Part II
Computability

6 Models of Computation

What is a computable function? Our modern intuition is that a function $f : \mathbb{N} \to \mathbb{N}$ is computable if there is a a program in a computer language like C^{++}, PASCAL or LISP such that if we had an idealized computer, with unlimited memory, and we ran the program on input n it would eventually halt and output $f(n)$.

While this definition could be made precise, it is rather unwieldy to try to analyze a complex modern programming language and an idealized computer. We begin this section by presenting two possible candidates for the class of computable functions. The first will be based on register machines, a very primitive version of a computer. The second class will be defined more mathematically. We will then prove that these two classes of functions are the same. *Church's Thesis* will assert that this class of functions is exactly the class of all computable functions.

Church's Thesis should be though of as a statement of philosophy or physics rather than a mathematical conjecture. It asserts that the definitions we have given completely capture our intuition of what can be computed. There is a great deal of evidence for Church's Thesis. In particular, there is no know notion of deterministic computation, including C^{++}-computable or the modern notion of quantum computable, that gives rise to computable functions that are not included in our simple classes. [1]

Register Machines

We will take register machines as our basic model of computations. The programming language for register machines will be a simple type of assembly language. This choice is something of a compromise. Register machines are not as simple as Turing machines, but they are much easier to program. It

[1]I do not include notions of random computations or analog computations. But our model could easily be extended to take these into account.

On the other hand issues like time and space complexity or feasability may vary as we change the model.

is not as easy to write complicated programs as it would be in a modern programming language like C^{++} or PASCAL, but it will be much easier to analyze the basic steps of a computation.

In our model of computation we have infinitely many registers $R_1, R_2, \ldots$. At any stage of the computation register R_i will store a nonnegative integer r_i.

Definition 6.1 A *register machine program* is a finite sequence $I_1, \ldots, I_n$ where each I_j is one of the following:

i) Z(n): set R_n to zero; $r_n \leftarrow 0$;

ii) S(n): increment R_n by one; $r_n \leftarrow r_n + 1$;

iii) T(n,m): transfer contents of R_n to R_m; $r_m \leftarrow r_n$;

iv) J(n,m,s), where $1 \leq s \leq n$: if $r_n = r_m$, then go to I_s otherwise go to the next instruction;

v) HALT

and I_n is HALT.

A register machine must be provided with both a program and an initial configuration of the registers. A *computation* procedes by sequentially following the instructions. Note that for any program P there is a number N such, no matter what the initial configuration of the registers is, any computation with P will use at most registers $R_1, \ldots, R_N$.

Example 6.2 *We give a program which, if we start with n in R_1, ends with R_1 containing $n - 1$ if $n > 0$ and 0 if $n = 0$.*

```
1)   Z(2)
2)   J(1,2,10)
3)   Z(3)
4)   S(2)
5)   J(1,2,9)
6)   S(2)
7)   S(3)
8)   J(1,1,4)
9)   T(3,1)
10)  HALT
```

We first test to see if R_1 contains 0. If it does we halt. If not, we make $r_2 = 1$ and $r_3 = 0$ and test to see if $r_1 = r_2$ have the same contents. If they do, we move r_3 to R_1 and halt. Otherwise, we increment r_2 and r_3 until $r_1 = r_2$. Since r_3 will always be one less than r_2, this produces the desired result.

Example 6.3 *We give a program which if adds the contents of R_1 and R_2 and leaves the sum in R_1.*

We set $r_3 \leftarrow 0$. We increment R_3 and R_1 until $r_3 = r_2$.

```
1) Z(3)
2) J(2,3,6)
3) S(1)
4) S(3)
5) J(1,1,2)
6) HALT
```

Example 6.4 *We give a program to multiply the contents of R_1 and R_2 and leave the product in R_1.*

The main idea is that we will add r_1 to itself r_2 times using R_3 to store the intermediate results. R_4 will be a counter to tell us how many times we have already added r_1 to itself. We add r_1 to r_3 by incrementing R_3, r_1 times. We use R_5 to count how many times we have incremented R_3.

```
1)  Z(3)
2)  Z(4)
3)  J(2,4,10)
4)  Z(5)
5)  J(1,5,9)
6)  S(3)
7)  S(5)
8)  J(1,1,5)
9)  S(4)
10) T(3,1)
11) HALT
```

Note that lines 4)–8) are just a slightly modified version of Example 6.3. We add r_1 to r_3 storing the result in R_3. We can think of this as a "subroutine". It is easy to see that we could add a command to our language A(n,m,s) that does:

$$r_s \leftarrow r_n + r_m.$$

Any program written with this additional command could be rewritten in our original language. Similarly, using Example 6.2 we could add a command D(n) that decrements R_n if $r_n > 0$ and leaves R_n unchange if $r_n = 0$.

We give one more example of a program that, on some initial configurations, runs for ever.

Example 6.5 *We give a program such that if r_1 is even halts with $r_1/2$ in R_1 and otherwise never halts.*

```
1)  Z(2)
2)  Z(3)
3)  J(1,2,8)
4)  S(2)
5)  S(2)
6)  S(3)
7)  J(1,1,2)
8)  T(3,1)
9)  HALT
```

We next define what it means for a function $f : \mathbb{N}^k \to \mathbb{N}$ to be computable by a register machine. The last example shows that we need to take partial functions into account.

Suppose P is a register machine program. If $\overline{x} = (x_1, \ldots, x_k)$ we consider the computation where we begin with initial configuration $r_1 = x_1, \ldots, r_k = x_k$ and $r_n = 0$ for $n > k$. If this computation halts we say that P halts on input $\overline{x}$.

Definition 6.6 Suppose $A \subseteq \mathbb{N}^k$. We say $f : A \to \mathbb{N}$ is an *RM-computable* partial function if there is a register machine program P such that:
i) if $\overline{x} \notin A$, then P does not halt on input $\overline{x}$;
ii) if $\overline{x} \in A$, then P halts on input $\overline{x}$ with $f(\overline{x})$ in register R_1.

We could start showing more and more functions are RM-computable by writing more complicated programs. Instead we will give mathematically define an interesting class of fuctions and prove it is exactly the class of RM-computable functions.

Primitive Recursive Functions

Definition 6.7 The class of *primitive recursive functions* is the smallest class C of functions such that:
i) the zero function, $z(x) = 0$ is in C,
ii) the sucessor function $s(x) = x + 1$ is in C,
iii) for all n and all $i \leq n$ the projection function $\pi_i^n(x_1 \ldots x_n) = x_i$, is in C (in particular the identity function on $\mathbb{N}$ is in C),

iv) (Composition Rule) If $g_1 \dots g_m, h \in C$, where $g_i : \mathbb{N}^n \to \mathbb{N}$ and $h : \mathbb{N}^m \to \mathbb{N}$, then

$$f(\overline{x}) = h(g_1(\overline{x}) \dots g_m(\overline{x}))$$

is in C,

v) (Primitive Recursion) If $g, h \in C$ where $g : \mathbb{N}^{n-1} \to \mathbb{N}$ and $h : \mathbb{N}^{n+1} \to \mathbb{N}$, then $f \in C$ where:

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

We now give a large number of examples of primitive recursive functions with derivations showing that they are primitive recursive. A derivation is a sequence of functions $f_1 \dots f_m$ such that each f_i is either z, s or π_i^n or is obtained from earlier functions by compostion or primitive recursion.

We will use Church's lambda notation. For example $\lambda x, y, z[xy + z]$ is the function f where $f(x, y, z) = xy + z$.

1) the n-ary zero function: $\lambda x_1 \dots x_n[0]$
$f_1 = \pi_i^n$, $f_2 = z$, $f_3 = f_2 \circ f_1$.

2) the constant function $\lambda x[2]$
$f_1 = s$, $f_2 = z$, $f_3 = s \circ z$, $f_4 = s \circ f_3$.

3) $\lambda x, y[x + y]$
$f_1 = \pi_1^1 = \lambda x[x]$, $f_2 = \pi_3^3$, $f_3 = s$, $f_4 = f_3 \circ f_4 = \lambda x, y, z[z + 1]$, $f_5 = \lambda x, y[x + y]$ (by primitive recursion using $g = f_1$ and $h = f_4$).

The formal derivations are not very inlightening so we give an informal primitive recursive defintion of addition (and henceforth only give informal defintions):
$x + 0 = x$
$x + (y + 1) = s(x + y)$.

4) multiplication
$x \cdot 0 = 0$
$x \cdot (y + 1) = xy + x$.

5) exponentiation
$x^0 = 1$
$x^{y+1} = x^y \cdot x$.

6) predecesor:

$$pr(x) = \begin{cases} 0, & \text{if } x = 0; \\ x - 1, & \text{otherwise.} \end{cases}$$

$pr(0) = 0$
$pr(y+1) = y.$

7) sign

$$sgn(x) = \begin{cases} 0, & \text{if } x = 0; \\ 1, & \text{otherwise.} \end{cases}$$

$sgn(0) = 0$
$sgn(y+1) = 1$

8) $\dot{-}$

$$x \dot{-} y = \begin{cases} 0, & \text{if } y \leq x; \\ x - y, & \text{otherwise.} \end{cases}$$

$x \dot{-} 0 = x$
$x \dot{-} (y+1) = pr(x \dot{-} y)$

9) Factorials
$0! = 1$
$(n+1)! = n!(n+1)$

If $f(\overline{x}, y)$ is primitive recursive then so is

$$\lambda\overline{x}, n[\sum_{y \leq n} f(\overline{x}, y)]$$

.

$F(\overline{x}, 0) = f(\overline{x}, 0)$
$F(\overline{x}, y+1) = F(\overline{x}, y) + f(\overline{x}, y+1).$

Similary $\lambda\overline{x}, n[\prod_{y \leq n} f(\overline{x}, y)]$ is primitive recursive.

We say that $R(\overline{x})$ is a *primitive recursive predicate* if it is a 0-1 valued primitive recursive function. If P and Q are primitive recursive predicates then so are:
$P \wedge Q(\overline{x}) = P(\overline{x}) \cdot Q(\overline{x})$
$P \vee Q(\overline{x}) = sgn(P(\overline{x}) + Q(\overline{x}))$
$\neg P(\overline{x}) = 1 \dot{-} P(\overline{x})$

10) $x = y$ is a primitive recursive relation.
The characteristic function of $x = y$ is $1 \dot{-} (sgn(x \dot{-} y) + sgn(y \dot{-} x))$

Also if $P(\overline{x}, y)$ is a primitive recursive relation and $g(\overline{x})$ is primitive recursive, then

$$\exists y \leq g(\overline{x}) P(\overline{x}, y) = sgn(\sum_{y \leq g(\overline{x})} P(\overline{x}, y)), \text{ and}$$

$$\forall y \leq g(\overline{x}) P(\overline{x}, y) = sgn(\prod_{y \leq g(\overline{x})} P(\overline{x}, y))$$

are primitive recursive relations.

For example:

11) $x|y = \exists z \leq y \; xz = y$ is primitive recursive.

Exercise 6.8 Show that $x \leq y$ and $x < y$ are primitive recursive relations.

Exercise 6.9 (Definition by cases): Suppose g and h are primitive recursive functions and P is a primitive recursive predcate. Then f is primitive recursive where:

$$f(\overline{x}) = \begin{cases} g(\overline{x}), & \text{if } P(\overline{x}); \\ h(\overline{x}) & \text{otherwise.} \end{cases}$$

Exercise 6.10 Suppose $f(\overline{x}, y)$ is primitive recursive. Let $g(\overline{x}, z) = max\{f(\overline{x}, y) : y \leq z\}$ and $h(\overline{x}, z) = min\{f(\overline{x}, y) : y \leq z\}$. Show that g and h are primitive recursive.

If $P(\overline{x}, y)$ is a primitive recursive function define $\mu y \; P(\overline{x}, y)$ to be the least y such that $P(\overline{x}, y)$ if such a y exists and otherwise $\mu y \; P(\overline{x}, y)$ is undefined. In general $\lambda \overline{x}[\mu y \; P(\overline{x}, y)]$ is only a partial function. Even if it is total, it will not in general be primitive recursive. The next excercise gives the best we can do primitive recursively.

Exercise 6.11 Let $P(\overline{x}, y)$ be a primitive recursive predicate and $g(\overline{x})$ a primitive recursive function. Let

$$f(\overline{x}) = \begin{cases} 0, & \text{if } \forall y \leq g(x) \; \neg P(\overline{x}, y); \\ \mu y \; P(\overline{x}, y), & \text{otherwise.} \end{cases}$$

Then f is primitive recursive.

We next show that coding and decoding of sequences is primitive recursive.

12) "x is prime" is a primitive recursive predicate.
x is prime if and only if $x \neq 0 \wedge x \neq 1 \wedge \forall y \leq pr(x)\ \neg(y|x)$.

13) We next show that the function $\lambda n[p_n]$ is primitive recursive, where p_n is the n^{th} prime number (and $p_0 = 1$). To show this we use the following consequence of Euclid's proof that there are infinitely many primes. For any number $n \geq 1$ there is a prime number p wich that $n < p \leq n! + 1$. Thus:
$p_0 = 1$
$p_{n+1} = \mu x \leq p_n!\ Prime(x)$.

We code the sequence $(n_1 \dots n_m)$ by $x = \prod p_i^{n_i+1}$.

14) x codes a sequence is a primitive recursive predicate.
$Seq(x)$ if and only if $\forall p \leq x \forall q \leq p\ [(Prime(p) \wedge Prime(q) \wedge p|x) \rightarrow q|x]$.

15) Define $l(x)$ to be 0 if x does not code a sequence, otherwise let $l(x)$ be the length of the sequence coded by x.

$$l(x) = \begin{cases} 0, & \neg Seq(x), \\ max\ m(p_m|x), & \text{otherwise.} \end{cases}$$

16) Define $(x)_i$ to be the i^{th} element of the sequence coded by x if x codes a sequence of length at least i, otherwise it is zero.

$$(x)_i = \begin{cases} max\ n(p_i^{n+1}|x), & Seq(x) \wedge i \leq l(x), \\ 0, & \text{otherwise.} \end{cases}$$

We next show that we can simultaneously define several functions by primitive recursion.

Lemma 6.12 *Suppose $g_1, \dots, g_n : \mathbb{N}^k \rightarrow \mathbb{N}$, $h_1, \dots, h_n : \mathbb{N}k + n + 1 \rightarrow \mathbb{N}$ are primitive recursive and we define $f_1, \dots, f_n : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ by*

$$\begin{aligned} f_i(\overline{x}, 0) &= g_i(\overline{x}) \\ f_i(\overline{x}, m+1) &= h_i(\overline{x}, m, f_1(\overline{x}, m), \dots, f_n(\overline{x}, m)). \end{aligned}$$

Then $f_1, \dots, f_n$ are primitive recursive.

Proof We define a primitive recursive function $F : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ such that $F(\overline{x}, m) = \prod_{i=1}^n p_i^{f_i(\overline{x}, m)}$. Then we will have $f_i(\overline{x}, m) = v(p_i, F(\overline{x}, m))$. Let

$$F(\overline{x}, 0) = \prod_{i=1}^n p_i^{g_i(\overline{x})}$$

$$F(\overline{x}, m+1) \quad = \quad \prod_{i=1}^{n} p_i^{h_i(\overline{x},m,v(p_1,F(\overline{x},m)),\ldots,v(p_n,F(\overline{x},m)))}.$$

Then F is primitive recursive and $f_1, \ldots, f_m$ are primitive recursive.

The primitive recursive functions do not exhaust the functions computable by algorithms. Each primitive recursive function has a derivation. As usual we can code each derivation by a natural number. We give a listing of all the primitive recursive functions. Define F_n to be z if n is does not code a derivation, otherwise F_n is the function with derivation coded by n. Intuitively we can do this on such a way that $G = \lambda n, x[F_n(x)]$ is "computable". If this function is primitive recurive, then so is the function $f(x) = G(x,x)+1$. But f can not be primitive recursive, for if $f = F_n$, then $f(n) = G(n,n)+1 = F_n(n)+1 = f(n)+1$. Thus G is "computable", but not primitive recursive.

Exercise 6.13 We can give a more concrete example of a "computable" non-primitive recursive function. For any function F we define the n^{th} iterate of F as follows:
$F^{(0)}(x) = x$
$F^{(n+1)}(x) = F(F^{(n)}(x))$
We now define a sequence of functions $f_0, f_1 \ldots$.
$f_0(x) = x+1$
$f_{n+1}(x) = f_n^{(x)}(x)$.
Define the Ackermann function, $A(x) = f_x(x)$.
a) Show that each f_i is primitive recursive.
b) We say $f \ll g$ if there is a number n such that for all $m > n$, $f(m) < g(m)$. Show that for any primitive recursive function g there is an n such that $g \ll f_n$.
c) Show that for all n, $f_n \ll A$. Thus the Ackermann function is not primitive recursive.

The argument above shows that for any class of total computable functions, we can not give a listing $H_1, H_2 \ldots$ such that the function $\lambda x, y[H_x(y)]$ is computable. For this reason we will also consider partial computable functions. The following definition gives a natural extension of the primitive recursive functions.

The Recursive Funcitons

Definition 6.14 The class of *recursive functions* is the smallest class C of partial functions, containing the the zero function, succesor, and all projection functions and closed under composition, primitive recursion and

vi) (Unboundend Search) If $f(\overline{x}, y)$ is in C, then so is F where $F(\overline{x})$ is the least y such that $f(\overline{x}, y) = 0$ and for all $z < y f(\overline{x}, z)$ is defined. As above we denote F as $\mu y\ f(\overline{x}, y) = 0$.

We use $\uparrow$ to denote "undefined".

For example let

$$g(n) = \begin{cases} 1, & \exists x, y, z\ x^n + y^n = z^n; \\ \uparrow, & \text{otherwise.} \end{cases}$$

Then g is a recursive function.

Our intuition tells us that every recursive function is computable. We will prove that the RM-computable functions are exactly the partial recursive functions.

Theorem 6.15 *Every recursive function is RM-computable.*

Proof Clearly the basic functions z, s and π_i^n are RM-computable. Thus we need only show that the RM-computable functions are closed under composition, primitive recursion and unbounded search.

claim 1 Suppose $f_1, \ldots, f_n : \mathbb{N}^m \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$ are RM-computable. Let $h(\overline{x}) = g(f_1(\overline{x}), \ldots, f_n(\overline{x}))$, then h is RM-computable.

Suppose the computation of P_i is a program to compute f_i. By modifying the program slightly we may assume that:

- P_i does not destroy the input (ie. does not alter registers $R_1, \ldots, R_m$)
- uses only registers $R_{n+i+1}, R_{n+i+2}, \ldots$
- halts with $f_i(\overline{x})$ in R_{n+i}.

[If necessary we modify P_i to P_i^* which starts by copying R_j into R_{n+i_j} for $j \leq n$, and then is identical to P_i except that for all j the role of R_j is played by R_{n+i+j}.]

The program for computing h begins by running the programs $P_1, \ldots, P_m$ (except that HALTS are replaced by jumping to the begining of the next program). Once we run these programs the registers contain $a_1, \ldots, a_n, f_1(\overline{a}), \ldots, f_m(\overline{a})$.

We next write $f_1(\overline{a}), \ldots, f_m(\overline{a})$ into the first m-registers and erase all of the other registers which were used in the earlier computations. We now run the program to compute g.

claim 2 Suppose $h : \mathbb{N}^m \to \mathbb{N}$ and $g : N^{m+2} \to \mathbb{N}$ are RM-computatble (possibly partial) functions. Let f be defined from g and h by primitive recursion. Then f is RM-computable.
step 0:
We start with $\overline{x}, y$ in the first $m+1$-registers.
• let $r_{m+2} \leftarrow 0$; this will be a counter
• copy $\overline{x}$ into $R_{m+3}, \ldots, R_{2m+2}$
• run the program for h suitably modified such that we end with the configuration

$$(\overline{x}, y, 0, h(\overline{x}), 0, 0, \ldots).$$

In general at the end of step s we will have $(\overline{x}, y, s, f(\overline{x}, s), 0, 0, \ldots)$.
step $s+1$
• if $r_{m+2} = r_{m+1}$ we are done, fiddle with things so that the configuration is

$$(f(\overline{x}, s), 0, 0, \ldots)$$

and halt, otherwise.
• increment r_{m+3}. Move things around so that we have configuration

$$(\overline{x}, y, s+1, \overline{x}, y, f(\overline{x}, s), 0, 0, \ldots).$$

Run the program for g suitably modified so that we end with configuration

$$(\overline{x}, y, s+1, f(\overline{x}, s+1), 0, 0, \ldots).$$

• Go to next step.
This program computes f

claim 3 If $f(\overline{x}, y)$ is RM-computable, then $\mu y\, f(\overline{x}, y) = 0$ is RM-computable.
Consider the following program:
• Start with configuration $(\overline{x}, 0, 0, 0, \ldots)$.
<u>stage s</u>:
• At the beging of stage s we will have configuration.

$$(\overline{x}, s, 0, 0, \ldots)$$

• Change configuration to $(\overline{x}, s, \overline{x}, s, 0, \ldots)$.

• Run modified version of program for f if this halts we will have configuration

$$(\overline{x}, s, f(\overline{x}, s), 0, 0 \ldots).$$

• If $f(\overline{x}, s) = 0$ halt with configuration $(s, 0, 0, \ldots)$. If not change to configuration $(\overline{x}, s+1, 0, 0, \ldots)$ and go to next stage.

If there is an s such that $f(\overline{x}, s) = 0$ and for all $t < s$, $f(\overline{t}, s) \downarrow\neq 0$, then we will eventually halt and output s. Otherwise the search will continue forever.

Thus every recursive function is RM-computable.

Theorem 6.16 *Every RM-computable function is recursive.*

Proof Let $f : \mathbb{N}^n \to \mathbb{N}$ be RM-computable (possibly partial). Let $I_1, \ldots, I_m$ be a program which computes f. Suppose this program uses only registers $R_1, \ldots, R_N$. We define two primitive recursive functions $g_1, \ldots, g_N : \mathbb{N}^{m+1} \to \mathbb{N}$ and $j : \mathbb{N}^{m+1} \to \mathbb{N}$ such that:

$$g_i(\overline{x}, s) = \text{contents of } R_i \text{ at stage } s \text{ on input} \overline{x}$$

and

$$j(\overline{x}, s) = \begin{cases} 0 & \text{if the machine on input } x \text{ has halted by stage } s \\ j & \text{if } I_j \text{ is the next instruction to be executed.} \end{cases}$$

Let $h(x) = \mu s\ j(x, s) = 0$. Then $f(x) = g_1(x, h(x))$.

The construction of g_i and j are routine but tedious primitive recursions. We give one example. Consider the program to compute

$$f(x, y) = \begin{cases} x - y & y \leq x \\ \uparrow & y > x. \end{cases}$$

```
1)  Z(3)
2)  J(1,2,6)
3)  S(2)
4)  S(3)
5)  J(1,1,2)
6)  T(3,1)
```

$$j(x,y,s) = \begin{cases} 1 & s=0 \\ 2 & s=1 \text{ or } j(x,y,s-1)=5 \\ 3 & j(x,y,s-1)=2 \\ 4 & j(x,y,s-1)=3 \\ 5 & \text{j(x,y,s-1)=4} \\ 6 & j(x,y,s-1)=2 \text{ and } g_2(x,y,s-1)=g_1(x,y,s-1) \\ 7 & j(x,y,s-1)=6 \\ 0 & j(x,y,s-1)=7 \text{ or } 0. \end{cases}$$

$$g_1(x,y,0)=x$$

$$\begin{aligned} g_2(x,y,0) &= y \\ g_2(x,y,s+1) &= \begin{cases} g_2(x,y,s) & j(x,y,s)\neq 3 \\ g_2(x,y,s)+1 & \text{otherwise.} \end{cases} \end{aligned}$$

$$\begin{aligned} g_3(x,y,0) &= 0 \\ g_3(x,y,s+1) &= \begin{cases} g_3(x,y,s) & j(x,y,s)\neq 4 \\ g_3(x,y,s)+1 & \text{otherwise.} \end{cases} \end{aligned}$$

These functions are clearly primitive recursive.

Church's Thesis

Church's Thesis *A partial function is computable if and only if it is partial recursive*

Church's Thesis asserts that the partial recursive functions, or the RM-computable functions, completely capture our intuitve notion of computability.

We will use Church's Thesis frequently in arguments by giving an intuitive argument that a function is computable and then asserting that therefore it is recursive or RM-computable. Whenever we make such an argument, we are asserting that, if challenged, we could produce the RM-machine code that would compute the function.

There is a great deal of evidence for Church's Thesis. Any reasonable notion of "computabile function" has been shown to be equivalent to "partial recursive" or "RM-computable". Indeed, Church first stated the conjecture for functions definable in λ-calculus, an ancestor of the LISP programming language.

Random Access Machines

We give one more argument towards the plausibility of Church's thesis. One aspect of modern computing that is missing in register machines is dynamic access to memory. In a modern computer language we can compute a number n and then store another number in memory cell n. We will describe a generalization of register machines that allows this kind of dynamic access and prove that they do not allow us to compute new functions.

Definition 6.17 A *Random Access Machine* is one where we have memory locations $M_0, M_1, M_2, M_3, \ldots$. Let m_i be the contents of M_i. A program for a random access machine is a finite sequence of instructions $I_1, \ldots, I_m$. Where the allowable instructions are:
i) Z(n); set m_n to zero
ii) S(n); increment m_n
iii) J(i,j,l); if $m_i = m_j$, go to instruction l.
iv) T(i,j); transfer the contents of M_{m_i} to M_{m_j}
v) HALT

The key difference is that we are allowed to specifiy in the program what address we want to store something in.

A function f is said to be *RAM-computable* if there is a random access machine program which given initial configuration $(x, 0, 0, \ldots)$ halts with $f(x)$ in M_0 if $x \in \text{dom}(f)$ and does not halt if $x \notin \text{dom}(f)$.

Exercise 6.18 Every RM-computable function is RAM-compuable.

We next out line the proof that every RAM-computable function is RM-computable. The key idea is to code configurations of the RAM as a single number. Suppose at some stage s, n is the largest memory location that we have used. Then the configuration of the machine is given by the sequence $(m_1, \ldots, m_n, 0, 0, 0, \ldots)$.

We code this configuration with the number $\prod p_i^{m_i}$. All of the operations of the machine correspond to simple arithmetic operations on the code.

Let $v(p, x)$ = largest power of p dividing x. Note that $v(p_i, x)$ extracts the contents of M_i from the code x.

For example: • Z(n): corresponds to the operation

$$x \mapsto \frac{x}{p_n^{v(p,x)}}.$$

• S(n): corresponds to

$$x \mapsto xp_n.$$

• T(i,j): Let $l = v(p_j, x)$ and $k = v(p_i, x)$. The new configuration is coded by

$$\frac{x}{p_l^{v(p_l,x)}} p_l^{v(p_k,x)}$$

Exercise 6.19 Using the above idea show that any RAM computable function is RM computable.

Henceforth we will usually use Church's thesis blindly. We will say that a partial function is *computable* if it is RM-computable with full confidence that anything which is intuitively computable can be done with a register machine.

7 Universal Machines and Undecidability

Our main goal in this section is to prove that there is a computable partial function $\Psi : \mathbb{N}^2 \to \mathbb{N}$ such that if ϕ_n is the function

$$\phi_n(x) = \Psi(n, x)$$

then $\phi_0, \phi_1, \ldots$ is an enumeration of all computable partial functions.

We will code register machine programs by natural numbers and we will arange the coding so that each number codes a program. If P_n is the program with code n, then $\phi_n(x)$ will be the result of running P_n on input x.

The register machine computing Ψ is a *universal register machine.* It behaves like a modern compiler. If f is a computable function we find e such that $f = \phi_e$ and compute $f(x)$ by computing $\Psi(e, x)$.

Our first task is to code register machine programs. We will use a more subtle coding than the one of §6 to insure that every natural number codes a program.

Let $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by $\pi(m,n) = 2^m(2n+1) - 1$.

Lemma 7.1 *π is a bijection and both π and π^{-1} are computable (indeed primitive recursive).*

Proof Clearly π is primitive recursive. To calculate $\pi^{-1}(x)$, factor $x+1 = yz$ where y is a power of 2 and z is odd. Then $m = \log_2 y$ and $n = \frac{z-1}{2}$.

Once we can code pairs it is easy to code triples. We view (a,b,c) as $((a,b),c)$. Let $\psi : \mathbb{N}^3 \to \mathbb{N}$ by

$$\psi(p,q,r) = \pi(\pi(p,q),r).$$

Let I be the set of all instructions for register machines. There is $\beta : I \to \mathbb{N}$ a computable bijections.

$\beta(\text{HALT}) = 0$
$\beta(\text{Z}(n)) = 4(n-1)+1$
$\beta(\text{S}(n)) = 4(n-1)+2$
$\beta(\text{T}(m,n)) = 4(\pi(m-1,n-1))+3$
$\beta(\text{J}(m,n,r)) = 4(\psi(m-1,n-1,r-1))+4$

β is easy to decode. For example for what i is $\beta(i) = 47$? Since $47 \prec 3(\text{mod}4)$, i must code T(m,n) for some m and n, where $\pi(m-1,n-1) = \frac{47-3}{4} = 11$. Since $11+1 = 2^2(2\cdot 1+1)$, $\pi(2,1) = 11$. Thus i codes the instruction $T(3,2)$.

We also want

$$\tau : \bigcup_{k>0} \mathbb{N}^k \to \mathbb{N}$$

a computable bijection with computable inverse. We let

$$\tau(a_1,\ldots,a_k) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} \ldots + 2^{a_1+\ldots+a_k+k-1} - 1.$$

Given x we calculate $\tau^{-1}(x)$ as follows:

i) find the binary expansion of $x+1 = 2^{b_1} + \ldots + 2^{b_k}$

ii) Let $a_1 = b_1$ and $a_{i+1} = b_{i+1} - b_i - 1$ for $1 \le i < k$.

For example we calculate $\tau^{-1}(47)$: $47+1 = 2+2^2+2^3+2^5$. Thus $a_1 = 1$, $a_2 = 0$, $a_3 = 0$,and $a_4 = 1$. Thus $\pi^{-1}(47) = (1,0,0,1)$ [note: $47 = 2^1 + 2^{1+0+1} + 2^{1+0+0+2} + 2^{1+0+0+1+3} - 1$]

We now give a method for coding all register machine programs. Let P be the program $I_1, \ldots, I_m$ by

$$\gamma(P) = \tau(\beta(I_1), \ldots, \beta(I_m)).$$

For $m \in \mathbb{N}$, let $P_m = \gamma^{-1}(m)$. Let $\phi_m^{(n)}$ be the n-ary function computed by program P_m. Clearly $\phi_0^{(n)}, \phi_1^{(n)}, \ldots$ is a list of all partial recursive functions in n-variables. [We will supress the superscript if it is clear]

If f is computable we say that n is an *index* for f if $f = \phi_n$. There will usually be many indicies for f.

Consider the partial function $\Psi^{(n)} : \mathbb{N}^{n+1} \to \mathbb{N}$ by $\Psi^{(n)}(e, \overline{x}) = \phi_e^{(n)}(\overline{x})$.

Theorem 7.2 *The functions $\Psi^{(n)}$ are computable.*

Proof For notational simplicity we will consider only the case $n = 1$.

Informally we compute $\Psi(e, x)$ by decoding e to obtain the program P_e. Simulate program P_e on input x.

We use one number to store the register configuration in the simulation. Suppose we are using registers $R_1, \ldots, R_m$ and R_i contains r_i. We will code this configuration by

$$c = \prod_{i=1}^{m} p_i^{r_i}.$$

We call c the *configuration code* of the machine. The *current state* of the machine will be $\sigma = \pi(c, j)$ where j is the next instruction to be executed (and if we have halted $j = 0$) [here π is the pairing function].

Define $c(e, x, t) =$ configuration after t steps of program P_e on input x if we have not yet halted. If we have halted let $c(e, x, t)$ be the final configuration.

Let $j(e, x, t) =$ number of the next instruction if the computation of P_e on input x has not halted by step t and let it be 0 otherwise.

Let $\sigma(e, x, t) = \pi(c(e, x, t), j(e, x, t))$.

claim c, j and σ are computable (indeed they are primitive recursive).

• $c(e, x, 0) = 2^x$ and $j(e, x, 0) = 1$.

• Given $c = c(e, x, t)$ and $j = j(e, x, t)$, we compute $j(e, x, t + 1)$ and $c(e, x, t + 1)$.

• If $j = 0$, then $c(e, x, t + 1) = c$ and $j(e, x, t + 1) = j$.

• If $j > 0$, then decode e to find I_j.

• If I_j is $I(m)$ then $c(e, x, t+1) = c \cdot p_m$ and $j(e, x, t+1) = j+1$.

• If I_j is $Z(m)$ then $c(e, x, t+1) = \frac{c}{p_m^l}$ where l is the largest such that p_m^p divides c, and $j(e, x, t+1) = j+1$.

• If I_j is $T(n, m)$ then $c(e, x, t+1) = c \cdot p_m^{l-k}$ where l is largest such that p_n^l divides c and k is largest such that p_m^l divides c. Let $j(e, x, t+1) = j+1$.

• If I_j is $J(n, m, i)$ then $c(e, x, t+1) = c$ and $j(e, x, t+1) = i$ if the largest k such that p_m divides c is equal to the largest l such that p_n divides c, and otherwise $j(e, x, t+1) = j+1$.

• If I_j is HALT, then $c(e, x, t+1) = c$ and $j(e, x, t) = 0$.

Once we know that c and j are computable (indeed primitive recursive), we obtain a general recursive $h(e, x) = \mu t j(e, x, t) = 0$. Then $\Psi(e, x)$ is the largest n such that 2^n divides $c(e, x, h(e, x))$. Clearly Ψ is computable.

The machine that computes Ψ is called the *Universal Register Machine*.

Definition 7.3 Let $T = \{(e, x, s) : P_e$ on input x halts by stage $s\}$. This is called *Kleene's T-predicate*. The arguments above show that T is primitive recursive.

The following theorem is often useful. (For some reason it is often refered to as the *s-m-n theorem*).

Lemma 7.4 (Parameterization Lemma) *If $f(x, y)$ is a computable partial function then there is a total computable function $k(x)$ such that for all x, $k(x)$ is an index for the function $y \mapsto f(x, y)$. Indeed the function $k(x)$ can be choosen one to one.*

Proof Let P be a program computing $f(x, y)$ [starting with x in R_1 and y in R_2. Consider the following program Q_n. Start with y in register 1.

```
1)      T(2,1)      $r_2 \leftarrow r_1$
2)      Z(1)        $r_1 \leftarrow 0$
3)      S(1)        $r_1 \leftarrow 1$
4)      S(1)        $r_1 \leftarrow 2$
⋮       ⋮
n+2)    S(1)        $r_1 \leftarrow n$
        $P$
```

If we start with input y, after step $n+2$ we will have n in R_1 and y in R_2. Running the program P will compute $f(n, y)$.

Thus the program Q_n is a program to compute $\lambda y[f(n, y)]$. The function k is the function which takes us from n to a code for the program P_m. k is easily seen to be one to one.

Definition 7.5 We say that a set $A \subset \mathbb{N}^m$ is *recursive* if it's characteristic function

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

is computable.

Since there are $2^{\aleph_0}$ subsets of $\mathbb{N}$ and only $\aleph_0$ possible algorithms, most subsets of $\mathbb{N}$ are not computable. Turing gave an important natural example.

Let $H = \{(e, x) : \phi_e(x) \downarrow\}$. We call H the *halting problem.*

Let $K = \{e : \phi_e(e) \downarrow\}$.

Theorem 7.6 (Unsolvability of the Halting Problem) *Neither H nor K is not recursive.*

Proof If H were recursive then K would be recursive so it suffices to show that K is not recursive. Suppose K is recursive. Let P be a program computing the characteristic function of K. Consider the following program $\widehat{P}$.

• On input x, run program P. If P outputs 0, then halt. If P outputs 1, then go into an infinite loop.

Suppose $I_1, \ldots, I_m$ is the program P. Let $\widehat{I_1}, \ldots \widehat{I_m}$ be the same program where every HALT has been replaced by J(1,1,$m+1$), then $\widehat{P}$ is

```
1)      $\widehat{I_1}$
⋮       ⋮
m)      $\widehat{I_m}$
m+1)    Z(2)
m+2)    J(1,2,$m+4$)
m+3)    J(1,1,$m+2$)
m+4)    HALT
```

For some e, $\widehat{P} = P_e$. Then

$$\phi_e(x) = \begin{cases} 0 & x \notin K \\ \uparrow & x \in K. \end{cases}$$

Is $e \in K$?

$$e \in K \Leftrightarrow \phi_e(e)\downarrow \Leftrightarrow e \notin K$$

a contradiction. Thus K is not recursive.

Definition 7.7 Let $Tot = \{e : \phi_e \text{ is total}\}$.

We argue that Tot is not recursive. Suppose it were, let g be the characteristic function of Tot. Let

$$f(x) = \begin{cases} \phi_x(x) + 1 & \text{if } g(x) = 1 \\ 0 & \text{if } g(x) = 0. \end{cases}$$

If g is computable, then f is computable. In fact

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

Thus for some e, $f = \phi_e$. Also f is easily seen to be total. But then $\phi_e(e)\downarrow$ and $f(e) = \phi_e(e) + 1$, a contradiction.

We will give other natural examples in §8.

We will finish this section with an application to logic.

Theorem 7.8 (Church) *The set of valid sentences of first order logic is not recursive.*

Proof For any P and any natural number n we will give a sentence θ_n^P such that θ_n^P is valid if and only if P halts on input n. If we had a program to decide if a sentence is valid, then we would have an algorithm to decide the halting problem.

Suppose P uses registers $R_1, \ldots, R_m$. Let $P = I_1, \ldots, I_s$. Let $\mathcal{L} = \{0, s, R\}$ where s is a unary function symbol and R is an $m+1$-ary predicate. We use $s^n(x)$ to denote

$$\underbrace{s(s(\ldots(x)\ldots))}_{n \text{ times}}.$$

The intended interpretation is that $s^n(0) = n$ and $R(s^{n_1}(0), \ldots, s^{n_m}(0), s^j(0))$ holds iff and only if one possible configuration of the machine is that R_i is n_i and the next instruction is j.

For each instruction I_i we write down an axiom τ_i where:

i) If I_i is $Z(l)$, then τ_i is

$$\forall x_1, \ldots, x_m\ (R(x_1, \ldots, x_m, s^i(0)) \rightarrow R(x_1, \ldots, x_{l-1}, 0, x_{l+1}, \ldots, x_m, s^{i+1}(0))).$$

ii) If I_i is $S(l)$, then τ_i is

$\forall x_1, \ldots, x_m\,(R(x_1, \ldots, x_m, s^i(0)) \rightarrow R(x_1, \ldots, x_{l-1}, s(x_l), x_{l+1}, \ldots, x_m, s^{i+1}(0)))$.

iii) If I_i is $T(i, l)$, then τ_i is

$\forall x_1, \ldots, x_m\,(R(x_1, \ldots, x_m, s^i(0)) \rightarrow R(x_1, \ldots, x_{l-1}, x_i, x_{l+1}, \ldots, x_m, s^{i+1}(0)))$.

iv) If I_i is $J(i, l, j)$, then τ_i is

$\forall x_1, \ldots, x_m\,(R(\overline{x}, s^i(0)) \rightarrow ((x_i = x_l \rightarrow R(\overline{x}, s^j(0)) \wedge ((x_i \neq x_l \rightarrow R(\overline{x}, s^{i+1}(0))$

v) If I_i is HALT, then τ_i is

$$\forall \overline{x}\ R(\overline{x}, s^i(0)) \rightarrow R(\overline{x}, 0).$$

The sentence

$$R(s^n(0), 0, \ldots, 0, s(0))$$

corresponds to the initial configuration on input n.

Let θ_n^P be

$$(R(s^n(0), 0, \ldots, 0, s(0)) \wedge \bigwedge_{i=1}^{s} \tau_i) \rightarrow \exists x\ R(\overline{x}, 0)$$

Then P halts on input n if and only if θ_n^P is valid. If validity were recursive then we could decide the halting problem.

8 Recursively Enumerable and Arithmetic Sets

Definition 8.1 A set $X \subset \mathbb{N}$ is *recursively enumerable* if X is the range of a partial recursive function.

Intuitively there is a recurisve function such that $f(0), f(1), \ldots$ enumerates all of X.

Proposition 8.2 *Let X be nonempty. The following are equivalent:*

i) X is recursively enumerable;

ii) $X = \emptyset$ or X is the range of a total recursive function.

iii) there is a recursive $Y \subset \mathbb{N}^{m+1}$ such that $X = \{y : \exists \overline{x}\ (\overline{x}, y) \in Y\}$;

iii) X is the domain of a partial recursive function;

Proof

i)⇒ii) Suppose $X \neq \emptyset$ is the domain of the partial recursive function f. Let $x_0 \in X$. Let $g : \mathbb{N}^2 \to \mathbb{N}$ by

$$g(x,s) = \begin{cases} f(x) & \text{if } T(e,x,s) \\ x_0 & \text{otherwise} \end{cases}$$

where $T(e,x,s)$ is the Kleene T-predicate asserting P_e halts on input x by stage s. Then g is total recursive and the range of X is equal to the range of g. If $\sigma : \mathbb{N} \to \mathbb{N}^2$ is a recursive bijection, then $\widehat{g} = g \circ \sigma$ is the desired function.

ii)⇒iii) Let X be the range of f. Let $Y = \{(x,y) : f(x) = y\}$. Then Y is recursive and $X = \{y : \exists x\ f(x) = y\}$.

iii)⇒v) Let $Y \subset \mathbb{N}^{m+1}$. Let $\sigma : \mathbb{N} \to \mathbb{N}^m$ be a recursive bijection. Let $f : \mathbb{N} \to \mathbb{N}$, by $f(n) = \mu x\ (\sigma(x), n) \in Y$. f is partial recursive and X is the domain of f.

iii)⇒iv) Let X be the domain of f. Let

$$g(x) = \begin{cases} x & f(x)\downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

Then g is partial recursive and the range of g is the domain of f.

We next fix an enumeration of the recursively enumerable sets.

Definition 8.3 Let $W_e = \{x : \phi_e(x)\downarrow\} = \text{dom}\phi_e$. Then $W_0, W_1, W_2, \ldots$ is an enumeration of the recursively enumerable sets.

The Halting set $H = \{(e,x) : \phi_e(x)\downarrow\}$ is the domain of the universal function Ψ. Thus H is recursively enumerable. Similarly $K = \{e : \phi_e(e)\downarrow\}$ is the domain of $e \mapsto \psi(e,e)$ and hence recursively enumerable. Thus there are recursively enumerable sets which are not recursive.

Recursively enumerable sets arise naturally in logic when we take the set of logical consequences of a theory. For the moment this will be informal (since we are talking about sets of sentences rather than natural numbers). They will me made precise in §11 when we talk about coding formulas.

Suppose T is a recusive set of sentences. Then $Cn(T) = \{\phi : T \vdash \phi\}$ is recursively enumerable as $Cn(T) = \{\phi : \exists p\ p \text{ is a proof of } \phi \text{ from } T\}$. By ii) $Cn(T)$ is recursively enumerable.

Proposition 8.4 *If A and B are recursively enumerable, then $A \cup B$ and $A \cap B$ are recursively enumerable.*

Proof We give intuitive arguments which can easily be made precise.

Suppose we have programs enumerating A and B. To enumerate $A \cup B$, we enumerate x whenever we see x appear in either the enumeration of A or the enumeration of B.

To enumerate $A \cap B$, we enumerate x once we see x appear in the enumeration of both A and B.

Proposition 8.5 *Every recursive set is recursively enumerable.*

Proof Let f be the characteristic function for A and let

$$g(x) = \begin{cases} 1 & f(x) = 1 \\ \uparrow & f(x) \neq 0. \end{cases}$$

Then $A = \mathrm{dom} g$.

Proposition 8.6 *A is recursive if and only if A and $\neg A$ are recursively enumerable.*

Proof If A is recusive, then $\neg A$ is recursive. Thus, by Proposition 8.5 both A and $\neg A$ are recursively enumerable.

If A and $\neg A$ are recursively enumerable, then we can decide if $x \in A$ as follows: start enumerating A and $\neg A$. We will eventually find x in one of the two lists. If x is enumerated into A, then output x. If x is enumerated into $\neg A$, output no.

Corollary 8.7 *$\neg K$ and $\neg H$ are not recursively enumerable.*

Proof Otherwise K and H are recursive by 8.6.

Definition 8.8 $A \leq_m B$ (A is many-one reducible to B) if there is a total recursive $f : \mathbb{N} \to \mathbb{N}$ such that $x \in A \Leftrightarrow f(x) \in A$.

If $A \leq_m B$ then B is at least as complicated as A. We can reduce problems about A to probelms about B. We next show that the Halting Problem is the most complicated recursively enumerable set.

Lemma 8.9 *Suppose $A \leq_m B$. If B is recursive, then so is A. Also if B is recursively enumerable so is A.*

Proof If B is recursive this is clear. Suppose B is recursively enumerable. Suppose g is partial recursive and $B = \text{dom}g$. Suppose f is total recursive and $n \in A$ iff $f(n) \in B$. Then $A = \{n : g(f(n)) \downarrow\}$ a recursively enumerable set.

Lemma 8.10 *If A is recursively enumerable, then $A \leq_m H$.*

Proof Suppose A is the domain of ϕ_e. Let $f(n) = (e, n)$. Then

$$\begin{aligned} n \in A &\Leftrightarrow \phi_e(n) \downarrow \\ &\Leftrightarrow \Psi(e, n) \downarrow \\ &\Leftrightarrow f(e, n) \in H. \end{aligned}$$

Lemma 8.11 *If A is recursively enumerable $A \leq_m K$.*

Proof If suffices to show $H \leq_m K$. There is a total recursive function g such that for all e, x, y, $\phi_{g(e,x)}(y) = \phi_e(x)$. Intuitively g is a function which on input e and x outputs a program P, such that on any input y, P runs P_e on input x.

More formally let $G(e, x, y) = \Psi(e, x)$. Apply the Parameterization Lemma (to obtain a total recursive $g(e, x)$ such that $\phi_{(g(e,x)}(y) = G(e, x, y) = \phi_e(x)$. Then $(e, x) \in H$ if and only if for all y, $\phi_{g(e,x)}(y) \downarrow$ if and only if $\phi_{g(e,x)}(g(e, x)) \downarrow$.

Thus $(e, x) \in H$ if and only if $g(e, x) \in K$, so $H \leq_m K$.

Thus A is recursively enumerable if and only if $A \leq_m H$ if and only if $A \leq_m K$.

Recall that $Tot = \{e : \phi_e \text{ is total}\}$. We will show that

Lemma 8.12 *i) $K \leq_m Tot$*
ii) $\neg K \leq_m Tot$
iii) Neither Tot nor $\neg Tot$ is recursively enumerable.

Proof
i) Define a total recursive function $f(x)$ such that for all e, $\phi_{f(e)}(y) = \phi_e(e)$. (The existence of such an f follows from the parameterization lemma.) Then $e \in K \Leftrightarrow f(e) \in Tot$.

ii) Define a total recursive function $f(x)$ such that

$$\phi_{f(e)}(s) = \begin{cases} 1 & \phi_e(e) \text{ has not halted by stage } s \\ \uparrow & \text{otherwise.} \end{cases}$$

Let

$$G(e,s) = \begin{cases} 1 & \neg T(e,e,s) \\ \uparrow & \text{otherwise} \end{cases}$$

and apply the paramterization lemma to obtain a total recursive g such that $\phi_{g(e)}(s) = G(e,s)$. Then $e \notin K$ if and only if there is an s such that $T(e,e,s)$ if and only if there is an s such that $\phi_{g(e)}(s)\uparrow$. Thus $e \in \neg K \Leftrightarrow g(e) \in K$.

iii) If Tot were recursively enumerable, then since $\neg K \leq_m Tot$, $\neg K$ would be recursively enumerable and K would be recursive.

Note that if $x \in A \Leftrightarrow f(x) \in B$, then $x \notin A \Leftrightarrow f(x) \notin B$. So $A \leq_m B \Leftrightarrow \neg A \leq_m \neg B$. Thus since $K \leq_m Tot$, $\neg K \leq_m \neg Tot$. If $\neg Tot$ were recursively enumerable then $\neg K$ would be recursively enumerable, a contradiction.

Definition 8.13 We say that $X \subset \mathbb{N}^m$ is Σ_1 if and only if there is a recursive $Y \subset \mathbb{N}^{m+n}$ such that

$$X = \{\overline{x} \in \mathbb{N}^m : \exists \overline{y}\ (\overline{x}, \overline{y}) \in Y\}.$$

We say that $X \subset \mathbb{N}^m$ is Π_n if and only if $\neg X$ is Σ_n. X is Σ_{n+1} if and only if there is a Π_n set $Y \subset \mathbb{N}^{m+k}$ such that

$$X = \{\overline{x} : \exists \overline{y}\ (\overline{x}, \overline{y}) \in Y\}.$$

We say that X is Δ_n if and only if X is Σ_n and X is Π_n.

By 8.2 the Σ_1 sets are exactly the recursively enumerable sets. Note that the Δ_1 sets are the recursive sets. It is easy to see that $\Sigma_n \cup \Pi_n \subseteq \Delta_{n+1}$.

Definition 8.14 We say that X is *arithmetic* if $X \in \cup_n \Sigma_n$.

Proposition 8.15 *i) If A_0 and A_1 are Σ_n (Π_n), then $A_0 \cap A_1$ and $A_0 \cup A_1$ are Σ_n (Π_n).*

ii) If $A \subset \mathbb{N}^{m+1}$ is Σ_n, then $\{\overline{x} : \exists y\ (\overline{x}, y) \in A\}$ is Σ_n.

iii) If $A \subset \mathbb{N}^{m+1}$ is Π_n, then $\{\overline{x} : \forall y\ (\overline{x}, y) \in A\}$ is Π_n.

iv) If $A \subset \mathbb{N}^{m+1}$ is Σ_n and $f : \mathbb{N}^m \to \mathbb{N}$ is total recursive, then $\{\overline{x} : \forall y < f(\overline{x})\ (\overline{x}, y) \in A\}$ is Σ_n.

v) If $A \subset \mathbb{N}^{m+1}$ is Π_n and $f : \mathbb{N}^m \to \mathbb{N}$ is total recursive, then $\{\overline{x} : \exists y < f(\overline{x})\ (\overline{x}, y) \in A\}$ is Π_n.

vi) If A is Σ_n (Π_n) and $B \leq_m A$, then B is Σ_n (Π_n).

Proof

i) Let $A_i = \{\overline{x} : \exists \overline{y}\ (\overline{x}, \overline{y}) \in B_i\}$ where B_i is Π_{n-1} (or recursive if $n = 1$. Then $A_0 \cup A_1 = \{\overline{x} : \exists \overline{y}\ ((\overline{x}, \overline{y}) \in B_0 \cup B_1)\}$. By induction $B_0 \cup B_1$ is Π_{n-1}. Thus $A_0 \cup A_1$ is Σ_n.

Similarly $A_0 \cap A_1 = \{\overline{x} : \exists \overline{y_0} \exists \overline{y_1}\ ((\overline{x}, \overline{y_0}) \in B_0 \wedge (\overline{x}, \overline{y_1} \in B_1\}$.

ii) and iii) are similar.

iv) Suppose $A = \{(\overline{x}, y) : \exists \overline{z}(\overline{x}, y, \overline{z}) \in B\}$. Then $\forall y < f(\overline{x}) \exists \overline{z}(\overline{x}, y, \overline{z}) \in B$ iff and only if $\exists \sigma(\overline{x}, y, \sigma) \in B^*$, where we think of σ as coding a finite sequence $(\overline{z_0}, \ldots, \overline{z_{f(\overline{x})-1}})$ and B^* asserts that forall $y < f(\overline{x})$, $(\overline{x}, y, \overline{z_y}) \in B$. Since Π_{n-1} sets are closed under $\forall y$, B^* is Π_{n-1}. Thus our set is Σ_n.

v) is similar

vi) Suppose A is Σ_n. Let f be a total recursive function such that

$$x \in B \Leftrightarrow f(x) \in A.$$

Let

$$Y = \{(x, y) : y \in A \wedge f(x) = y\}.$$

Then $Y \in \Sigma_n$ and $B = \{x : \exists y\ (x, y) \in A\}$ is Σ_n.

Examples

Below let $W_e^s = \{x : \phi_e(x) \downarrow \text{ by stage } s\}$. Clearly W_e^s is recursive.

- $Tot = \{e : \phi_e \text{ is total}\}$ is Π_2 as

$$e \in Tot \Leftrightarrow \forall n \exists s x \in W_e^s.$$

- $Fin = \{e : W_e \text{ is finite}\}$ is Σ_2 as

$$e \in Fin \Leftrightarrow \exists n \forall y \forall s\ (y < x \vee y \notin W_e^s).$$

- $\{(a, b, c, d, e) : \exists x, y \forall z\ az^3 - bxz = cx^2 - dxy^2 + ey^3\}$ is Σ_2.

- $\{e : W_e \text{ is recursive}\}$ is Σ_3 as W_e is recursive if and only there is an i such that $\neg W_e = W_i$. Thus W_e is recursive iff and only if

$$\exists i \forall x\ ((x \in W_e \vee x \in W_i) \wedge (x \notin W_e \vee x \notin W_i)).$$

This is equivalent to

$$\exists i \forall x (\underbrace{\underbrace{\exists s (x \in W_e^s \vee x \in W_i^s)}_{\Sigma_1} \wedge \underbrace{\forall s (x \notin W_e^s \vee x \notin W_i^s)}_{\Pi_1}}_{\Pi_2}).$$

Thus $\{e : W_e \text{ is recursive}\}$ is Σ_3.

Complete Sets

Definition 8.16 For Γ be Σ_n or Π_n. We say that X is Γ-*complete* if $X \in \Gamma$ and for all $Y \in \Gamma$, $Y \leq_m X$.

By 8.11 K and H are Σ_1-complete.

Proposition 8.17 *Tot is Π_2-complete.*

Proof Let X be Π_2. Then there is a recurisve $R(x, y, z)$ such that

$$x \in X \Leftrightarrow \forall y \exists z\ R(x, y, z).$$

Let $f(x, y) = \begin{cases} 1 & \exists z\ R(x, y, z) \\ \uparrow & \text{otherwise} \end{cases}$. Clearly f as computable as on input x, y we search for a z such that $R(x, y, z)$. If there is one we will evenutally find it and halt. If not we will search forever.

By the parameterization theorem there is a recursive function $k(x)$ such that

$$\phi_{k(x)}(y) = f(x, y).$$

But then $x \in X$ if and only if $\phi_{k(x)}$ is total.

Proposition 8.18 *Fin is Σ_2-complete.*

Proof

Let $X \in \Sigma_2$. Suppose $x \in X$ if and only if $\exists y \forall z\ R(x, y, z)$ where R is recursive.

Let

$$f(x, y) = \begin{cases} 1 & \forall w \leq y \exists z\ \neg R(x, w, z) \\ \uparrow & \text{otherwise.} \end{cases}$$

By the parameterization theorem there is a total recursive g such that $\phi_{g(x)}(y) = f(x, y)$.

Then $W_{g(x)} = \{y : \forall w < y \exists z \ \neg R(x,w,z)\}$. Thus $x \in X$ if and only if $g(x) \in Fin$.

Definition 8.19 Let $U \subset \mathbb{N}^2$. For $e \in \mathbb{N}$, let $U_e = \{x : (e,x) \in U\}$. We say that U is Γ-*universal* if $U \in \Gamma$ and for any $X \in \Gamma$, there is an e such that $X = U_e$.

Clearly every Γ-universal set is Γ-complete

Lemma 8.20 *For $\Gamma = \Sigma_n$ or Π_n, there is U_Γ which is Γ-universal.*

Proof Let $U_{\Sigma_1} = \{(e,n) : n \in W_e\} = \{(e,n) : \Psi(e,n) \downarrow\}$ is Σ_1 and clealy universal.

If U_{Σ_n} is universal for Σ_n then $\neg U_{\Sigma_n}$ is universal for Π_n.

Let U_{Π_n} be universal Π_n. Let $\pi : \mathbb{N}^2 \to \mathbb{N}$ be a recursive bijection. Then

$$\{(e,n) : \exists y (e, \pi(x,y)) \in U_{\Pi_n}\}.$$

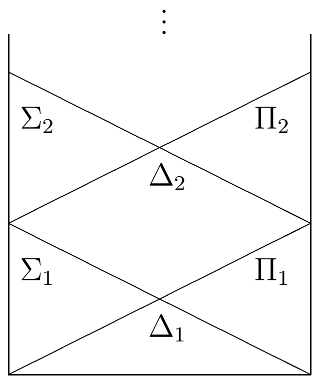
is universal Σ_{n+1}.

Proposition 8.21 *The universal Σ_n set is not Π_n.*

Proof Let U be the universal Σ_n set. Let $V = \{e : (e,e) \notin U\}$. If U were Π_n then V would be Σ_n. In that case there would be an e_0 such that $V = U_{e_0}$. But then

$$e_0 \in V \Leftrightarrow (e_0,e_0) \notin U \Leftrightarrow e_0 \notin U_{e_0} \Leftrightarrow e_0 \notin V.$$

Thus $\Sigma_n \supset \Delta_n$ and $\Pi_n \supset \Delta_n$. This gives the following picture of the arithemtic hierarchy.



$\vdots$
Σ_2
Π_2
Δ_2
Σ_1
Π_1
Δ_1