programs; and we all know how to write an interpreter for programming languages.

5 Reductions

Reductions are an extremely important technique, both for the design of algorithms and for proving impossibility or hardness results. The name "reduction" seems to be confusing to students at times; perhaps "transformation" would be easier, since it does not have a connotation of making anything easier or smaller. But the name "reduction" is standard, so we'll stick with it.

The idea is quite simple. If you have two problems — say, X and Y — a reduction from X to Y would be a transformation that would take an input to X and turn it into an input to Y. Then, by solving the transformed version, you would also solve the original problem. Of course, the transformation (or reduction) must be designed carefully in order to have this property. If we can come up with such a reduction, it serves two potential purposes:

(a) If we already know how to solve Y, then we have just discovered a way to solve X as well. We have made use of this several times in class already, while designing algorithms for most reliable paths or for deciding if a graph is bipartite.

(b) If we already know that X is impossible to solve (or takes a long time), then we have shown the same for Y. For if Y were easy to solve, we could use it to solve X as well. This second view is the one that is most relevant in the context of recursion theory or the theory of NP-hardness.

5.1 An introductory example

Before we look a little more into the formalisms here, let's look at a somewhat whimsical example that doesn't really have anything to do with computation, but might help clarify the concepts because of that[1]. Imagine that we are considering two problems, one called Eradicate World Hunger, and the other Cold Fusion. The first problem is to ensure that everyone in the world has enough to eat. The second one is the scientific problem of accomplishing energy generation using cold fusion.

Both of these problems appear quite difficult, and we don't know if we can solve either of them. But that does not keep us from coming up with the following dream scenario. Imagine that we could solve Cold

[1] I would like to thank Matthew Haughom and Ben Lisbakken for many office hours discussions which led to the creation of this example.

Fusion. Then, with all the energy we can cheaply generate, we could desalinate water, build greenhouses, and by many other ways produce easily accessible plentiful food all over the world. (Yes, I admit that this is not very carefully planned.) Thus, we could use Cold Fusion to solve Eradicate World Hunger. What we have just done is sketch out a *reduction* from Eradicate World Hunger to Cold Fusion: we have shown a way to transform the problem to eradicate world hunger to that of solving cold fusion. Notice that in the process, we outlined how to use a *solution* to Cold Fusion, but we transformed the problems the other way.

What can we infer from this? For one, that Cold Fusion is at least as difficult as Eradicate World Hunger. For as soon as we had a solution for the former, we would also have one for the latter. As optimists, we would now hope for a solution to Cold Fusion. As pessimists, we might suspect (or prove?) that all world hunger can never be eradicated. If that were true, it would immediately prove that cold fusion is impossible as well. (All of this of course is relying on the reduction being correct.) What it does not prove is that Eradicate World Hunger is difficult. After all, it could be that there is a much easier way to accomplish the goal, without relying on Cold Fusion at all. (For instance, there might be an easy way to accomplish this using agricultural progress.) The reduction only showed that Cold Fusion is *at least* as difficult as Eradicate World Hunger.

Make sure you really understand what happened here. Reductions between problems for which we don't know how hard they are will keep popping up for the rest of the semester, and it is crucial to know what these implications mean.

5.2 More formal definitions

We have given some intuitive discussions of the notion of reductions, but no formal definitions. Let us now supply a more formal definitions.

Definition 7 *Let X and Y be problems (i.e., sets). A function f is a* reduction from X to Y *if: (1) $x \in X$ implies $f(x) \in Y$, and (2) $x \notin X$ implies $f(x) \notin Y$.*

Thus, a correct reduction function maps the "Yes" instances of the problem X to "Yes" instances of the problem Y, and "No" instances of the problem X to "No" instances of the problem Y.

In this part of class, we want to use reductions mostly for the following purpose: if we have a reduction from X to Y, then Y must be "at least as difficult" as X. (See the above example.) In order to justify this conclusion, we must also require that the reduction itself not be too powerful. After all, suppose that $X = \text{HALT}$, and $Y = \{i \mid i \text{ is odd}\}$. Clearly, Y is a very easy problem to decide, while X is hard. However, if the reduction function f itself were allowed to solve the halting problem, then we could reduce from X to Y.

Thus, for our purposes here, we will restrict reductions to *total and recursive functions*. That is, each reduction must be computable by a program which always terminates. Later on, when we get to the theory of NP-completeness, we will also require that the reduction program must run in polynomial time. If there is a total and recursive reduction from X to Y, we write $X \leq_m Y$. The "m" stands for "many-to-one", stressing the fact that multiple different inputs x, x' can be mapped to the same output $f(x) = f(x')$ by the reduction function f[2]. The notation $X \leq_m Y$ also suggests that Y is "at least as difficult" as X.

5.3 Examples

In this section, we will use reductions to prove two other problems undecidable. We could also do a fresh diagonalization argument like the proof of Theorem 1, but that would be way more work than reusing it.

(a) Define **42** := $\{P \mid$ Program P outputs 42 for at least one input$\}$. Thus, the problem consists in deciding, given a program, whether it can be made to output 42. What we will do is give a reduction from

[2]There are reasons why reductions mapping all x to distinct $f(x)$ are interesting to study, but these reasons are beyond this class.

HALT to **42** to prove that this problem is undecidable. Given that the input to either problem is a program, the reduction will be a "program transformation". It will get a program P, and change it around to some new program P'. The goal is that if $P(P) \downarrow$ (i.e., P terminates on its own source code as input), then the new program P' outputs 42 on at least one input. Otherwise, if $P(P) \uparrow$, then P' never outputs 42.

Having spelled out what we want our reduction to accomplish, it is actually now fairly easy to define the reduction: Given the program P, our new program is as follows:

```
1. Call Program P on input P.
2. If and when it returns, output 42.
```

Clearly, given the code for program P, outputting this code is an easy transformation. It just uses P as a subroutine. Notice also that the new program P' completely ignores its input.

What can we say about P'? If $P(P) \downarrow$, then for *every input*, P' will make it to line 2, and will output 42. On the other hand, if $P(P) \uparrow$, then for every input, P' will get stuck in step 1, and will never reach line 2. Thus, either P' outputs 42 for all inputs, or it does not terminate for any inputs. This means that if $P \in$ HALT, then $P' \in$ **42**, and if $P \notin$ HALT, then $P' \notin$ **42**. Thus, we have defined a correct reduction.

If there were a way to decide *for each program* P whether $P \in$ **42**, then we could use that decision procedure, together with the reduction, to solve the HALT problem. But we've proved that impossible earlier. Thus, **42** itself is also not decidable.

(b) Let's do another example. An important class of functions is that of all total functions, those which terminate on all inputs. So we define TOT $:= \{P \mid P(x) \downarrow \text{ for all inputs } x\}$. We will show that TOT is also undecidable. Again, let's specify what a reduction from HALT would have to do. We would need to take a program P, and transform it to a new program P', such that if $P(P) \downarrow$, then P' terminates on all inputs, and if $P(P) \uparrow$, then P' does not terminate for at least one input.

It turns out that exactly the same reduction as above works. Given a program P, we produce the following new program P'.

```
1. Call Program P on input P.
2. If and when it returns, output 42.
```

If $P(P) \downarrow$, then P' will always terminate (and output 42, although that doesn't matter here). On the other hand, if $P(P) \uparrow$, then P' will not terminate on any input. In particular, it will not be total. (Notice that P' is again ignoring its input.) This proves that the reduction is correct, and thus, TOT is undecidable.

In the next homeworks, you will get opportunities to practice several more examples of reductions like this.

6 Rice's Theorem

So far, we have seen a number of undecidable properties of programs, such as HALT, TOT, **42**, and others on homework assignments. In fact, it's starting to look like saying anything about the behavior of programs is quite hard. Are there any properties of programs that *are* decidable?

Well, that depends on what we mean by "property". Certainly, syntactic properties such as "the program contains at least 3 `while` loops" or properties such as "the first three executed commands on input 16 include an assignment" can be easily decided. But if we ask about properties of the *final output* of the program, things are much more dire. More specifically, we would like to know if there are any properties of the *function* computed by a program that are decidable. Let's make that precise.

Definition 8 *(a) We write $P \equiv P'$ to denote that P and P' compute the same functions, i.e., $P(x) = P'(x)$ for all inputs x.*

(b) A set S of programs respects functions *if whenever $P \in S$ and $P \equiv P'$, then also $P' \in S$.*

Intuitively, this means that if a set contains one program for a particular function, it must contain all programs for that function. So the set is not allowed to distinguish *how* the functions are computed, only *which* functions are in the set.

Theorem 9 (Rice's Theorem) *If S respect functions, $S \neq \emptyset, \overline{S} \neq \emptyset$, then S is undecidable.*

This theorem says that the only properties of functions that can be decided are the trivial ones: always say "Yes" or always say "No". Anything beyond that is undecidable.

The proof of this theorem is actually not terribly difficult; it is only a slight generalization of the reductions we did in Section 5.3. However, we will skip it in this class.

7 Kleene's Recursion Theorem

A lot of our arguments and diagonalization has implicitly used the notion of "self reference"; programs running on their own source code, a program "simulating itself" etc. This notion of self reference is at the core of much of recursion theory, and one of the most concise way of expressing it is Kleene's Recursion Theorem.

Theorem 10 (Kleene's Recursion Theorem) *Let f be any total computable function. Then, there is a program P_i such that P_i and $P_{f(i)}$ compute exactly the same function, i.e., $P_i \equiv P_{f(i)}$. (P_i is called a* fixpoint *of f.)*

What this theorem says is quite remarkable. Think of f as a "program rewriting function": it takes as input a program P_i, and alters it (a little bit, or completely) into a new program $P_{f(i)}$. Never mind how complicated the rewrites, there is a program P_i for which the new program computes exactly the same function as the old one. Notice that f may even completely garble the program. In all fairness, for many natural functions you could come up with, a program that will not be altered is the infinite loop. It might amuse you to try out some program rewrite functions, and find a fixpoint for each of them by hand.

The proof of this theorem is not very long, but also not all that intuitive, so we will not cover it in this class. However, we can obtain from it a very interesting corollary:

Corollary 11 *There is a program that exactly outputs its own source code.*

Notice that this would of course be easy with file access or something like that, but achieving this without file access (just `print`, `if`, `while`, and function calls) is not nearly as obvious.

Proof. We define the following program rewrite function f. As input, it gets a program P. It transforms P to a new program $f(P)$ (or, using the previous notation $P_{f(i)}$ if the input was P_i). The new program $f(P)$ is very simple: it just outputs the source code of P. That is, it completely ignores its own input.

```
f(P):
output P
```

By the Recursion Theorem, there is some program P such that $P \equiv f(P)$. But we know that $f(P)$ outputs the source code of P, and therefore, because the two are equivalent, P must also output the source code of P. ∎

7.1 A C Program printing its own source code

Having proved the existence of a program printing itself in the abstract, here is an actual program printing itself, for your amusement.

```
main(){char q=34,n=10,*a="main(){char q=34,n=10,*a=%c%s%c;printf(a,q,a,q,n);}%c";printf(a,q,a,q,n);}
```

8 Beyond Computable

By now, we're pretty comfortable with the fact that some functions cannot be computed. In particular, the problem HALT seems rather tough. At the beginning of this chapter, we were dreaming about magic commands which would let us solve all kinds of problems. So let's dream on for a bit. Let's take our programming language, call it C, and create a much improved version, called C+. The way in which C+ is better is that it has a built-in command for solving the halting program. That is, given i, it will correctly return whether $P_i(i) \downarrow$ or $P_i(i) \uparrow$. Never mind how this is implemented (we know it cannot be). This would be quite useful to have, for instance in a compiler. Now that we can solve the HALT problem, can we compute everything?

The answer is: not even close. After all, any program in C+ is still a finite string. So we can redo exactly the same proof as for Theorem 1, and prove that there is now a new halting problem which cannot be solved in C+, even though C+ is much more powerful. The new halting problem simply asks: given a C+ program P, will P terminate on its own source code as input? Since the function we added was only useful for deciding whether a C-program halted, it won't help us here. Of course, we could now add this new halting problem as a feature, and create a new language C++. And we could continue this as long as we want. With every additional halting problem solver, we get a more powerful language. But so long as programs are finite (and they will always be), even those new languages will compute only countably infinitely many functions, while there are uncountably many total. Thus, we will never even get close to being able to compute all functions.

In this way, we can define a hierarchy of harder and harder problems. There are alternate ways to define such hierarchies, based on quantifiers in logic formulas. Again, there is plenty more here to learn, but we can only cover so much in this class.

9 Some Historical Context

At the time when many of these results were first proved, either there were no computers, or they were in their infancy. Rather than "What can computers compute?", the question mathematicians were interested in was "What can mathematicians compute?", as well as "What can mathematicians prove?". The two were considered essentially the same question. This led to efforts, in the late 19^{th} century, to formalize mathematics completely, in terms of formal systems. These consisted of some basic axioms, and rules for deducing new facts from old ones formally, by entirely syntactic manipulations (symbol pushing). You might remember modus ponens rules and similar ones from CSCI 271. A common approach was to combine basic axioms about natural numbers with first-order logic ($\wedge, \vee, \neg, \forall, \exists$). The initial goal of the mathematicians pushing this program was to prove that *every* correct statement could be proved by these rules. A celebrated result by Gödel proved the contrary:

Theorem 12 (Gödel's Incompleteness Theorem [1]) *For every "sufficiently powerful" formal system including the natural numbers, there is a true formula that cannot be proved.*

We will not define formally what "sufficiently powerful" means, but suffice it to say that it includes most systems that one would consider. The idea of Gödel's proof is in fact quite similar to our diagonalization proof of Theorem 1. The main idea of the proof there was to construct a program that said "I am different from all programs" by making it differ from all other programs explicitly. Gödel constructed a formula essentially stating "I am different from all formulas provable using the formal system". The main difficulty

that Gödel had to overcome was encoding things like logical inference and the rules of the formal system in natural numbers. It turns out that arithmetic is rather cumbersome as far as programming languages go. Most of Gödel's paper is devoted to building up enough definitions to express logical inference in terms of arithmetic. Notice that a formula saying "I am different from all formulas provable using the formal system" must necessarily be true. If it were false, then it would be provable, and the system would permit proving a false statement. Such systems would be useless. So our "meta-reasoning" shows that the formula is true, yet it cannot be proved in the system.

In a sense, Turing's paper can be considered a significantly simplified version of Gödel's result, by using a much more intuitive and powerful model of computation, namely the Turing Machine. In that way, he avoided the significant work of building up the arithmetization of logic.

Following Gödel's result, there has been a lot of "interpretation" of what these results mean. Some, taking the formal systems to accurately reflect human reasoning, interpret it as proving limits on human understanding and discovery. In some cases, this inference is accompanied by metaphysical or religious overtones. Others interpret it as proving the superiority of humans over computers, since humans can do meta-thinking outside the box, while computers are bound to formal reasoning. A very entertaining, if not always scientifically sound, view on these and many related topics can be found in the book "Gödel, Escher, Bach" [2] by Douglas Hofstadter.

References

[1] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[2] D. Hofstsadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1999.

[3] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2006.

[4] A. Kfoury, R. Moll, and M. Arbib. *A Programming Approach to Computability*. Springer, 1982.

[5] H. Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.

[6] M. Sipser. *Introduction to the Theory of Computation*. Thomson, 2005.

[7] A. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2:230–265, 1936.