Decidable, Undeciable, and Beyond
Exposition by William Gasarch

1 Defining Decidable

How to pin down what is meant by "computable?" This definition is motivated by actual computers and resembles a machine. The definition is similar to that of a Deterministic Finite Automaton, or Push Down Automaton, but it can do much much more. Keep in mind that our final goal is to show that this model can compute a lot of functions.

We first give the formal definition, and then explain it intuitively.

Def 1.1 A *Turing Machine* M is a quintuple $(Q, \Sigma, \delta, q_0, h)$ where

1. Q is a finite set of *states*
2. Σ is the *alphabet* (the function computed by the Turing Machine will go from Σ^* to Σ^*), It contains a special symbol B which stands for BLANK.
3. δ is the *next move function*, it goes from $(Q - \{h\}) \times \Sigma$ to $Q \times \{\Sigma \cup \{L, R\}\}$
4. $q_0 \in Q$ is the *start state*
5. $h \in Q$ is the *halting state*

The machine acts in discrete steps. At any one step it will read the symbol in the "tape square", see what state it is in, and do one of the following:

1. write a symbol on the tape square and change state,
2. move the head one symbol to the left and change state,
3. move the head one symbol to the right and change state.

We now formally say how the machine computes a function. This will be followed by intuition.

Def 1.2 Let M be a Turing Machine. An *Instantaneous Description* (ID) of M is a string of the form $\alpha_1 q \alpha_2$ where $\alpha_1, \alpha_2 \in \Sigma^*$, $q \in Q$, and the last symbol of α_2 is not B. Intuitively, an ID describes the current status of the TM and Tape.

Def 1.3 Let M be a Turing Machine. Let $\alpha_1, \alpha_2, \alpha_3, \alpha_4 \in \Sigma^*$, and $q, q' \in Q$. Let $\alpha_1 = x_1 x_2 \cdots x_k$, and $\alpha_2 = x_{k+1} x_{k+2} \cdots x_n$. The symbol $\alpha_1 q \alpha_2 \vdash_M \alpha_3 q' \alpha_4$ means that one of the following is true:

1. $\delta(q, x_k) = (q', y)$, $\alpha_3 = x_1 x_2 \cdots x_{k-1} y$ and $\alpha_4 = \alpha_2$.
2. $\delta(q, x_k) = (q', L)$, $\alpha_3 = x_1 x_2 \cdots x_{k-1}$ and $\alpha_4 = x_k x_{k+1} \cdots x_n$.
3. $\delta(q, x_k) = (q', R)$, $\alpha_3 = x_1 x_2 \cdots x_{k+1}$ and $\alpha_4 = x_{k+2} x_{k+3} \cdots x_n$.

Intuitively, the above definition is saying that if the Turing Machine is in ID C, and $C \vdash_M D$, then after *one* more move the TM will be in D. The next definition is about what will happen after *many* moves.

Def 1.4 If C and D are IDs then $C \vdash_M^* D$ means that either $C = D$ or there exist a finite set of IDs $C_1, C_2, \ldots, C_k$ such that $C = C_1$, for all i, $C_i \vdash_M C_{i+1}$, and $C_k = D$.

Def 1.5 Let M be a Turing Machine. Recall that the *partial function computed by Turing Machine M* is the following partial function: $f(x)$ is the unique y (if it exists) such that $xq_0 \vdash_M^* yh$. If no such y exists then $M(y)$ is said to diverge.

Intuitively we start out with x laid out on the tape, and the head looking at the rightmost symbol of x. The machine then runs, and if it gets to the halt state with the condition that there are only blanks to the right of the head, then the string to the left of the head is the value $f(x)$.

For examples of Turing machines, and exercises on building them to do things, see Lewis and Papadimitriou's text "Elements of the Theory of Computation.", or the Hopcroft-Ullman White book. Other books also contain this material.

Note that, just like a computer, the computation of a Turing machine is in discrete steps.

Def 1.6 Let M_e be a Turing Machine and s be a number. The partial function computed by $M_{e,s}$ is the function that, on input x, runs $M_e(x)$ for s steps and if it has halted by then, outputs whatever $M_e(x)$ output, else diverges.

Note that the function computed by $M_{e,s}$ is intuitively computable. Although it is a partial function we can tell when it will be undefined so we can think of it as being total.

Notation 1.7 If $M(y)$ is defined we write $M(y) \downarrow$. If $M(y)$ diverges then we write $M(y) \uparrow$.

2 Variations of a Turing Machine

There are many variations on Turing Machines that could be defined- allowing extra tapes, extra heads, allowing it to operate on a two dimensional grid instead of a one dimensional tape, etc. All of these models end up being equivalent. This adds to the intuition that Turing Machines are powerful.

Def 2.1 A k-tape Turing Machine is a quintuple $(Q, \Sigma, \delta, q_0, h)$ such that Q, Σ, q_0, and h are as in a normal Turing Machine, but δ is a function from $(Q - \{h\}) \times \Sigma^k$ into $Q \times \{\Sigma \cup \{L, R\}\}^k$. A *configuration* is a k-tuple of strings of the form $\alpha q \beta$ where $q \in Q$, $\alpha, \beta \in \Sigma^*$ the last symbol of β is not B (where B is the special blank symbol), and all the q in all the tuples are the same. If the input is x then the standard initial configuration is formed by assuming that x is on the first tape, the head on the first tape is pointing to the rightmost symbol of x, and on all other tapes the head is at the leftmost symbol of the tape.

It turns out that extra tapes do not increase power.

Theorem 2.2 *If f can be computed by a k-tape Turing Machine then f can be computed by an ordinary Turing Machine.*

A careful analysis of the proof of the above theorem reveals that the 1-tape machine is not that much more inefficient then the equivalent 2-tape machine. In particular, we have actually shown that if the 2-tape machine halts on inputs of length n in $T(n)$ steps, then the 1-tape machine will halt, on inputs of length n, in $T(n)^2$ steps. While this is not important for recursion theory, it will be a significant fact in complexity theory. The best known simulation of a multitape Turing Machine by a fixed number of tape machine is that any function f that can be computed by k-tape Turing Machine in $T(n)$ steps on inputs of length n can be computed by a 2-tape machine in $T(n) \log T(n)$. (See Hopcroft-Ullman, the White book.)

Other enhancements to a Turing Machine such as extra heads, two-dimensionality, allowing a 2-way infinite tape, do not add power. Note that a Turing Machine with many added features resembles an actual computer.
Exercise Discuss informally how to convert various variants of a Turing Machine to a 1-tape 1-head 1-dim Turing Machine. Comment on how runtime and number of states are affected.

3 Godelization

By using variations of Turing Machines it would not be hard to show that standard functions such as addition, multiplication, exponentiation, etc. are all computable by Turing Machines. We wish to examine functions that, in some sense, take Turing Machines as their input. In order to do this, we must code machines by numbers. In this subsection we give an explicit coding and its properties. The actual coding is not that interesting or important and can be skipped, but should at least be skimmed to convince yourself that it really can be carried out. The properties of the coding are very important. A more abstract approach to this material would be to DEFINE a numbering system as having those properties. We DO NOT take this approach, but will discuss it at the end of this section.

Def 3.1 A *Godelization* is an onto mapping from N to the set of all Turing Machines such that given a Turing Machine, one can actually find the number mapped to, and given a number one can actually find the Turing Machine that maps to it.

We define a Godelization. Let $M = (Q, \Sigma, \delta, q_0, h)$ be a Turing Machine. We assume the following:

- there are $n + 1$ states labeled 1,2,3,$\ldots$, $n + 1$,

- state $n+1$ is the halting state,
- the alphabet is the numbers 3,4,5$\ldots, m$.
- L, R are represented by the numbers 1 and 2. (We still denote L and R by L and R. Note that L and R have numbers different from those in the alphabet.)

We first show how to encode a rule as a number:

Let $q_1, q_2 \in Q$ and $\sigma_1, \sigma_2 \in \Sigma$. (By our convention, $q_1, q_2, \sigma_1, \sigma_2$ are numbers). The rule

$$\delta(q_1, \sigma_1) = (q_2, \sigma_2)$$

is represented by the number $2^{q_1}3^{\sigma_1}5^{q_2}7^{\sigma_2}$. The representations for rules that have L or R in the last component are defined similarly. In any case we denote the rule that says what $\delta(q, \sigma)$ does by $c(q, \sigma)$.

We now code the entire machine M as a number. Let p_i denote the ith prime. Let $\langle -, - \rangle$ be such that the map $(i, j) \to \langle i, j \rangle$ is a bijection from $\mathsf{N} \times \mathsf{N}$ to N which is computable by a Turing Machine. The Turing Machine M is coded by the number

$$C(M) = \prod_{i=1}^{n} \prod_{j=1}^{m} p_{\langle i,j \rangle}^{c(i,j)}$$

With our current coding, although all Turing Machines correspond to numbers, not all numbers correspond to Turing Machines. We alleviate this by convention:

Def 3.2 Turing Machine M_i is the machine corresponding to number i, if such a machine exists, and is the machine $(\{q\}, \{a\}, \delta, q, q)$ where $\delta(q, a) = (q, a)$, (i.e. the easiest machine that halts on all inputs) otherwise. The function computed by M_i is denoted φ_i We may also say that i is the index of M_i or φ_i.

It is easy to see that a program could be written to, given a Turing Machine M, find x such that $M = M_x$. (We will later be assuming that a Turing Machine could carry out such a task). It is also easy to see that a program could be written to, given a number x, determine M_x.

The number x codes all the information about M_x that one might wish to know.

Exercise Informally show that the following functions are computable:

- Given x, determine the number of states in M_x.
- Given x, determine the number of symbols in the alphabet of M_x.
- Given numbers x, y, s, determine if M_x halts on y in less than s steps.
- Given numbers x and y, produce the code for the Turing Machine that computes the composition of the functions computed by M_x and M_y.

Our coding has some very nice properties that we now state as theorems. There is nothing inherently good about the coding we used, virtually any coding one might come up with has these properties. The properties essentially say that we can treat the indices of a Turing Machines as though they were programs. We will be using these theorems informally, without explicit reference to them, for most of this course.

Theorem 3.3 *($s-0-0o$ Theorem) There exists a primitive recursive function $s_{1\text{-}1}$ such that for all x, y, and z*

$$M_x(y, z) = M_{s_{1\text{-}1}(x,y)}(z)$$

.

Intuitively this is saying that parameters and code are interchangeable. If JAVA was being used instead of Turing Machines, the $s-0-0$ function would merely be replacing a READ statement with a CONST statement.

The above theorem is better known in its generalized form:

Theorem 3.4 *($s-m-n$ Theorem) For every m and n there exists a primitive recursive function $s_{m\text{-}n}$ such that for all x, $\langle y_1, \ldots, y_{n+1}\rangle$, and $\langle z_1, \ldots, z_m\rangle$*

$$M_x(\langle y_1, \ldots, y_{n+1}\rangle, \langle z_1, \ldots, z_m\rangle) = M_{s_{m-n}(x,\langle y_1,\ldots,y_{n+1}\rangle)}(\langle z_1, \ldots, z_m\rangle)$$

.

Both the $s-0-0$ theorem and the $s-m-n$ theorem are proven by actually constructing such functions. These constructions were of more interest when they were proven than they are now, since now the notion of treating data and parameters the same has been absorbed into our culture.

The next theorem says that there is one Turing Machine that can simulate all others. It is similar to a mainframe: you feed it programs and inputs, and it executes them.

Theorem 3.5 *(Universal Turing Machine Theorem, or Enumeration Theorem) There is a Turing Machine M such that $M(x, y)$ is the result of running M_x on y. (Note that this might diverge.)*

Convention 3.6 We will always denote the Universal Turing Machine by U.

We will be using the $s - m - n$ Theorem and the existence of a Universal Turing Machine, throughout this course (usually implicitly). A more abstract approach would have been to build these two properties into definitions:

Def 3.7 An *acceptable programming system* (henceforth APS) is a list of all the Turing computable functions $\varphi_1, \varphi_2, \ldots$ such that the $s-m-n$ Theorem, and the enumeration theorem are true relative to that numbering.

One concern might be that if we prove theorems for our particular APS will it be true for all APS's. The following theorem says YES, as it says that all APS's are essentially the same.

Theorem 3.8 *(Rogers isomorphism theorem) Let $\varphi_1, \varphi_2, \ldots$ and $\phi_1, \phi_2, \ldots$ be two APS's. There exists a bijection f, computable by a Turing Machine, such that $\varphi_i \equiv \phi_{f(i)}$.*

4 Other Models and the Moral of the Story

Many models of computation have been proposed. All of them have a notion of discrete time steps, as does a Turing Machine.

1. Turing Machines were proposed by Alan Turing in 1936.

2. λ-calculus was proposed by Alonzo Church in 1941. The λ-calculus enables one to speak of functions from sets of functions to sets of functions. The language LISP is based on λ-calculus.

3. post Systems were proposed by Emil Post in 1943. They are a generalization of Grammars.

4. Wang machines were proposed by Hao Wang in 1957.

5. Markov Algorithms were proposed by Andrei Andreivich Markov in the 1940's.

6. register machines were proposed by Abraham Robinson and Calvin Elgot in the 1960's, and Random Access Machines were proposed by Steven Cook and Robert Rechow in the 1970's. Both resemble an actual computer more than most models.

These models of computation had very different motivations. Now for the surprise: THEY ALL COMPUTE THE SAME CLASS OF (PARTIAL) FUNCTIONS! In addition, the time loss in going from one to the other is (in most cases) only a polynomial e.g. if a Markov algorithm can compute a function f and use $T(n)$ steps on inputs of length n, then there is a Turing Machine that can computes f and takes $T(n)^k$ steps on inputs of length n.

We have been trying to formalize what it means for a function to be "intuitively computable." This seems like a hard concept to define rigorously. But several people who tried to formalize this notation came to the SAME class. This leads one to make a leap of faith and conclude that yes indeed, this class of functions suffices:

Church's Thesis: Any (partial) function that is intuitively computable (e.g. we can write down a program for it in some informal language) is computable by a Turing Machine (thus by the λ-calculus, etc.).

For the remainder of this course we will speak in terms of Turing Machines, but will virtually never have to worry about the formal details of a machine. To show a function is computable we will write an informal program that computes it, and show that it works.

We repeat two definitions that we made earlier, noting that in both the term "Turing Machine" can be replaced by any of the above models.

Def 4.1 A function computed by a Turing Machine is a *partial computable function.* If the function is total then we say it is *computable.*

We now give examples of computable functions.

1. $f(x)$ is the xth prime, computable.

2. $f(x) = x^7 + 12x^5$, computable.

3. Ackerman's function is computable.

4. Any JAVA program that halts on all inputs you can think of is computing a computable function.

We give an interesting example of a partial computable function. We want a function that will, on input e, output some PRIME that M_e halts on. If M_e does not halt on any prime, then the function will be undefined.

First attempt (which will fail): run $M_e(2)$. If it halts then output 2, else run $M_e(3)$. If it halts then output 3, else run $M_e(5)$. This will not work since you cannot tell if $M_e(2)$ halts.

So what to do?

Well, we can try to run $M_e(2)$ for a few steps, then try $M_e(3)$ for a few steps, then go back to $M_e(2)$ and try out various other primes as we go. We try $M_e(p)$ for s steps for many primes p and numbers s. This process is known as DOVETAILING. Before presenting the formal algorithm we'll need pairing functions.

Def 4.2 Let π_1 and π_2 be computable function such that the set $\{(\pi_1(x), \pi_2(x)) : x \in \mathsf{N}\}$ is all of $\mathsf{N} \times \mathsf{N}$.

Algorithm for f:

```
1. Input($e$)
2. $i := 1$
   FOUND := FALSE
   While NOT FOUND
       $x := \pi_1(i)$
       $s := \pi_2(i)$
       Run $M_e(x)$ for $s$ steps.
       If $x$ is prime and $M_e(x)$ halts within $s$ steps then
           output($x$)
           FOUND := TRUE
       else $i := i + 1$
```

The algorithm looks at ALL possible pairs (x, s) and if we find that $M_e(x)$ halts in s steps, and x is prime, then we halt. Note that if M_e halts on SOME prime then $f(x)$ will be such a prime; however, if M_e does not halt on any prime, then the algorithm will diverge (as it should).

5 Some strange examples of computable functions

Functions that are almost always 0 are very easy to compute: just store a table.

Example 5.1 Let f be the function that is $f(0) = 12$, $f(10) = 20$, $f(14) = 7$, f is zero elsewhere. The function f is easily seen to be computable. Just write a program with a lot of 'if' statements in it. It will output 0 on values that are not 0,10, or 14.

In the above example, the function f was given EXPLICITLY so it was easy to write the program. Even if a function is not given to us explicitly, we may be able to show that it is computable.

Example 5.2 Let f be the function that is nonzero on values less than 10, and on those values always outputs the input squared. From the description we can deduce that $f(1) = 1$, $f(2) = 4$, $f(3) = 9$, $f(4) = 16$, $f(5) = 25$, $f(6) = 36$, $f(7) = 49$, $f(8) = 64$, $f(9) = 81$, and f is zero elsewhere.

In the above example, even though we were not given the function explicitly, we could derive an explicit description from what was given. In the next example this is no longer the case, but the function is still computable.

INTERESTING EXAMPLE

One needs to know what the Goldbach Conjecture to appreciate this example: Goldbach's conjecture is still unknown. It is: every even is the sum of two primes.

Example 5.3 Let f be the function such that if Goldbach's conjecture is true then f is 74 on all numbers less than 4 and zero elsewhere, and if Goldbach's conjecture is false then f is 17 on all less than 3 and zero elsewhere. Since we don't know whether or not Goldback's Conjecture is true, WE DO NOT KNOW what f is. But we DO know that EITHER

1. $f(1) = 74$, $f(2) = 74$, $f(3) = 74$, $f(x) = 0$ elsewhere, OR
2. $f(1) = 17$, $f(2) = 17$, $f(x) = 0$ elsewhere.

So THERE EXISTS a JAVA program for f. In fact, we can write down two programs, and know that one of them computes f, but we don't know which one. But to show that f is computable WE DO NOT CARE WHICH ONE! The definition of computability only said THERE EXISTS a JAVA program, it didn't say we could find it.

An even more interesting example:

Example 5.4 Let f be the following function: if Goldbach's conjecture is false then f is 888 on the the smallest even n such that n cannot be written as the sum of two primes, and 0 elsewhere. if Goldbach's conjecture is true then f is always 0. If Goldbach's conjecture is false then f is one of the following.

1. $f(2) = 888$, f is zero elsewhere
2. $f(4) = 888$, f is zero elsewhere
3. $f(6) = 888$, f is zero elsewhere
4. etc. $\vdots$

The fact that this list is infinite should not bother us. It is still the case that f is computable since one of the functions on this list is f, or f is always 0.

These functions are computable EVEN THOUGH WE CAN"T FIND CODE FOR THEM.

If I asked you what a computable function was you might say

f is computable if there exists a TURING MACHINE to compute it.

I might say

f is computable if THERE EXISTS a Turing machine to compute it.

The key thing is that THERE EXISTS a Turing machine, even if I can't find it.

AN EXAMPLE OF 'I DO NOT KNOW AND I DO NOT CARE', that is not related to computer science:

Example 5.5 Do there exists two irrational numbers x and y such that x^y is rational? I will show you pairs (a, b), and (c, d) such that either

1. a and b are irrational and a^b is rational, OR
2. c and d are irrational and c^d is rational.

Even at the end of the proof I won't know which pair works.

Let $a = \sqrt{2}$, $b = \sqrt{2}$, $c = (\sqrt{2})^{\sqrt{2}}$, $d = \sqrt{2}$. We already know that $\sqrt{2}$ is irrational. If $(\sqrt{2})^{\sqrt{2}}$ is rational, then (a, b) works. If $(\sqrt{2})^{\sqrt{2}}$ is irrational then c is irrational, d is irrational, and

$$c^d = ((\sqrt{2})^{(\sqrt{2})})^{\sqrt{2}} = (\sqrt{2})^2 = 2$$

so the pair (c, d) works. I DO NOT KNOW whether or not $(\sqrt{2})^{\sqrt{2}}$ is irrational, but in either case, I get what I want, so for now I DO NOT CARE.

6 Computable and Computably Enumerable Sets

Up to this point we have been speaking of functions. Sets are easier to study and more flexible. Most of the rest of the course will be about sets.

Def 6.1 A set A is *computable* if there exists a Turing Machine M that behaves as follows:

$$M(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{if } x \notin A. \end{cases}$$

Computable sets are also called decidable or solvable. A machine such as M above is said to decide A.

Some examples of computable sets.

1. The primes.
2. The Fibaonoacci numbers (any number in the set $1, 2, 3, 5, 8, 13, \ldots$ where every number is the sum of the previous number). If you want to know if a number x is a Fib number, just calculate the Fib numbers until you either spot x or surpass it. If you spot it then its a Fib number, if you surpass it, its not.
3. (x, y, s) such that $M_x(y)$ halts within s steps.

4. Most sets you can think of are computable.

Are there any noncomputable sets? Cheap answer: The number of SETS is uncountable, the number of COMPUTABLE SETS is countable, hence there must be some noncomputable sets. In fact, there are an uncountable number of them. I find this answer rather unenlightening.

7 The HALTING Problem

In this subsection we exhibit a concrete example of a set that is r.e. but not computable. Recall that M_x is the xth Turing Machine in the Godelization defined earlier.

Def 7.1 The HALTING set is the set

$$K_0 = \{\langle x, y\rangle \mid M_x(y) \text{ halts }\}.$$

Let us ponder how we would TRY to determine if a number $\langle x, y\rangle$ is in the halting set. Well, we could try RUNNING M_x on y. If the computation halts, then GOOD, we know that $\langle x, y\rangle \in K_0$. And if it doesn't halt then – WHOOPS– if it never halts we won't know that!! It seems hard to determine with certainty that the machine will NOT halt EVER.

Theorem 7.2 *The set K_0 is not computable.*

Proof:

We show that K_0 is NOT computable, by using diagonalization. Assume that K_0 is computable. Let M be the Turing Machine that decides K_0. Using M we can easily create a machine M' that operates as follows:

$$M'(x) = \begin{cases} 0 & \text{if } M_x(x) \text{ does not halt,} \\ \uparrow & \text{if } M_x(x) \text{ does halt.} \end{cases}$$

Since M' is a Turing Machine, it has a Godel number, say e, so $M_e = M'$. We derive a contradiction by seeing what M_e does on e.

If $M'(e) \downarrow$ then by the definition of M', we know that $M_e(e)$ does not halt, but since $M' = M_e$, we know that $M_e(e)$ does halt. Hence the scenario that $M'(e) \downarrow$ cannot happen. (This is not a contradiction yet)

If $M'(e)\uparrow$ then by the definition of M', we know that $M_e(e)$ does halt'; but since $M' = M_e$, we know that $M_e(e)$ does not halt. Hence the scenario that $M'(e)\uparrow$ cannot happen. (This alone is not a contradiction)

By combining the two above statements we get that $M'(e)$ can neither converge, nor diverge, which is a contradiction. ▮

This proof may look unmotivated— why define M' as we did? We now look at how one might have come up with the halting set if one's goal was to come up with an explicit set that is not decidable:

We want to come up with a set A that is not decidable. So we want that M_1 does not decide A, M_2 does not decide A, etc. Let's make A and machine M_i differ on their value of i. So we can DEFINE A to be

$$A = \{i \mid M_i(i) \neq 1\}.$$

This set can easily be shown undecidable— for any i, M_i fails to decide it since A and M_i will differ on i. But looking at what makes A hard intuitively, we note that the "$\neq 1$" is a red herring, and the set

$$B = \{i \mid M_i(i)\downarrow\}$$

would do just as well. This is essentially the Halting problem.

Corollary 7.3 *The set $K = \{e \mid M_e(e)\downarrow\}$ is undecidable.*

Proof: In the proof of Theorem 7.2, we actually proved that K is undecidable. ▮

Note 7.4 In some texts, the set we denote as K is called the Halting set. We shall later see that these two sets are identical in computational power, so the one you care to dub THE halting problem is not important. We chose the one we did since it seems like a more natural problem. Henceforth, we will be using K as our main workhorse, as you will see in a later section.

8 Computablely Enumerable Sets

K_0 and K are not decidable. Well, what CAN we say about K_0 that is positive. Lets look back at our feeble attempt to solve K_0. The algorithm was: on input x, y, run $M_x(y)$ until it halts. The problem was that if $\langle x, y\rangle \notin K_0$ then the algorithm diverges. But note that if $(x, y) \in K_0$ then this algorithm converges. SO, this algorithm DOES distinguish K_0 from $\overline{K}_0$. But not quite in the way we'd like. The following definition pins this down

Def 8.1 A set A is *computablely enumerable* (henceforth "r.e.") if there exists a Turing Machine M that behaves as follows:

$$M(x) = \begin{cases} \downarrow & \text{if } x \in A, \\ \uparrow & \text{if } x \notin A. \end{cases}$$

Exercise Show that K and K_0 are r.e.
Exercise Show that if A and B are computable then $A \cap B$, $A \cup B$, and $\overline{A}$ are computable. Which of these are true for r.e. sets?

There is a definition of r.e. that is equivalent to the one given, and is more in the spirit of the words "computablely enumerable."

Theorem 8.2 *Let A be any set. The following are equivalent:*

1. *A is the domain of a partial computable function (i.e. A is r.e.)*
2. *A is the range of a total computable function or $A = \emptyset$ (this definition is more like enumerating a set).*

Proof: We show 1) $\rightarrow$ 2) $\rightarrow$ 1).

1) $\rightarrow$ 2): Let A be the domain of a partial computable function f. Let M be a Turing Machine whose domain is A. If A is empty, then 2) is established. Assume that A is nonempty and let $a \in A$. Let g be the (total) computable function computed by the following algorithm:

1. Input(n).
2. If $n = 0$ then output a.
3. Compute $X = \{g(0), g(1), g(2), \ldots, g(n-1)\}$.
4. Let $Y = \{0, 1, 2, \ldots, n\}$. If $Y - X$ is empty then output a. If $Y - X$ is not empty then run M on every element of $Y - X$ for n steps. If there is some $y \in Y - X$ such that $M(y)$ halts within n steps then output the least such y. Else output a.

We show that range(g) =domain(f). If y is in the range of g then it must be the case that $M(y)$ halted, so y is in the domain of f. If y is in the domain of f then let n be the least number such that $M(y)$ halts in n steps and $y \le n$. If there is some $m < n$ such that $g(m) = y$ then we are done. Otherwise consider the computation of $g(n)$. In that computation $y \in Y$ but might not be output if there is some smaller element of Y. The same applies to $g(n+1), g(n+2), \ldots$. If there are z elements smaller than y in A then one of $g(n), g(n+1), \ldots, g(n+z)$ must be y.

2) $\rightarrow$ 1). Assume that A is either empty or the range of a total computable function. If A is empty then A is the domain of the partial computable function that always diverges, and we are done. Assume A is the range of a total computable function f. Let g be the partial computable function computed by the following algorithm:

1. Input(n).
2. Compute $f(0), f(1), \ldots$ until (if it happens) you discover that there is an i such that $f(i) = n$. If this happens then halt. (if it does not, then the function will end up diverging, which is okay by us).

We show that an element n is in the range of f iff $g(n)$ halts. If n is in the range of f then there exists an i such that $f(i) = n$; this i will be discovered in the computation of g on n, so $g(n)$ will be 1. If $g(n)$ halts then an i was discovered such that $f(i) = n$, so n is in the range of f.

▮

Several questions arise at this point:

- Are there any sets that are r.e. but not computable?
- Are there any sets that are NOT r.e.?
- If a set is r.e., then is its complement r.e. ?

The second question can be answered in a cheap way: since there are an uncountable number of sets and a countable number of r.e. sets (since there are only a countable number of Turing Machines), there are an uncountable number non-r.e. sets. While this is true, it is not a satisfying answer. We will give more concrete answers to all these questions.

First we relate r.e. and computable sets.

Theorem 8.3 *A set A is computable iff both A and $\overline{A}$ are r.e.*

Proof: If A is computable then $\overline{A}$ is computable. Since any computable set is r.e. both are r.e.

Assume A and $\overline{A}$ are r.e. Let M_a be a Turing Machine that has domain A and M_b be a Turing Machine that has domain $\overline{A}$. The set A is computable via the following algorithm: on input x run both $M_a(x)$ and $M_b(x)$ simultaneously; if $M_a(x)$ halts then output YES, if $M_b(x)$ halts then output NO. Since either $x \in A$ or $x \in \overline{A}$, one of these two events must happen. ▮

This theorem links two of our questions: there exists an r.e. set that is not computable iff r.e. sets are not closed under complementation.

9 Undecidable sets, m-reductions, and Rice's Theorem

Now that we have K undecidable, we can show that other sets are undecidable as well. Our proofs will be along the lines of "to show that A is undecidable we show that if A were decidable, then so would be K, thus A cannot be undecidable."

We start with an easy one:

Example 9.1 The set

$$A = \{x \mid x - 17 \in K\}$$

is undecidable. Assume that A is decidable via Turing Machine M. Using this, we show that K is decidable.

1. Input(x)
2. Run M on the input $x + 17$. (Output whatever it outputs.)

Note that $x \in K$ iff $x + 17 \in A$, so the algorithm decides K. This is a contradiction, so A is undecidable.

EXERCISE Show that the set

$$B = \{x \mid x + 17 \in K\}$$

is undecidable.

The key part of the proof that A is undecidable is to find a very nice question to ask M, in this case the question $x + 17 \in A$? In most proofs, the hard part is finding the right question to ask, a question whose answer is informative in terms of determining if $x \in K$.

We now give a harder example.

Example 9.2 Show that the set

$$A = \{x \mid \varphi_x(17) \downarrow\}$$

is undecidable. Assume A is decidable via Turing Machine M. We use M in an algorithm to decide K.

1. Input(x)
2. Create a machine M' that does the following (DO NOT RUN THIS MACHINE!!!!!!!! ONLY CREATE IT.)
   (a) Input(z)
   (b) Run $M_x(x)$ (note that we are NOT using the input z here).
3. Find the Godel number y of M' (so φ_y is the function computed by M')
4. Run $M(y)$. (Output whatever it outputs.)

To see that this algorithm decides K note that

$x \in K \rightarrow M_x(x) \downarrow \rightarrow M'$ will halt on all inputs $\rightarrow M'$ will halt on 17 $\rightarrow$ the Godel number of M' is in $A \rightarrow M(y)$ will say YES.

$x \notin K \rightarrow M_x(x) \uparrow \rightarrow M'$ will not halt on any input $\rightarrow M'$ will not halt on 17 $\rightarrow$ the Godel number of M' is not in $A \rightarrow M(y)$ will say NO.

EXERCISE In the above proof, the number 17 was not relevant. (It was a 'red herring' or a 'paper tiger') Name some other sets for which the proof that A is undecidable applies equally well to.

All the proofs that sets are undecidable, in this subsection, have had a very similar flair. We now codify these proofs— that is, define some terms and prove some theorems that will be used in all future proofs without a need for explicit mention.

As mentioned before, the key point is finding the appropriate question (or questions) to run M on to find out things about K. The following definitions and theorems make this notion rigorous:

Def 9.3 Let A and B be any two sets. A is *m-reducible to* B, denoted $A \leq_{\mathrm{m}} B$ if there is a computable function f such that $x \in A$ iff $f(x) \in B$.

The 'm' in the above definition is because f can be many-to-one, as opposed to the following definition:

Def 9.4 Let A and B be any two sets. A is *one-reducible to* B, denoted $A \leq_1 B$ if there is a computable one-to-one function f such that $x \in A$ iff $f(x) \in B$.

We will not be concerned with whether our reductions are 1-1 except in an occasional exercise.

EXERCISE Show that if $A \leq_{\mathrm{m}} B$ then $\overline{A} \leq_{\mathrm{m}} \overline{B}$. Show that if $A \leq_1 B$ then $\overline{A} \leq_1 \overline{B}$.

Theorem 9.5 *If A and B are sets, B is computable, and $A \leq_{\mathrm{m}} B$, then A is computable.*

Proof: Let f be computed by M and B be decided by N. The following algorithm decides A

1. Input(x)
2. Run M on x and call the result y (note that y is $f(x)$).
3. Run N on y. (Output whatever it outputs.)

$x \in A \rightarrow f(x) \in B \rightarrow N(y)$ will say YES.
$x \notin A \rightarrow f(x) \notin B \rightarrow N(y)$ will say NO. ∎

Corollary 9.6 *If $K \le_{\rm m} A$ then A is undecidable.*

Proof: By the above theorem, if A is decidable then K is decidable. This is a contradiction. ▮

Corollary 9.7 *If $\overline{K} \le_{\rm m} A$ then A is undecidable.*

Proof: Similar. ▮

We will use Corollary 9.6 implicitly for the rest of these notes: one way to show a set is undecidable will be to exhibit an algorithm for a reduction f that reduces K to that set. All sets shown undecidable in this subsection can be recast in that form. We do that:

Example 9.8 We show that the set

$$A = \{e \mid \varphi_e(17) \downarrow\}$$

is undecidable. We show $K \le_{\rm m} A$.

1. Input(x)
2. Create a machine M' that does the following (DO NOT RUN THIS MACHINE!!!!!!!! ONLY CREATE IT.)
   (a) Input(z)
   (b) Run $M_x(x)$
3. Find the Godel number y of M' and output it.

$x \in K \rightarrow M_x(x) \downarrow \rightarrow M'$ will halt on all inputs, including $17 \rightarrow$ the Godel number of M' is in A

$x \notin K \rightarrow M_x(x) \uparrow \rightarrow M'$ will not halt on any inputs, including $17 \rightarrow$ the Godel number of M' is not in A

(MINOR comment— the above proof uses Church's thesis in that we are assuming that whatever we can write down and describe informally can be carried out by a Turing Machine. For this proof, and most proofs in this course, all we really need are the s-m-n Theorem (Theorem 3.4) and the existence of a universal Turing Machine (Theorem 3.5).)

We now give MANY examples of sets that are undecidable.

Example 9.9 The set

$$A = \{e \mid \varphi_e \text{ is total }\}$$

is undecidable. The following algorithm computes an m-reduction f of K to A.

1. Input(x)
2. Create a machine M' that does the following:
    (a) Input(z)
    (b) Run $M_x(x)$
3. Output the index of M'.

The reasoning is left as an exercise.

Example 9.10 The set

$$A = \{e \mid \varphi_e \text{ computes the square function }\}$$

is undecidable. The following algorithm computes an m-reduction f of K to A.

1. Input(x)
2. Create a machine M' that does the following:
    (a) Input(z)
    (b) Run $M_x(x)$
    (c) (Note that this step will not be reached unless $M_x(x)$ halts) Compute z^2 and output it.
3. Output the index of M'.

$x \in K \to$ for all z the computation of $M'(z)$ will complete the computation of $M_x(x)$ and then compute z^2, so it will always output $z^2 \to$ the index of M' is in A.

$x \notin K \to$ for all z the computation of $M'(z)$ will diverge while it is trying to compute $M_x(x) \to M'$ diverges on all inputs $\to$ the index of M' is NOT in A.

Example 9.11 The set

$$A = \{e \mid \text{ the domain of } \varphi_e \text{ is a noncomputable set } \}$$

is undecidable. The following algorithm computes an m-reduction f of K to A.

1. Input(x)
2. Create a machine M' that does the following:
   a) Input(z)
   b) Run $M_x(x)$
   c) (Note that this step will not be reached unless $M_x(x)$ halts) Run $M_z(z)$
3. Output the Godel number of M'

$x \in K \to$ for all z, the computation of $M'(z)$ will finish step b, and then proceed to run $M_z(z)$, hence the domain of M' is $K \to$ domain(M') is noncomputable $\to$ Godel number of M' is in A.

$x \notin K \to$ for all z, the computation of $M'(z)$ will not finish step b hence the domain of M' is $\emptyset \to$ domain(M') is computable $\to$ Godel number of M' is not in A.

All of these proofs look the same, so the question arises: can we prove one GENERAL theorem of which these will all be corollaries? Is there some property of these sets that we can exploit and generalize? The answer is YES!

Let A be one of the sets above that is proven undecidable. The question "Is 75 in A?" depends only on φ_{75} and not on any other property of 75.

Assume that φ_{75} and φ_{197} are the same partial function (e.g., both φ_{75} and φ_{197} are the square function, or both φ_{75} and φ_{197} output 1 on elements of K and diverge otherwise). As far as A is concerned 75 and 197 will have the SAME status. Since the set A is only concerned with a number in so far as it represents a partial computable function, if two numbers represent the same partial computable function then those two numbers will either both be IN A or both be OUT of A.

Def 9.12 An *index set* is a set A such that for all x, y if φ_x and φ_y both compute the same function then either $x, y \in A$ or $x, y \notin A$.

Now for the big theorem:

Theorem 9.13 *(Rice's Theorem) If A is any index set such that $A \neq \emptyset$ and $A \neq \mathsf{N}$ then A is noncomputable.*

Proof: Let A be an index set, $A \neq \emptyset$, $A \neq \mathsf{N}$. Let n be such that φ_n is the function that diverges on all inputs. There are two cases:
Case 1: $n \notin A$. Let a be some element of A (such exists since $A \neq \emptyset$). We use a and n in the following reduction of K to A.

1. Input(x)
2. Create a machine M' that does the following:
   (a) Input(z)
   (b) Run $M_x(x)$.
   (c) (This step will only be reached if $M_x(x)$ halts.) Run $M_a(z)$. Output whatever it outputs.
3. Output the Godel Number of M'

If $M_x(x)$ halts then M' will always get to execute the third step, so $M'(z) = M_a(z)$. Let b be the Godel number of M'. Note that φ_b and φ_a compute the same function. Since A is an index set and $a \in A$, $b \in A$.

If $M_x(x)$ does not halt then M' will diverge on all inputs. Let b be the Godel number of M'. Note that φ_b and φ_n compute the same function. Since A is an index set and $n \notin A$, $b \notin A$.
Case 2: Assume $n \in A$. The set $\overline{A}$ is an index set and $n \notin \overline{A}$. Hence Case 1 applied to $\overline{A}$ shows that $\overline{A}$ is not computable, which implies that A is not computable. ∎

10 Turing Reductions

In the last section we showed that sets A were undecidable by showing that if A was decidable, then by using a program for A we could decide K. But we only used that program in a limited way. We only called it once.

We want a more powerful type of reduction. For example, informally it is clear that $A \times \overline{A} \leq A$ under some type of reduction.

Def 10.1 (Informal) Let A be any set. A set B is computable in A if there is a Turing Machine that, together with a "subroutine" for A can decide B. The set A is called an oracle. We denote the fact that B is computable in A by $B \leq_T A$, and say that "B is Turing-less than A"

It is easy to see that for all sets A, $A \times \overline{A} \leq_{\mathrm{T}} A$.

We will not deal with Turing reductions in this course except in an occasional exercise. Professional recursion theorists deal mostly with Turing reductions.

11 Non-r.e. sets and the Extended Rice's Theorem

We want a general tool to show that index sets are not r.e. We already know that $\overline{K}$ is not r.e. and will use that.

The line of reasoning in that corollary generalizes: EXERCISE Show that if A is a noncomputable set that is r.e. then $\overline{A}$ is not r.e.

We now look at a non r.e. set that is proven non-r.e. in a different way.

Theorem 11.1 *Let*

$$TOT = \{e \mid \varphi_e \textit{ is total }\}.$$

The set TOT is not r.e.

Proof: Assume, by way of contradiction, that TOT is r.e. Let f be a total computable function such that TOT is the range of f. Consider the following function

$$g(x) = \varphi_{f(x)}(x) + 1.$$

It is easy to see that $g(x)$ is total computable. Therefore there is some i such that g is $\varphi_{f(i)}$. BUT note that g and $\varphi_{f(i)}$ differ on the value i, namely

$$g(i) = \varphi_{f(i)}(i) + 1 \neq \varphi_{f(i)}.$$

This is a contradiction, hence no such f exists and TOT is not r.e. ▮

The technique used in the above theorem is diagonalization. It was also used to show that the reals are uncountable, and that K is noncomputable. It will be used later as well, but it is not that useful for showing sets other than TOT to be non r.e.

The following theorem is very useful in showing that sets are not r.e., and is needed to prove the extended Rice's theorem.

Theorem 11.2 *If $B \leq_{\mathrm{m}} A$ and A is r.e., then B is r.e.*

Proof: Let f be such that $B \leq_{\mathrm{m}} A$ via f. Let g be a partial computable function such that A is the domain of g. The following is a partial computable function whose domain is B.

1. Input(x)
2. Compute $g(f(x))$.

   $x \in B$ iff $f(x) \in A$ iff $g(f(x)) \downarrow$. ▮

Theorem 11.3 *If A is a nontrivial index set and the index for the function that always diverges is in A, then A is not r.e.*

Proof: By examining the proof of Rice's Theorem (Theorem 9.13) note that if B is an index set that does not contains the index for the null function, then $K \leq_{\mathrm{m}} B$. Hence $K \leq_{\mathrm{m}} \overline{A}$, so $\overline{K} \leq_{\mathrm{m}} A$, and A is not r.e. ▮

The extended Rice's theorem has three parts. The first two are of the form "If A is an index set and blah-de-blah then A is NOT r.e." and the third is of the form "If A is an index set then A is r.e. iff blahblahblah." We first give two proofs of sets not being r.e. to give the flavor of the first two theorems.

Example 11.4 Let

$$A = \{e \mid \varphi_e \text{is undefined on some even number(s) }\}.$$

We show that A is not r.e. by showing that $\overline{K} \le_{\rm m} A$.

1. Input(x)
2. Create a machine M' that behaves as follows
   (a) Input(z)
   (b) Run $M_x(x)$
3. Output the index for M'

We show that this works:

If $x \in \overline{K}$ then $M_x(x)$ diverges, so M' diverges on all inputs, hence it diverges on some (in fact all) even inputs so the index of M', $f(x)$, is in A.

If $x \notin \overline{K}$ then $M_x(x)$ halts, so M' halts on all inputs, hence it halts on all even inputs, so the index of M', $f(x)$, is not in A.

Example 11.5 Let

$$B = \{e \mid \varphi_e \text{is defined on all of the even numbers }\}.$$

We show that B is not r.e. by showing that $\overline{K} \le_{\rm m} B$.

1. Input(x)
2. Create a machine M' that behaves as follows:
   (a) Input(z)
   (b) Run $M_x(x)$ for $|z|$ steps. (this is the length of the string z.) If it halts within that time then diverge, else converge.
3. Output the index for M'

We show that this works:

If $x \in \overline{K}$ then $M_x(x)$ diverges. For all z, $M'(z)$ converges, so the function computed by M' is total, and thus halts on all the evens. Therefore the index of M' is in B.

If $x \notin \overline{K}$ then $M_x(x)$ converges. Assume $M_x(x)$ converges in t steps. For all z such that $|z| \geq t$ $M'(z)$ diverges. Hence the index of M' is not in B.

Note that the last two examples each show a set that is not r.e., and for which the complement is not r.e.

Def 11.6 Let f and g be two partial functions. If domain(f) $\subseteq$domain(g) and for all $x \in$domain(f) $f(x) = g(x)$ then g *is an extension of* f, and f *is a contraction of* g. If f has finite domain then f *is a finite contraction of* g.

Theorem 11.7 *If A is an r.e. index set then A is closed under extensions, i.e. if $c \in A$ and d is such that φ_d extends φ_c, then $d \in A$.*

Proof: We show this by showing its contrapositive, i.e. we show that if B is an index set that is NOT closed under extensions, then B is NOT r.e. Let B be an index set such that $c \in B$, $d \notin B$ and φ_d extends φ_c. We use c, and d in the following reduction f of $\overline{K}$ to B.

1. Input(x)
2. Create a machine M' that does the following:
    (a) Input(z)
    (b) By dovetailing run $M_x(x)$ and $M_c(z)$ at the same time. If $M_c(z)$ halts, then output what it outputs and HALT. If $M_x(x)$ halts, then go to the next step. (If neither halts then this computation will end up diverging).
    (c) (You only reach this step if $M_x(x)$ halted) Compute $\varphi_d(z)$.
3. Output the index for M'

We show that this works:

If $x \in \overline{K}$ then $M_x(x)$ does not halt. Hence, on any input z the computation of $M'(z)$ will, in step 2, always behave like $M_c(z)$. Hence M' will compute φ_c. Since B is an index set and $c \in B$, the index of M' is in B. Hence $f(x)$ is in B.

If $x \notin \overline{K}$ then $x \in K$ and $M_x(x)$ halts. Look at what M' will do on input z in step 2. There are several cases, but they all end up computing $M_d(z)$:

1. $M_c(z)$ converges and the computation of $M_c(z)$ finishes before the computation of $M_x(x)$, hence $M_c(z)$ is output. Since M_d is an extension of M_c, the value output is $M_d(z)$.
2. $M_c(z)$ converges, but the computation of $M_x(x)$ finishes before the computation of $M_c(z)$. Step 3 will be reached, and the results will be $M_d(z)$.
3. $M_c(z)$ diverges. The computation of $M_x(x)$ will finish and step 3 will be reached, hence $M_d(z)$ will be output.

In any case $M_d(z)$ is output. Hence M' computes φ_d, and since $d \notin B$, and B is an index set, the index of M' is not in B. Therefore $f(x)$ is not in B.

Combining all this reasoning we get

$$x \in \overline{K} \iff f(x) \in B.$$

▮

Theorem 11.8 *If A is an r.e. index set then every element of A "has a finite reason for being in A" i.e. if $c \in A$ then there exists $d \in A$ such that φ_d has a finite domain and φ_c extends φ_d.*

Proof: We show this by showing its contrapositive, i.e. we show that if B is an index set that does NOT have this property, then B is NOT r.e. Let B be an index set such that $c \in B$ but no finite contraction of φ_c is in B. We use c in the following reduction f from $\overline{K}$ to B.

1. Input(x)
2. Create a machine M' that does the following::

    (a) Input(z)

    (b) Run $M_x(x)$ for $|z|$ steps. If it halts within that many steps, then diverge.

    (c) (You only reach this step if $M_x(x)$ has not halted within $|z|$ steps.) Compute $\varphi_c(z)$.

3. Output the index for M'

We show that this works:

If $x \in \overline{K}$ then $M_x(x)$ does not halt. Hence, on any input z the computation of $M'(z)$ will get to step 3, and behave like $M_c(z)$. Hence M' will compute φ_c. Since B is an index set and $c \in B$, the index of M' is in B. Hence $f(x)$ is in B.

If $x \notin \overline{K}$ then $x \in K$ and $M_x(x)$ halts. Assume it halts in t steps. For all inputs z, $|z| \geq t$, $M'(z)$ will diverge. Hence M' has finite domain. Since the function computed by M' is a finite contraction of φ_c, its index is not in B. Therefore $f(x) \notin B$.

Combining all this reasoning we get

$$x \in \overline{K} \iff f(x) \in B.$$

▮

These two theorems are powerful and will suffice to prove that most non-r.e. index sets are indeed not r.e. But we still lack a exact characterization of r.e. index sets. The original Rice's theorem (Theorem 9.13) can be restated as:

If A is an index set then

$$A \text{ is computable iff } (A = \emptyset \text{ or } A = \mathsf{N}).$$

We seek a similar theorem.

Lets look at some index sets that are r.e. and see what is true of all of them. EXERCISE Show that the following sets are r.e.

1. $A = \{e \mid \varphi_e \text{ is defined on 2,3 or 89}\}$

2. $B = \{e \mid \varphi_e \text{ is defined on some prime}\}$

3. $C = \{e \mid \varphi_e$ is defined on some element of K $\}$

4. $D = \{e \mid \varphi_e$ is defined on at least 47 elements of K $\}$

What do all the sets have in common? Well, if $e \in A$ (B, C, D), then there is a finite contraction of φ_e which is in A (B, C, D). BUT also, ANY extension of that finite contraction is in A (B, C, D). The set of finite contractions is itself somewhat nicely behaved. We need a definition to pin down what it means for the finite contractions to be well behaved.

Def 11.9 Recall that $\langle -, - \rangle$ is a computable bijection from $\mathsf{N} \times \mathsf{N}$ onto N. Let n be a natural number. The finite-domain function f_n is

$$f_n(x) = \begin{cases} y & \text{if the } \langle x, y \rangle \text{th bit of } n \text{ is } 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The mapping $n \rightarrow f_n$ is called a *canonical indexing of finite functions.* Note that not all the f_n will be functions— some may have more than one y corresponding to some x.

The representation of a finite function by a f_n is very useful since from n we can determine all information about the function. This is NOT the case if a finite function is represented by one of its Godel numbers. EXERCISE Show that there is no partial computable function f that behaves in any of the following manners:

1. $f(e)$ returns the set of ordered pairs that make up φ_e if φ_e is a finite function.

2. $f(e)$ returns the size of the domain of φ_e if φ_e is finite.

So having a canonical index of a finite function is far more informative than just having a Godel number.

We are now ready to state and prove the real extended Rice's theorem, which is an exact characterization of r.e. index sets.

Theorem 11.10 *Let A be an index set. A is an r.e. index set iff there exists an r.e. set B such that*

$$A = \{e \mid (\exists n \in B)[\varphi_e \textit{ is an extension of } fi_n]\}$$

Proof: Assume that A is an r.e. index set. Assume that M is a Turing Machine whose domain is A. We first show that the set of canonical indices of finite functions in A is r.e., i.e. the set

$$B = \{n \mid (\exists e)[\varphi_e = f_n \text{ and } e \in A]\}$$

is r.e.

The following algorithm halts on the canonical indices for finite functions in A.

1. Input(n)

2. Build a Turing Machine M_e that computes f_n.

3. Run $M(e)$

If f_n is a function whose Godel number is in A, then since A is an index set, all Godel numbers of f_n will be in A. Hence the number e will be in A, so $M(e)$ will halt.

If f_n is a function whose Godel number is not in A, then since A is an index set, all Godel numbers of f_n will not be in A. Hence the number e will not be in A, so $M(e)$ will not halt.

Hence the set B is r.e. We now show that A is the set of all indices of partial computable functions which have a finite contraction in B.

If $e \in A$ then by Theorem 11.8 there exists d such that φ_d is a finite contraction of φ_e and $d \in A$. Since B is the set of canonical indices of ALL finite functions in A, there exists an $n \in B$ such that $f_n = \varphi_d$.

If e is such that there exists an $n \in B$, f_n is a finite contraction of φ_e, then there exists a Godel number d such that $f_n = \varphi_d$ and $d \in A$. Since φ_e extends φ_d, and $d \in A$, by Theorem 11.7, $e \in A$.

Thus it is proven.

We now need to prove the converse: if A is of that weird form, then A is r.e. Assume that A is such that the set B defined in the statement of this theorem is r.e. Then A is r.e. by the following algorithm that halts only on elements of A:

1. Input(e)

2. Search (by dovetailing) for an $n \in B$ and a time s such that f_n is a contraction of $\varphi_{e,s}$. If such is found, then halt.

This only halts on elements of A: if $e \in A$ then such an n exists and will be found by the definition of B; if $e \notin A$ then no such n can exist, and the algorithm will diverge while trying to look for it. ▮

12 The Recursion Theorem

Intuitively, the recursion theorem says that a program can use its own index as a parameter. As an example, (to be shown rigorously later) there is a Turing machine M_e such that the only input that M_e halts on is e itself. Offhand there is no obvious way to construct such a machine, since you do not know the index of a machine until it is built, and it seems to be the case that you need to know the index in order to build it.

Theorem 12.1 *Let $\varphi_1, \varphi_2, \varphi_3, \ldots$ be an APS. For any total computable function t there exists a number a such that*

$$(\forall x)\varphi_a(x) = \varphi_{t(a)}(x).$$

(INTUITION: Let's say you write a JAVA program that has at the top the statement "CONST= 0". Call this program $\varphi_{t(0)}$. If I come along and replace that 0 with a 17, we will call that program $\varphi_{t(17)}$. More generally, if that 0 is replaced by b, it is program $\varphi_{t(b)}$. What our theorem says is that there is some number a such that if you use a for the parameter then the program produced is actually program a. The resulting program can be said to "Know its own index.")

Proof:

Let a_{ij} be the INDEX of the following function:

1. Input(x)
2. Run $\varphi_i(j)$ (this might not halt)
3. (If you got to this step then $\varphi_i(j)$ halted.) Let $e = \varphi_i(j)$. Run $M_e(x)$.

Note that the function that takes (i, j) to a_{ij} is computable. All it does is produce the INDEX of the above code. Hence the function that maps i to a_{ii} is also computable. Since t is total computable, the function that takes i to $t(a_{ii})$ is computable.

Let e be such that $M_e(i) = t(a_{ii})$.

For any j, $M_{a_{ej}}$ computes the same function as $M_{t(a_{jj})}$.

Let $j = e$ to obtain that $M_{a_{jj}}$ and $M_{t(a_{jj})}$ compute the same function. ∎

Example 12.2 We will show that there is a partial computable function φ_a such that the domain of φ_a is just $\{a\}$. Let t be the computable function such that

$$\varphi_{t(a)}(x) = \begin{cases} 1 & \text{if } x = a \\ \uparrow & \text{otherwise} \end{cases}$$

There exists an a such that φ_a is $\varphi_{t(a)}$. It is easy to see that the domain of φ_a is just $\{a\}$.

EXERCISE Show that for any computable function f there exists a computable function g such that $g(0)$ is an index for f, and for all $x > 0$ $g(x) = f(x)$.

The recursion theorem is an important theorem in many branches of recursion theory. It is the bread-and-butter of Inductive Inference. Note that its proof did not use the HALTING problem. In fact, the undecidability of HALT, and Rice's theorem can be derived FROM the Recursion Theorem. APS's can be defined in terms of it, that is, if a programming system has an s-m-n theorem and a Recursion theorem, then it is an APS.

Both Rogers text and Soare's text on Recursion Theory treat the Recursion Theorem as a fixed point theorem. John Case (and his students-including our own Carl Smith) use it in the intuition described here, that is, as a program that knows its own index. Both views are equivalent, but the view that a program can know its own index has more intuitive appeal. In as much as Recursion theory can have controversy, this is a controversial topic in Recursion Theory.

12.1 Undecidable problems that are NOT based on Turing Machines

All the undecidable problems encountered so far have been sets or functions that deal with Turing machines. The question arises, are there any "NATU-

RAL" undecidable problems. One could argue that HALT actually is natural, but we seek problems that do not mention Turing machines.

There are some such problems. The proofs that they are unsolvable usually entail showing that a Turing machine computation can be coded into them. However they are, on the face of it, natural. We list them but do not give proofs.

1. POST'S CORRESPONDENCE PROBLEM. Let Σ be a finite alphabet.

   INPUT: $A = \{\alpha_1, \ldots, \alpha_n\}$, $B = \{\beta_1, \ldots, \beta_n\}$ where $\alpha_i, \beta_j \in \Sigma^*$.

   OUTPUT: YES if there is a word w and an integer m such that w can be formed out of m symbols of A (repeats allowed), and also out of m symbols in B (repeats allowed). NO otherwise. (Formally we say YES if there exists $m, i_1, \ldots, i_m, j_1, \ldots j_m$ such that

   $$\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_m} = \beta_{i_1}\beta_{i_2}\cdots\beta_{i_m}$$

2. HILBERT'S TENTH PROBLEM. Given a polynomial in many variables $p(x_1, \ldots, x_n)$ with integer coefficients, does there exist integers $a_1, \ldots, a_n$ such that $p(a_1, \ldots, a_n) = 0$.

3. HILBERT'S TENTH PROBLEM (improvements) Given a polynomial in just 13 variables $p(x_1, \ldots, x_{13})$ with integer coefficients, does there exist integers $a_1, \ldots, a_{13}$ such that $p(a_1, \ldots, a_{13}) = 0$.

4. CFG UNIV. Given a Context Free Grammar, does it generate EVERYTHING.

5. CFG EQUIV. Given two Context Free Grammars, do they generate the same set?

6. CFG NON-EMPTYNESS. Given a Context Free Grammar, does it generate any strings?

7. OPTIMAL DPDA PROBLEM Given a Push Down Automata for a language, and promised that the language is actually recognizable by a deterministic Push Down Automata, find the size of the smallest Deterministic Push Down Automaton that will recognize it.

8. WORD PROBLEM FOR GROUPS (If you do not understand this problem do not worry, the point is that there are unsolvable problems in a branch of math called Group theory, which is NOT a branch of Logic or Recursion Theory.) Given a group by generators and relations, and then given a word, is it the identity?

9. TRIANGULATION PROBLEM FOR MANIFOLDS (Even if you do not understand this problem the point is that there are undecidable problems in Geometry.) Given two triangulations of four-dimensional manifolds, are those manifolds homeomorphic?

13 Sets that are even harder than HALT

Are there sets that are even "harder to decide" then HALT? We first say what this means formally:

Def 13.1 If $A \le_{\mathrm{T}} B$, but $B \not\le_{\mathrm{T}} A$, then B is *harder than* A.

In this section we exhibit sets that are harder than K but do not prove this.

Recall that K can be written as

$$K = \{e \mid (\exists s) M_e(e) \text{ halts in } s \text{ steps } \}.$$

Note that we have one quantifier followed by a COMPUTABLE statement.

How can TOT be written:

$$TOT = \{e \mid (\forall x)(\exists s) M_e(x) \text{ halts in } s \text{ steps}\}.$$

This is two quantifiers followed by a computable statements.

It turns out that TOT cannot be written with only one quantifier and is harder than K. We can classify sets in terms of how many quantifiers it takes to describe them. Adjacent quantifiers of the same type can always be collapsed into one quantifier.

Def 13.2 Σ_n is the class of all sets A that can be written as

$$A = \{x \mid (\exists y_1)(\forall y_2) \cdots (Q y_n) R(x, y_1, y_2, \ldots, y_n)\},$$

where R is a computable relation and Q is $\exists$ if i is odd, and $\forall$ if i is even.

Def 13.3 Π_n is the class of all sets A that can be written as

$$A = \{x \mid (\forall y_1)(\exists y_2)\cdots(Qy_n)R(x, y_1, y_2, \ldots, y_n)\},$$

where R is a computable relation and Q is $\forall$ is i is odd, and $\exists$ if i is even.

Def 13.4 A set is Σ_n-complete if $A \in \Sigma_n$ and for all sets $B \in \Sigma_n$, $B \leq_{\mathrm{m}} A$.

We now state a theorem without proof.

Theorem 13.5 *For every i there are sets in $\Sigma_i - \Pi_i$, there are sets in $\Sigma_{i+1} - \Sigma_i$, there are Σ_i-complete sets, and there are Π_i-complete sets.*

Exercise (You may use the above Theorem.) Show that a Σ_i-complete set cannot be in Π_i.
Exercise Show that K is Σ_1-complete. Show that $\overline{K}$ is Π_1-complete.
Exercise Show that if A is Π_i-complete then $\overline{A}$ is Σ_i-complete.

We show that FIN (the set of indices of Turing machines with finite domain) is in Σ_2 and that COF (the set of Turing machines with cofinite domains) is in Σ_3. It turns out that FIN is Σ_2-complete, and COF is Σ_3-complete, though we will not prove this. As a general heuristic, whatever you can get a set to be, it will probably be complete there.

$$FIN = \{e \mid (\exists x)(\forall y, s)[\text{ If } y > x \text{ then } M_{e,s}(y) \uparrow\}$$

$$COF = \{e \mid (\exists x)(\forall y)(\exists s)[\text{ If } y > x \text{ then } M_{e,s}(y) \downarrow\}$$