CS_226 Computability Theory
Lecture Notes

Anuj Dawar

September 1998

1 Introduction

1.1 What is Computability Theory?

Computability theory (or *theory of computation*) is a branch of the subject of *mathematical logic* that began about sixty years ago. Mathematical logic seeks to understand the *nature*, *method* and *limitations* of mathematical reasoning — that is, the foundations of mathematics. Computability theory is the study of those aspects of mathematical logic that involve *effective* computation — that is, computations that can be carried out, at least in principle, by some kind of computing *agent* in a *mechanical*, *deterministic* and *finitistic* way. The theory of computation is therefore *directly relevant* to the conceptual and technical problems of computer science.

Today, in the 1990s, there are many different types of computer and many different programming languages, and it is natural to ask:

1. "Can problem P be solved on machine M using language L?"
2. "Is the class of problems solvable by machine M with language L larger, smaller, or the same as, the class of problems solvable by machine M' with language L'?"
3. "Given that problem P can be solved on machine M using language L, can it be done so *efficiently*?"

In this course we study (1) and (2). Note that these questions ask whether or not tasks can be performed *at all*, irrespective of space and time requirements (and subjective qualities such as 'elegance'.)

The study of (3), namely *complexity theory*, is properly taught as a separate course although we will touch upon this subject if time permits.

1.2 Aims and Objectives

The aims of this course are:

1. To develop an appreciation for the limits of 'computation' as we understand the term today.

2. To become familiar with fundamental models of computation and the relationships between them.

To meet these aims, the objectives of this course are:

1. To study the URM and GKS models of computation and (time permitting) other fundamental models, as well as relationships among them.

2. To introduce undecidable problems (that is, non-computable functions) and methods for proving their undecidability.

3. To survey the historical developments in mathematical logic that gave rise to the subject of computability theory.

1.3 Historical Origins

The development of routines, procedures or algorithms for solving problems goes back to ancient history. Routines for solving mathematical problems were known to the Babylonians and ancient Greeks. (For example, *Euclid's Algorithm* for computing greatest common divisors, and *Eratosthene's Sieve* for computing prime numbers.) We mention below some basic motivating examples.

1.3.1 Geometric Construction

An early existence problem known to the Greeks was that of finding an algorithm for geometric construction known as 'squaring the circle'. Two thousand years later Lindemann *proved* that no such construction exists. This proof required an *exact theory* of geometric construction and precise statements concerning *what constructions were allowed.*

1.3.2 Mechanical Theorem Proving

As early as 1670, Leibnitz aspired to reduce mathematical argument to calculation — that is, to reducing the activity of proving mathematical facts to a mechanical, deterministic and finitistic procedure. This problem was still unresolved in 1928 when *David Hilbert* posed the *Entscheidungs Problem.* This is the (lesser) problem of finding an algorithm that calculates whether or not a given statement of arithmetic is true. (For example, on entering the input '$x + 2 > x$' the algorithm would announce 'true', whereas the input '$x \neq x$' would give 'false'.) In 1936 *Alonzo Church* and *Alan Turing* simultaneously, but independently, developed theories of effective computability that allowed them to prove that Leibnitz's dream was impossible.

1.3.3 Decision Problems

The Entscheidungs Problem is a very famous example of a *decision problem* ('entscheidung' is the German word for 'decision'). A decision problem is one where we are given a property π of some kind of object, and are then asked for any object x "Is it true or false that π holds of x?" Here it is important that π must be such that it must be either true or false of any object of the type in question. For example, a property of the natural numbers is "x is a prime number" — this is either true or false of every natural number.

Much computability theory is devoted to studying the nature of decision problems — in particular it is concerned with whether or not there is an algorithm that solves a given decision problem. Typical decision problems in computer science include:

- *The Halting Problem.* "Does program P halt when executed on input x?"
- *The Equivalence Problem.* "Do programs P and Q produce the same output when executed on the same input?"
- *The Virus Problem.* "Does program P contain virus V?"

1.4 Organizing Concepts

The theory of computation provides organizing concepts that allow us to address the questions and problems above.

1.4.1 Machines and Languages

A basic concept is that of the *duality of machines and languages*:

- Every machine M executes programs written in its machine language — that is, M is no more than the set of programs that it can execute. Thus to study the class of problems solvable by a particular type of machine, it is sufficient to study the problems for which programs in the machine language can be written.
- Every language L determines a 'virtual machine' — that is, a machine capable of executing programs written only in L, or equivalently, a machine for which L is the machine language. Thus to study the set of problems that are solvable by programs in L, it is sufficient to study the set of problems solvable by the virtual machine determined by L.

In practice then, it is sufficient to consider solving a problem with a given machine *or* a given language — whether or not one studies a machine as opposed to a language is entirely a matter of choice (often this is determined by the nature of the problem at hand). In the light of these remarks, we can restate the basic questions (1) and (2) of Section 1.1 as:

1. "Can problem P be solved using machine M (or language L)?"

2. "Is the class of problems solvable by machine M (or in language L) larger, smaller, or the same as, the class of problems solvable by machine M' (or in language L')?"

1.4.2 Abstract Machines

In computability theory it is traditional to work with machines rather than languages. However, rather than work with complex actual machines such as IBM PCs, Macintoshes, Sun Workstations etc., the method is to work with one simple abstract model of a computer — one that captures the basic notion of 'a computer' with the minimum of fuss, and then to invoke *equivalence theorems* which state that no more can be done with the abstract model than with an actual machine. (See Section 1.4.4.) This equivalence means that to show a given problem P is *not* solvable with a given machine we only need to work with the abstract model — this is simpler than dealing with the complexities of an actual machine.

For convenience and familiarity, the abstract model we will work with is the *unlimited register machine* or URM for short — this is an idealized model of programming a traditional computer in assembly language.

1.4.3 Computable Functions

The third organizing concept provided by computability theory is that the 'problems' that we will mainly be concerned with are those that can be expressed as the problem of evaluating or computing a function on the natural numbers $\mathbf{N} = \{0, 1, 2, \ldots\}$. That is, given a function $f : \mathbf{N} \to \mathbf{N}$, is it possible to get a machine (that is, a URM) to return the value $f(n)$ whenever the user enters a value for n? When this is possible f is said to be a *(URM-) computable function.*

One way of thinking about this is that we are considering only computation over the algebra of the natural numbers, $\mathbb{N}$.

algebra	$\mathbb{N}$
carriers	$\mathbf{N}$
constants	$0 :\to \mathbf{N}$
operations	$s : \mathbf{N} \to \mathbf{N}$

1.4.4 The Church-Turing Thesis

The class of computable functions is a large class and contains all the functions of practical use in computer science that have been considered until now. However, *non*-computable functions do exist and so it is natural to wonder if the concept of a computable function as one that is computable by a URM is too limited. Perhaps by adding more powerful

programming constructs to URMs, or by using a completely different machine model, we could compute even more?

Experience has shown that this is not the case. Many other models have been proposed over the years such as: *Kleene schemes*, *λ-definable functions*, *μ-recursive functions*, *Post production systems*, and perhaps the most famous of them all, *Turing machines*. However, all these models have been shown to be equivalent to URMs in the sense that for each model:

1. any function that can be computed in the model can also be computed by a URM, and

2. any function that can be computed by a URM can also be computed in the model.

Thus it seems that the computable functions are indeed 'universal' in the sense that any computation we may wish to perform can be accomplished with a URM. This is the *Church-Turing thesis* (sometimes called 'Church's Thesis'):

The class of intuitively computable functions is exactly the class of URM-computable functions.

Note that the Church-Turing thesis can never be proved because of the use of the word 'intuitively'. However, it is generally considered true because of the following evidence:

1. The many independent proposals for precise formulations of the intuitive idea of computation have all been shown to be equivalent to URMs.

2. No-one has ever found a function that is 'intuitively computable' and yet not computable with a URM.

3. It seems clear that URM-computable functions are 'intuitively computable'.

1.5 Uncomputable Functions

It is possible to see that there are uncomputable functions without considering a specific model of computation. This in fact follows simply from general principles that must be true of any model.

To see this, we first review some definitions.

1.5.1 Functions

Given two sets A and B, $A \times B$ is the set of all pairs of elements (a, b), where a is an element of A and b is an element of B. The set $A \times B$ is called the *Cartesian product* of A and B. A *relation* between A and B is any subset $R \subseteq A \times B$.

A *function* f from A to B is a relation between A and B which satisfies the following two properties:

1. For every element $a \in A$, there is at least one element $b \in B$ such that the pair (a, b) is in f.

2. For every element $a \in A$, there is at most one element $b \in B$ such that the pair (a, b) is in f.

To say that f is a function from A to B, we write $f : A \to B$. If f satisfies the second of the two conditions above, but not the first, we say that f is a *partial function* from A to B, and write $f : A \hookrightarrow B$. In either case, we write $f(a) = b$ to denote the fact that the pair (a, b) is in f. If f is a partial function, and there is no b such that $f(a) = b$, we say that f is undefined at a. The set of elements at which f is defined is called the *domain* of f.

The set that we are most often interested in is the set of natural numbers $\mathbf{N} = \{0, 1, 2, 3, \ldots\}$.

Example 1 *1. The set of pairs (m, n) such that $m < n$ is a relation on $\mathbf{N}$. This relation is not a function since, for example, $2 < 3$ and $2 < 4$.*

2. The set of pairs $(m, m + 1)$ is a function from $\mathbf{N}$ to $\mathbf{N}$, because for each number m, there is exactly one number n such that $n = m + 1$. This is called the successor function.

3. The set of pairs $(m, m-1)$ is a partial function from $\mathbf{N}$ to $\mathbf{N}$, because for all numbers m other than 0, there is exactly one number n such that $n = m - 1$, but there is no such number for 0. This partial function is called the predecessor function, and it is undefined at 0. Its domain is $\mathbf{N} \setminus \{0\}$.

4. Consider the set of pairs $((m, n), m+n)$, where the first element is itself a pair (m, n) and the second element is the number $m + n$. This is a function from $\mathbf{N} \times \mathbf{N}$ to $\mathbf{N}$.

1.5.2 Counting and Countability

A function $f : A \to B$ is:

injective if whenever $f(a) = f(a')$, then $a = a'$.

surjective if for every $b \in B$, there is an $a \in A$ such that $f(a) = b$.

bijective if it is both injective and surjective.

To say that two sets have the same number of elements is to say that there is a bijection between the two sets. This is obvious for finite sets. For infinite sets, we take it as the definition for the idea of two sets having the same number of elements. This is justified, to some extent, by the following, which can be proved for arbitrary (finite or infinite) sets:

Theorem 1 *If there is an injective function $f : A \to B$ and an injective function $g : B \to A$, then there is a bijection $h : A \to B$.*

Any set A for which there is a bijection $f : A \to \mathbf{N}$ is said to be countably infinite. If A is finite, then there is an injection $f : A \to \mathbf{N}$ but no bijection. On the other hand, if A is infinite, it is clear that there is an injection from $\mathbf{N}$ to A ($\mathbf{N}$ is, in this sense, the "smallest" infinite set). If there is also an injection from A to $\mathbf{N}$ then by Theorem 1 A is countably infinite. On the other hand, if there is no such injection, then A is uncountable. Intuitively, it is "bigger" than any countably infinite set.

The existence of uncountable sets was proved by Cantor in the 1890's. He showed that while the set of rational numbers (that is, those numbers that can be represented as fractions) is countable, the set of real numbers (those that can be represented by decimal expansions) is not countable. In the next section we will prove, essentially using Cantor's techniques, the following two facts:

1. The set of finite strings in any finite alphabet is countable.

2. The set of functions from $\mathbf{N}$ to $\mathbf{N}$ is uncountable.

Since any programming language is ultimately a set of strings in a finite alphabet, it follows that in any language, there are functions that cannot be computed.

1.5.3 Countability of Programs and Functions

Suppose Σ is a finite collection of symbols (also called an alphabet). Σ^* is the collection of finite strings of symbols from Σ. Then, there is an injective function $f : \Sigma^* \to \mathbf{N}$.

In order to define such a function, first, let us number the symbols in Σ. Suppose the symbols are $s_1, s_2, \ldots, s_k$ in some order. We assign the number 1 to s_1, 2 to s_2 and so on. Now, we define the function $f : \Sigma^* \to \mathbf{N}$ as follows. For any string $\sigma \in \Sigma^*$ of length m:

$$f(\sigma) = p_1^{e_1} p_2^{e_2} \ldots p_m^{e_m}$$

where,

p_i is the ith prime number, i.e. p_1 is 2, p_2 is 3, p_3 is 5 and so on; and

e_i is the number corresponding to the ith symbol in the string σ.

For example, suppose Σ is the two symbol alphabet $\{a, b\}$. We assign the number 1 to a and the number 2 to b. The string $aabab$ is then mapped by f to the number

$$2^1 \times 3^1 \times 5^2 \times 7^1 \times 11^2 = 127050.$$

Proposition 2 *The function f is injective.*

Proof. We now need to show that f is indeed injective. For this, recall the fact that every positive natural number has a *unique* prime decomposition. That is, for every x there exist numbers k and $y_1, \ldots, y_k$ such that $x = p_1^{y_0} p_2^{y_1} \cdots p_k^{y_k}$. For example,

$$220 = 2^4 \times 3^0 \times 5^1 \times 7^0 \times 11^1.$$

Moreover the numbers k and $y_1, \ldots, y_k$ are unique. For a given x, there is only one way to decompose it as a product of primes.

Now, to see that the function f we defined is injective, we need to see for any natural number n, there is *at most* one string σ such that $f(\sigma) = n$. Suppose, for the sake of contradiction, that there is a number n and two *different* strings σ and σ' such that

$$f(\sigma) = f(\sigma') = n.$$

Since σ and σ' are distinct strings, there are two possibilities:
Case 1: σ and σ' are of different lengths, m and m'. This means that in the prime factorisation of $f(\sigma)$, the first m primes occur, while in the prime factorisation of $f(\sigma')$ the first m' primes occur. Since we know that every number has a *unique* factorisation, it follows that that it cannot be the case that $f(\sigma) = f(\sigma') = n$.
Case 2: σ and σ' are of the same length m, but there is some $i \leq m$ such that the ith symbol of σ is not the same as the ith symbol of σ'. This means that the prime p_i is raised to a different power in the prime factorisation of $f(\sigma)$ than it is in the factorisation of $f(\sigma')$. Once again, we know that n does not have two different factorisations, so it cannot be the case that $f(\sigma) = f(\sigma') = n$.

This completes the proof that f is injective. It is also fairly easy to see that there is an injective function $g : \mathbf{N} \to \Sigma^*$. For example we could define it by $g(n) = a^n$. Thus, by Theorem 1, it follows that there is, in fact, a bijection between Σ^* and $\mathbf{N}$.

Next, we will prove that the set $F = \{f | f : \mathbf{N} \to \mathbf{N}\}$ of all functions from $\mathbf{N}$ to $\mathbf{N}$ is not countable. That is, there is no bijection between the natural numbers $\mathbf{N}$ and the set of all functions . This proof is similar to the proof by diagonalisation that there is no bijection between the natural numbers and the real numbers.

Proposition 3 *The set F is uncountable.*

Let us suppose that there is a bijection $b : \mathbf{N} \to F$. This means that for every natural number n, $b(n)$ is a function from $\mathbf{N}$ to $\mathbf{N}$, and moreover for every function $f : \mathbf{N} \to \mathbf{N}$, there is a number m such that $b(m) = f$. For convenience, we will write the function $b(n)$ as b_n. Now we define a new function $d : \mathbf{N} \to \mathbf{N}$ by the following rule

$$d(n) = b_n(n) + 1.$$

But, this means that there cannot be a number m such that d is the same function as b_m. Because, for every number m, there is at least one value, namely m, on which the functions d and b_m differ. Hence, b cannot be a bijection.

It follows from Propositions 2 and 3 that there are more functions than there are strings in any finite alphabet. Since the set of valid programs in any programming language is a subset of the set of finite strings in some appropriate finite alphabet, we conclude that there are more functions than there are programs. Thus, there must be functions that are not computable by any program.

2 Unlimited Register Machines

An *unlimited register machine* or *URM* is an abstract model of a computer with three important qualities:

1. It captures the two essential features of the informal notion of an algorithm:
   (a) an algorithm comprises a finite number of steps or commands, each completed in a finite time using finite resources.
   (b) if an algorithm produces output then that output emerges after a finite number of steps.
2. It is a *formal* model — this allows us to make the notion of 'computable' precise.
3. It is a *simple* model — this allows us to be clear about what is and what is not 'computable'.

The URM model is due to J. C. Shepherdson and H. E. Sturgis and was first published as "Computability of Recursive Functions", *Journal of the ACM* 10, pp. 217–255, 1963.

2.1 The URM Model

The URM has an infinite number of *registers* $R_0, R_1, R_2, R_3, \ldots$ each of which can hold a natural number of arbitrary size. The register R_0 is a special register called the *program counter* and is usually denoted by 'PC'. Note that in what follows, when we say 'register R_n' we will usually mean that $n \neq 0$ — that is, the program counter is not normally considered to be a register.

A URM can be made to carry out a sequence of operations on the contents of its registers by means of a *URM program* p. Such a program is a non-empty list $p = (I_1, \ldots, I_l)$ where for $i = 1, \ldots, l$, each I_i is a *URM instruction* of one of the following kinds:

1. *Zero Instructions.* These are of the form `Z(`n`)` where $n \in \{1, 2, 3, \ldots\}$.
2. *Successor Instructions.* These are of the form `S(`n`)` where $n \in \{1, 2, 3, \ldots\}$.
3. *Jump Instructions.* These are of the form `J(`m, n, q`)` where $m, n, q \in \{1, 2, 3, \ldots\}$.

2.1.1 Informal Operational Semantics

To compute with a URM on some inputs $a_1, \ldots, a_r$ it is usual to first place a_i in register R_i for $i = 1, \ldots, r$, and to initialize $R_{r+1}, R_{r+2}, \ldots$ to 0 (although this is not always necessary). A URM executes a program $p = (I_1, \ldots, I_l)$ by executing the instruction currently determined by the program counter. We usually assume the PC is initialized to one since the first instruction to execute is normally I_1 (although again, this need not be the case). The URM increases the PC by one after it has executed an instruction — unless the instruction is a jump instruction as described below.

The intention behind the three kind of instruction is as follows. An instruction of the form `Z(`n`)` simply places a zero in R_n. The instruction `S(`n`)` adds one to the contents of R_n. The instruction `J(`m,n,q`)` is only slightly more complicated: the idea is that if R_m and R_n hold the same value then the URM jumps to instruction I_q (by setting PC $= q$), otherwise the URM simply continues with the next instruction (by setting PC $=$ PC $+ 1$).

Finally, if the value in PC ever exceeds the length of the program (so there is no instruction to execute) then the program terminates.

2.1.2 Examples

Example 2 *(Transfer.) The following program transfers the contents of R_1 to R_2:*

```
1. Z(2)              /* ensure R2 contains zero to start */
2. J(1,2,5)          /* if R1 = R2 then halt */
3. S(2)              /* increment R2 */
4. J(1,1,2)          /* jump back to test R2 again */
```

This program works by initially setting R_2 to zero, and then by repeatedly increasing R_2 until its contents equal the contents of R_1. This is illustrated in the table below where we have assumed that the PC initially contains 1 and R_1 has been loaded with the value 2. The initial values in registers $R_2, R_3, R_4, \ldots$ do not affect the computation but must be initialized to hold some definite natural number — we have arbitrarily chosen the value 99 for all such initial values.

Step	PC	R_1	R_2	R_3	R_4	$\cdots$
0	1	2	99	99	99	$\cdots$
1	2	2	0	99	99	$\cdots$
2	3	2	0	99	99	$\cdots$
3	4	2	1	99	99	$\cdots$
4	2	2	1	99	99	$\cdots$
5	3	2	1	99	99	$\cdots$
6	4	2	2	99	99	$\cdots$
7	2	2	2	99	99	$\cdots$
8	5	2	2	99	99	$\cdots$

The following examples show how to compute some simple functions using a URM. In each case you are encouraged to draw up your own table showing how the values in the registers change from step to step as was done above for the preceding example. Also, by working through the examples, convince yourself that in each case the machine always terminates irrespective of the initial values (provided that the PC always starts at 1). This is an important feature that we will return to in Section 2.1.3.

Example 3 *(Addition.) The following program computes $f : \mathbf{N}^2 \to \mathbf{N}$ defined by $f(a,b) = a + b$ for any $a, b \in \mathbf{N}$. The program assumes a and b are initially placed in registers R_1 and R_2 respectively, and the result is returned in R_3.*

```
1.  Z(3)        /* copy contents of R1 to R3 */
2.  J(1,3,5)
3.  S(3)
4.  J(1,1,2)

5.  Z(4)        /* initialise counter to zero */
6.  J(2,4,10)   /* if counter = b then R3 = a+b, so exit */
7.  S(3)        /* increase result by one */
8.  S(4)        /* increase counter */
9.  J(1,1,6)    /* jump back to test */
```

Example 4 *(The predecessor function.) The following program computes the predecessor function* $f : \mathbf{N} \to \mathbf{N}$ *defined by*

$$f(a) = \begin{cases} 0 & \text{if } a = 0 \\ a - 1 & \text{otherwise} \end{cases}$$

for any $a \in \mathbf{N}$*. The program assumes* a *is placed in* R_1*, and the result is placed in* R_2*.*

```
1. Z(2)           /* initialise result = 0 */
2. J(1,2,9)       /* special case a = 0 */
3. Z(3)           /* set counter = 1 */
4. S(3)
5. J(1,3,9)       /* if counter = a then predecessor is in R2 */
6. S(2)           /* increase potential result */
7. S(3)           /* increase counter */
8. J(1,1,5)       /* repeat test */
```

Example 5 *(Multiplication.) The following program computes* $f : \mathbf{N}^2 \to \mathbf{N}$ *defined by* $f(a, b) = a \times b$ *for any* $a, b \in \mathbf{N}$*. The program assumes* a *and* b *are initially placed in registers* R_1 *and* R_2 *respectively, and the result is returned in* R_3*.*

```
1.  Z(3)        /* initialise result */
2.  J(1,3,20)   /* special case a = 0 */
3.  Z(4)        /* set counter to 0 */
4.  S(4)        /* increment counter */

5.  Z(5)        /* R5 := R2 + R3 */
6.  J(2,5,9)
7.  S(5)
8.  J(1,1,6)

9.  Z(6)
10. J(3,6,14)
```

```
11.  S(5)
12.  S(6)
13.  J(1,1,10)

14.  Z(3)       /* copy contents of R5 to R3 */
15.  J(5,3,18)
16.  S(5)
17.  J(1,1,15)

18.  J(1,4,20)
19.  J(1,1,4)   /* repeat addition */
```

2.1.3 Computing Functions

It is apparent that programs of Examples 2 – 5 always terminate for any initial register values. This is especially appropriate for Examples 3 – 5, since each of these programs was intended to compute a *total* function. (Recall that a total function is one that is defined on all possible arguments.) However, it is important to note that in general a URM program is not guaranteed to terminate on all possible inputs. For example, the following program will terminate only on initial values of R_1 that are even.

```
1. Z(2)
2. Z(3)
3. J(1,3,8)
4. S(2)
5. S(3)
6. S(3)
7. J(1,1,3)
```

Thus, under the assumption that the result of this program (when it does terminate) is returned in R_2, there is an obvious sense in which the program computes the *partial* function $f : \mathbf{N} \hookrightarrow \mathbf{N}$ defined by

$$f(a) = \begin{cases} a/2 & \text{if } \exists b \in \mathbf{N}.a = 2b \\ \perp & \text{otherwise.} \end{cases}$$

(The symbol '$\perp$' means 'undefined'.) As another example, the one-line program

```
1. J(1,1,1)
```

will never terminate on any initial values — it can be said to compute the everywhere-undefined function $f(a) = \perp$ for all values of a.

It is no accident that there appears to be a correspondence between the definedness of functions and the termination of programs — this is a central idea in computability theory, to which we will return later.

2.1.4 Discussion of the Model

On encountering the URM model for the first time, it is common for people to query the power of the model. These queries arise in two ways depending on whether we are trying to show that a given problem can, or cannot, be solved with a URM.

Realism If we claim that a problem *can* be solved by means of a URM program then it is commonplace to hear objections of the form: "Is it realistic to allow the machine to have an *infinite* number of registers, or for a register to hold a number of an *arbitrary* size? I accept that it is possible to solve problem X when one is allowed an infinite memory that can hold arbitrarily large data, but this does not mean the problem can be solved on a machine with a finite memory with a bounded word-size".

Clearly, such questions reflect a concern over the practical usefulness of our theoretical work. We are interested in what can be computed *at least in principle*, so we do not wish to be concerned with technology-dependent features of computation such as the size of memory on a given machine, nor the machine's word-size.

However, our assumptions about the URM model are not as unrealistic as they may appear at first sight. With respect to the 'infinite memory', notice that all URM instructions have fixed, constant arguments so that there is no 'indexed addressing' — for example, it is not possible to have an instruction `Z(`n`)` where 'n' is either an expression or is computed by another part of the program. Thus a first theorem of URM computability is: *any URM program only uses or changes the value of finitely many registers.* In other words although a URM is described as having infinitely many registers, any *particular* URM program only ever uses finitely many of them. We say that the URM's memory is 'potentially infinite'.

It is important to note here that the maximum number of registers needed by any given program is simply calculated by examining the program *text*.

Moreover, we can make an even stronger statement. It can, in fact, be shown that our assumption of an infinte number of registers is unnecessary in a very precise sense. That is, for every URM program, there is another program computing the same function but using only two registers. In other words, if we had defined the URM model to have only two registers instead of an infinite number of them, we would still be able to compute all the functions that a URM can compute. However, proving this involves a tedious amount of coding and is not very instructive. We simply adopt the convention of an infinite number of registers with the knowledge that this does not increase the power of our model

The objection that URMs are unrealistic for the reason that a register can hold a number of any size is somewhat harder to counter. Clearly, we could assume that registers can only hold numbers up to some fixed size and when the number in a given register exceeds this value then we can use another register to hold the 'overflow'. Of course, as the number grows and grows, we have to use more and more registers, but *provided the computation always terminates* then there will be an upper bound on the number of registers used, so we do not violate the 'practicality requirement' that the memory is finite — the memory needs to be as large as the computation requires, but it does not need to be infinite.

Notice that in this 'restricted register' model any terminating computation will only use finitely many registers, but it may not always be possible to determine the number of registers required in advance of the computation (unlike the situation where registers are not restricted). Thus from a practical perspective, this means that the computation is possible provided we are prepared to keep adding memory as the computation proceeds.

Note also that when we speak of computing a function, we are talking about functions over the natural numbers. That is, the functions take arbitrarily large numbers as inputs. This is also not practical. Any given physical computer will be limited to some finite range of inputs. However, we are talking about what is computable *in principle* – a finite function is always computable, just by encoding the entire function in a table. Another way of saying this is, when we ask whether a function is computable, we mean is there an algorithm that computes the function, i.e. given sufficient physical resources, the algorithm will produce the correct output for any input.

It is also possible that a computation never terminates which means that registers of arbitrary capacity (or a truly infinite memory) are necessary in general. This leaves us with two choices: (1) we disallow non-terminating computations, or (2) we allow arbitrary capacity registers. From some points of view, it is natural to want to choose the former on the grounds that a program that never terminates could not be of any conceivable use. However, quite apart from the fact that intentionally non-terminating programs *can* be useful (operating systems, for example), it is known (and we will prove) that there is no effective procedure for determining if a given program terminates on a given input or not — thus there is a strong sense in which we cannot exclude non-terminating programs even if we wished to do so.

Lack of Constructs The second kind of objection to the URM model arises in the context of problems that can *not* be solved. These objections take the form: "I accept that it is impossible to solve problem X even when one is allowed an unbounded memory that can hold arbitrarily large data, but surely this is because your model of computation does not involve Y" — where 'Y' is typically some specific type of data and operations on that data such as negative numbers and the subtraction operation, or programming constructs such as 'arrays', 'pointers', 'recursion', or 'procedures'.

Of course, it is natural to want our model to have sufficiently many programming constructs and data types to ensure that everyday computations can be accomplished within the model — it would hardly be surprising that the Halting Problem was unsolvable if in fact we could not even write a program to add two numbers together!

In this course we will restrict our attention to computation on the natural numbers — so-called *classical computability theory* — for two reasons. First, all of the basic data types such as Booleans, negative and floating-point numbers that are found in normal computer programming can be simulated using natural numbers, as discussed in the next section — thus there is a sense in which these data types are not essential. Second, *generalized computability theory* where natural numbers are not the only type of data is an advanced topic for which classical computability is a prerequisite.

With respect to other features of modern programming languages that are not present in the URM model, we will show that they are unnecessary (though no doubt convenient for practical programming). The advantage of not including such features lies in the fact that our computing formalism is extremely small, so if we have to show that a program does not exist then there are relatively few cases to consider since any such program can only be made of three different types of instruction.

2.1.5 Complex Constructs

In the simple examples of Section 2.1.2, we have already seen how simple URM programs can be combined to form more complex ones.

We can make this notion precise through the use of pseudo-instructions. Consider the program of Example 1, which transfers the contents of register R_1 to register R_2. We can denote this program `T(1,2)`. Indeed, we can generalize this to `T(`m, n`)` – a program that copies the contents of R_m into R_n. `T(`m, n`)` is obtained from `T(1,2)` by replacing all occurrences of 1 that appear as arguments to `Z` or `S` or in the first two argument places of `J` by m, and similarly replacing 2 by n.

We can now add the instruction `T` to our basic instruction set, secure in the knowledge that the resulting program can still be translated into a valid URM program (note that when we perform the translation, we must be careful that the references to intsruction numbers in the definition of `T(`m, n`)` are only relative offsets). Thus, we can rewrite the addition program from Example 2 by replacing the first four lines with the instuction `T(1,3)`.

Similarly, having written a program to perform addition, we can define the instruction `Add(`m, n, p, t`)` which places the sum of the numbers in R_m and R_n in the register R_p, using the register R_t for temporary storage. We need to specify this, as the addition program uses four registers. If we are to use the `Add` subroutine as a pseudo-instruction, we need to ensure that this register use does not conflict with other registers used in the program.

Moreover, we have repeatedly used a pattern of the following form in our examples:

```
        Z(m)
loop:   J(m', m, end)
        P
        S(m)
        J(m', m', loop)
end:
```

This can be written, instead, as:
for $R_m := 0$ **to** $R_{m'} - 1$ **do** P **od**.

Using such constructs, we can now rewrite the multiplication program of Example 4 as:

`Z(3)`

`J(1,3,end)`

for $R_4 := 0$ **to** $R_1 - 1$ **do**

`Add(2,3,5,6)`

`T(5,3)` **od**.

end:

This principle of pseudo-instructions can be further extended, and we could develop URMs into a full-fledged programming language. Looking at it another way, we could start with a programming language like Pascal and define a way of *compiling* Pascal programs into URMs. We have already seen how to do this for some simple arithmetic functions and for **for** loops.

One sticky point that remains is that URMs have only one data type – the natural numbers. A language like Pascal provides many other datatypes. However, any of these data types can be encoded into the natural numbers.

In general, a data type is given by a many-sorted algebra A, in some signature Σ:

algebra	A
carriers	$\ldots, A_s, \ldots$
constants	$\vdots$ $c_A \to A_s$ $\vdots$
operations	$\vdots$ $\sigma_A : A_{s(1)}, \ldots, A_{s(n)} \to A_s$ $\vdots$

An encoding of A into the natural numbers is a Σ-algebra A':

algebra	A'
carriers	$\ldots, A'_s, \ldots$
constants	$\vdots$ $c_{A'} \to A'_s$ $\vdots$
operations	$\vdots$ $\sigma_{A'} : A'_{s(1)}, \ldots, A'_{s(n)} \to A'_s$ $\vdots$

along with an isomorphism, $\iota : A \to A'$, with the property that the carrier sets, A'_s are pairwise *disjoint* subsets of $\mathbf{N}$.

As an example, consider the algebra of the natural numbers with Booleans:

signature	Σ
carriers	$\mathbf{N}, \mathrm{Bool}$
constants	$0 : \ \to \mathbf{N}$ $T, F : \ \to \mathrm{Bool}$
operations	$+ : \ \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ $\times : \ \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ $=: \ \mathbf{N} \times \mathbf{N} \to \mathrm{Bool}$ $\neg : \ \mathrm{Bool} \to \mathrm{Bool}$ $\& : \ \mathrm{Bool} \times \mathrm{Bool} \to \mathrm{Bool}$ $\vee : \ \mathrm{Bool} \times \mathrm{Bool} \to \mathrm{Bool}$

This can be encoded into the natural numbers by taking the carrier sets $\mathbf{N}' = \mathbf{N} - \{0, 1\}$, $\mathrm{Bool}' = \{0, 1\}$. We then have to define the interpretation of the constants and functions.

For instance:

$$
\begin{aligned}
&0' = 2, T = 1, F = 0 \\
&+'(m,n) = (m-2) + (n-2) + 2 \\
&\times'(m,n) = (m-2) \times (n-2) + 2 \\
&='(m,n) = \begin{cases} 1 & \text{if } m = n \\ 0 & \text{otherwise} \end{cases} \\
&\neg'(x) = 1 - x \\
&\&'(x,y) = x \cdot y \\
&\vee'(x,y) = 1 - ((1-x) \cdot (1-y))
\end{aligned}
$$

Using this encoding, it is not difficult to write URM programs for evaluating any Boolean or integer expression. Indeed, one can go on from there to show that an arbitrary **while** program (see J.V. Tucker's *Data, Programs and Correctness*) can be translated into a URM.

It is conceptually a small step from there (though a very tedious one to carry out in full detail) to prove that any Pascal program can be translated into a URM. We can write a compiler that would take Pascal programs as input and produce the corresponding URM as output. On the other hand, it is also clear that one can start with a URM and construct an equivalent Pascal program. Thus, the two models are, in fact equivalent. Any function that can be computed by a Pascal program can be computed by a URM and vice versa. The accumulation of such results for a large number of programming systems is what led to the formulation of the Church-Turing thesis.

2.2 The Computable Functions

As was stated in Section 1.4.3, the main use of URMs is to compute functions on $\mathbf{N}$. It is therefore natural to ask, if we are given a URM program P, what function does P compute? In order to be able to answer questions of this form, we must first adopt some conventions about what constitute the inputs to a program, where a program places its output and so on.

For our purposes, we will take the following definition as our convention:

Definition 4 *The (possibly partial) function $f : \mathbf{N}^k \hookrightarrow \mathbf{N}$, computed by a URM P is defined to be the function such that $f(m_1, , \ldots, , m_k) = n$ if, and only if, when the URM program P is started with m_1 in register R_1, m_2 in register R_2 and so on until m_k in register R_k, with 0 in all other registers, then the program eventually terminates with n in register R_1.*

There are a few remarks that need to be made about this definition:

First, note that the choice of R_1 as the register that holds the output is arbitrary. We have to choose some register beforehand, it may as well be the first. With this choice, the function $f : \mathbf{N}^2 \to \mathbf{N}$ computed by the addition program we wrote (Example 3) is not $f(m, n) = m + n$ at all, but $f(m, n) = m$! So, while the program performs the action $R_3 := R_1 + R_2$, the function it computes according to Definition 4 is another question altogether. However, this is merely an inconvenience, not a fact that changes the class of computable functions. In order to recover a program that computes the addition function in the sense of Definition 4, we just need to add to the end of Example 3 four lines that transfer the contents of register R_3 to R_1.

The second remark is that the same program P computes different functions as we vary our choice of k in Definition 4. So, if we write a program for computing the addition function by modifying Example 3 as explained above, we can say the following about the resulting program P:

1. The function of one input $f : \mathbf{N} \to \mathbf{N}$ computed by P is $f(n) = n$.
2. The function of two inputs $g : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ computed by P is $f(m, n) = m + n$.
3. The function of three inputs $h : \mathbf{N}^3 \to \mathbf{N}$ computed by P is $h(m, n, p) = m + n$.
4. and so on.

So, when we speak of the function computed by a program, it is important to specify the number of inputs intended, otherwise there is not a unique function.

Having adopted a convention which allows us to speak of functions computed by a program, we can now start to investigate the class of functions that are computable. We will proceed in the following way. We will first show that some very simple functions are computable. We will then show that certain ways of combining functions together to form more complex ones preserve the property of computability. In other words, we will see that we can combine programs already defined in certain fixed ways to get new functions.

These will allow us to show a whole range of functions to be computable without having to explicitly write programs for each one.

2.2.1 Basic Functions

Lemma 5 *For each $k > 0$, the function $\zeta_k : \mathbf{N}^k \to \mathbf{N}$ defined by $\zeta_k(a) = 0$ for all $a = (a_1, \ldots, a_k) \in \mathbf{N}^k$ is computable.*

Proof. Let P be the URM program

```
1. Z(1)
```

For every k, the k-ary function computed by P is exactly the function ζ_k.

Lemma 6 *The function $\sigma : \mathbf{N} \to \mathbf{N}$ given by $\sigma(n) = n + 1$ is computable.*

Proof σ is the unary function computed by the program:

```
1. S(1)
```

Lemma 7 *For each $n > 0$ and each i satisfying $1 \leq i \leq k$, the* projection function *$\pi_i^k : \mathbf{N}^k \to \mathbf{N}$ defined by $\pi_i^k(a) = a_i$ for all $a = (a_1, \ldots, a_k) \in \mathbf{N}^k$ is computable*

Proof. For each $k \geq 1$ and each i with $1 \leq i \leq k$, let $P_{k,i}$ be the program

```
1. Z(k+1)
2. J(i,k+1,5)
3. S(k+1)
4. J(1,1,2)
5. Z(1)
6. J(k+1,1,9)
7. S(1)
8. J(1,1,6)
```

The idea behind this program is to first copy the contents of R_i to the first register that is not used for the input data (R_{k+1}), and then copy that value into R_1. (An alternative strategy is to copy R_i directly into R_1, but this needs the case $i = 1$ to be treated as a special case — the program given here works for all values of i.) ⋈

2.2.2 Closure Properties

In this section we begin to look at closure properties of the class of computable functions. When we say that some set is closed under an operation, what we mean is that when we take some elements from the set and apply the operation to them to get a new object, it is also in the set. For instance, the set of natural numbers is closed under the operation of addition because when we add two natural numbers together the result is also a natural

number. Similarly, the set of natural numbers is closed under multiplication but not under subtraction. The set of integers is closed under addition, multiplication and subtraction but not under division. The set of rational numbers is closed under all of the above operations, but not under taking square roots. All these are closure properties of the respective sets.

Here, we are concerned with closure properties of the set of computable functions. So, the operations we will consider are operations that take functions already given and define new ones. One of the most basic such operations on functions is composition. Given two functions $\gamma : \mathbf{N} \to \mathbf{N}$ and $\beta : \mathbf{N} \to \mathbf{N}$, the composition of the two $\alpha = \gamma \circ \beta$ is a function $\alpha : \mathbf{N} \to \mathbf{N}$ given by

$$\alpha(n) = \gamma(\beta(n)).$$

We are going to consider a more general form of composition, where β is a m-ary function, and is composed simultaneously with m different l-ary functions.

Definition 8 *So, suppose that $\gamma : \mathbf{N}^k \hookrightarrow \mathbf{N}$, $\beta_1, \ldots, \beta_k : \mathbf{N}^l \hookrightarrow \mathbf{N}$ are (possibly partial) functions. The function $\alpha : \mathbf{N}^l \hookrightarrow \mathbf{N}$ defined by*

$$\alpha(a) = \gamma(\beta_1(a), , \ldots, , \beta_k(a)),$$

is obtained from γ and $\beta_1, \ldots, \beta_k$ by composition.

Note that α is defined for a particular input a if, and only if, each of the functions $\beta_1, \ldots, \beta_k$ is defined at a, and the function γ is defined at the input $(\beta_1(a), , \ldots, , \beta_k(a))$. In particular, if all of the functions γ and $\beta_1, , \ldots, , \beta_k$ are total, then so is α.

We are now in a position to prove that the class of computable functions is closed under composition. That is:

Theorem 9 *If γ is computable and $\beta_1, \ldots, \beta_k$ are all computable, and α is obtained from them by composition, then α is also computable.*

Proof: By assumption, there exist programs p_γ, $p_{\beta_1}, \ldots, p_{\beta_k}$ that compute γ, $\beta_1, \ldots, \beta_k$ respectively. We use these programs to construct a program p_α for α as follows. Note that in what follows we will assume initially that γ, $\beta_1, \ldots, \beta_k$ (and hence α) are total functions. The case where some of these functions are partial will be considered later.

We have to compute $\alpha(a)$ for any arguments $a = (a_1, \ldots, a_l) \in \mathbf{N}^n$. However, on recalling that $\alpha(a)$ is defined by

$$\alpha(a) = \gamma(\beta_1(a), \ldots, \beta_k(a))$$

it is clear that $\alpha(a)$ can be computed by first computing $b_1 = \beta_1(a), \ldots, b_k = \beta_k(a)$, and then computing $\gamma(b_1, \ldots, b_k)$. In more detail, the computation must begin in a state where $R_1, \ldots, R_l$ hold $a_1, \ldots, a_l$ respectively, and all other registers are initialised to zero. If we now run p_{β_1} on this input data then after some number of steps, the program will terminate with $b_1 = \beta_1(a)$ in R_1. The difficulty with this direct approach is that we next need to run p_{β_2} on the same input to compute $b_2 = \beta_2(a)$, but in running p_{β_1} we may over-write the initial values of $R_1, \ldots, R_l$. Thus our strategy is to

1. first make a copy of the initial values in $R_1, \ldots, R_l$ in some registers that are not used by any of the programs $p_{\beta_1}, \ldots, p_{\beta_k}$,

2. run p_{β_1},

3. for each of the programs $p_{\beta_2}, \ldots, p_{\beta_k}$, we copy the input values back into $R_1, \ldots, R_l$ before running the program.

Note that in copying the input back we will over-write the results of running $p_{\beta_1}, \ldots, p_{\beta_k}$ since these program return their results in R_1 — thus before we copy the input back into $R_1, \ldots, R_l$, we must make a copy of R_1 in further registers that are otherwise unused.

In fact, there is another detail to be considered. When we say that a program computes a function, this means that the program delivers the right result when the arguments to the function are placed in $R_1, \ldots, R_l$ *and all other registers contain 0.* (Recall Definition 4) Since this will not be true for p_{β_2} after having run p_{β_1}, we will need to explicitly re-initialise all the registers used by p_{β_1} before we run p_{β_2}.

To make this more precise, observe that any URM program p only uses finitely many registers — in general, let $r_p \geq 1$ be such that R_{r_p} is the highest-numbered register used in p. If we now let

$$r = 1 + max\{r_0, r_1, \ldots, r_k\}$$

where $r_0 = r_{p_\gamma}$ and $r_i = r_{p_{\beta_i}}$ for $i = 1, \ldots, m$, then none of $R_r, R_{r+1}, R_{r+2}, \ldots$ are used by any of p_γ, $p_{\beta_1}, \ldots, p_{\beta_k}$. Thus we can use $R_r, \ldots, R_{r+l-1}$ to hold the copy of the input, and registers $R_{r+l}, \ldots, R_{r+l+k-1}$ to hold $b_1, \ldots, b_k$ respectively. We must also set registers $R_{l+1}, \ldots, R_{r_1-1}$ to zero before executing program $p_{\beta_{i+1}}$ for $i = 1, \ldots, k-1$.

Under these conditions, all we need to do to compute $\alpha(a) = \gamma(b_1, \ldots, b_k)$ is to copy $R_{r+l}, \ldots, R_{r+l+k-1}$ into $R_1, \ldots, R_l$, set $R_{l+1}, \ldots, R_{r_0-1}$ to zero, and then run p_γ — this returns its result in R_1 which is precisely where it needs to be to say we have computed $\alpha(a)$.

To make this account of computing α more formal, we need to manufacture an appropriate program p_α. To do this, we will use the pseudo-instruction `T(`m,n`)` — recall that this instruction can be compiled out of a program using it by replacing occurrences of `T(`m,n`)` with a true URM program. We will also use two further pseudo-instructions both of which can be eliminated by replacing them with true URM code. First let `C(`x,y,z`)` denote a new instruction whose purpose is to copy $R_x, \ldots, R_{x+y-1}$ to $R_z, \ldots, R_{z+y-1}$ for any $x, y, z \geq 1$. It is straightforward to show that this instruction can be implemented by a URM program C although we will not go into details here. Second, let the pseudo-instruction `B(`m,n`)` denote the operation of setting (or 'blanking') $R_m, \ldots, R_n$ to zero.

Using these pseudo-instructions a suitable program p_α can be defined by

```
C(1,l,r)
beta1
T(1,r+l)
C(r,l,1)
B(l+1,r1-1)
```

```
beta2
T(1,r+l+1)
C(r,l,1)
B(l+1,r2-1)
...
C(r,l,1)
B(l+1,rk-1)
betak
T(1,r+l+k-1)
C(r+l,k,1)
B(k+1,r0-1)
gamma
```

This program uses programs `gamma`, `beta1`, ..., `betak` which are modified versions of p_γ, $p_{\beta_1}, \ldots, p_{\beta_k}$ respectively that we need to use so that the jump instructions of the original programs are unaffected by their position in p_α. For example, `beta1` is p_{β_1} modified where necessary so that the jump instructions in p_{β_1} are unaffected by the preliminary block transfer instruction. More precisely, if the length of the program for the `C`-instruction is l_C (say) then every instruction `J`(m, n, q) in p_{β_1} becomes `J`$(m, n, q + l_C)$ in `beta1`. Similarly p_{β_2} must be modified to take care of the preliminary `C`-instruction, the program `beta1`, a transfer instruction, a further `C`-instruction, and a `B`-instruction. If `beta2` is this modified program then every jump instruction `J`(m, n, q) in p_{β_2} needs to be `J`$(m, n, q + 2l_C + l_1 + l_T + l_B)$ in `beta2`, where l_1 is the length of `beta1`, l_T is the length of the program for the transfer instruction and l_B is the length of the program for the `B`-instruction. $\bowtie$

Before we look at examples of computable functions that are obtained from other computable functions by composition, it will be useful to consider another closure property of the class of computable functions, and that is that this class is closed under definitions by recursion. To formalise this notion, consider the typical case of a function defined by recursion. The factorial function $fact : \mathbf{N} \to \mathbf{N}$ is given by the following recursive definition:

$$\begin{aligned} fact(0) &= 0 \\ fact(n+1) &= (n+1) \times fact(n) \end{aligned}$$

In general, a recursive definition of a function $r : \mathbf{N} \to \mathbf{N}$ is obtained by giving a value (i.e. a constant) for $r(0)$, and specifying a function $i : \mathbf{N}^2 \to \mathbf{N}$ such that $r(n+1) = i(n, r(n))$. In the case of $fact$, the function i is specified by $i(a, b) = (a + 1) \times b$.

Still more generally, we can define a function of more than one input, by recursion. However, we only allow one of the inputs to be used for the purpose of recursion. So, in the base case, instead of a constant we have a function of all the other inputs, and in the recursive step we also have a function that takes into account other inputs. Formally, we have the following definition:

Definition 10 *Suppose that* $\beta : \mathbf{N}^k \hookrightarrow \mathbf{N}$ *and* $\gamma : \mathbf{N}^{k+2} \hookrightarrow \mathbf{N}$ *are (possibly partial)*

functions. The function $\alpha : \mathbf{N}^{k+1} \hookrightarrow \mathbf{N}$ *defined by:*

$$\begin{aligned} \alpha(a_1, , \ldots, , a_k, 0) &= \beta(a_1, , \ldots, , a_k) \\ \alpha(a_1, , \ldots, , a_k, n+1) &= \gamma(a_1, , \ldots, , a_k, n, \alpha(a_1, , \ldots, , a_k, n)) \end{aligned}$$

is obtained from β *and* γ *by primitive recursion.*

Once again, if the functions β and γ are total, it is easy to see that α must be total as well. If not, then the definedness of α on a particular input depends upon where β and γ are defined.

We are now ready to prove the second important closure property of the class of computable functions:

Theorem 11 *If* β *and* γ *are computable, and* α *is obtained from them by primitive recursion, then* α *is also computable*

Proof: By hypothesis there exist URM programs p_β and p_γ that compute β and γ respectively. We use these programs to construct a program p_α for α as follows. Similar to the case of the composition above, we will assume initially that β and γ are total functions.

To understand how to compute α, first recall how the function is defined in this case:

$$\begin{aligned} \alpha(a, 0) &= \beta(a) \\ \alpha(a, b+1) &= \gamma(a, b, \alpha(a, b)) \end{aligned}$$

for each $a = (a_1, \ldots, a_k) \in \mathbf{N}^k$ and each $b \in \mathbf{N}$. Notice that we can evaluate $\alpha(a, b)$ for any value of b in a 'bottom-up' style — that is, we can first compute $\alpha(a, 0)$, and then use this to compute $\alpha(a, 1)$, and then use this value to compute $\alpha(a, 2)$, and so on until we reach $\alpha(a, b)$. The following pseudo-code computes $\alpha(a, b)$ in exactly this way:

```
result := beta(a)
counter := 0
while counter < b do
  result := gamma(a,counter,result)
  counter := counter + 1
od
```

We can turn this code into a URM program p_α if we first decide on register allocation. Let R_{r_1} and R_{r_2} be the highest-numbered registers in p_β and p_γ respectively, and let $r = 1 + max\{r_1, r_2\}$. We can use $R_r, \ldots, R_{r+n}$ to hold a copy of the input similar to the case of composition, but note here that there are $k+1$ (and not k) inputs to be stored: $a_1, \ldots, a_k$ and b. Notice in particular that this means b is stored in R_{r+k}. We also need a register to play the rôle of '`counter`' in the pseudo-code — we will use R_{r+k+1} for this.

To compute $\alpha(a, b)$ from an initial state where $R_1, \ldots, R_{k+1}$ hold $a_1, \ldots, a_k, b$ respectively, we first make a copy of all the inputs in registers $R_r, \ldots, R_{r+k}$ for safe-keeping. At this point the machine is in exactly the right configuration to compute $\beta(a)$, except insofar

as p_β (being a program for a function of only k arguments) will expect R_{k+1} to contain zero. Thus if we initialise R_{k+1} to zero and then run p_β, we will obtain $\beta(a) = \alpha(a, 0)$ in R_1. If b is in fact 0, then there is nothing left to do — we have successfully computed $\alpha(a, b)$.

If b is non-zero then we must build up the value of $\alpha(a, b)$ by repeated execution of p_γ on the appropriate arguments. In general these arguments need to be: $a_1, \ldots, a_k$ in registers $R_1, \ldots, R_k$, the value of the counter in R_{k+1}, and the result of the previous execution of p_γ (or of the initial execution of p_β) in R_{k+2}. We must also ensure that $R_{k+3}, \ldots, R_{r_2}$ contain zero each time we execute p_γ.

A suitable program p_α is shown below. As in the case of composition above, we have used the pseudo-instructions `T(`m, n`)`, `C(`x, y, z`)` and `B(`m, n`)`, and we assume p_β and p_γ can be transformed appropriately with respect to jump instructions into the programs `beta` and `gamma` respectively.

```
     C(1,n+1,r)
     B(n+1)
     beta
     Z(r+n+1)
test J(r+n,r+n+1,end)
     T(1,n+2)
     C(r,n,1)
     T(r+n+1,n+1)
     B(n+3,r2)
     gamma
     S(r+n+1)
     J(1,1,test)
end
```

2.2.3 Building Computable Functions

Using the basic functions (the zero functions – ζ_k, the successor function – σ and the projection functions – π_i^k) along with the closure properties we proved in the last section, we can now begin to show that a large number of functions are computable, without having to explicitly give the URM programs that compute them. We do this by showing that these functions can be obtained by applications of the operations of composition and primitive recursion to the basic functions, or functions that have already been shown to be computable. In this section, we will begin to build up a library of computable functions by this means.

Example 6 *For each $k, c \in \mathbf{N}$, define the constant functions $K_c^k : \mathbf{N}^k \to \mathbf{N}$ by*

$$K_c^k(n_1, \ldots, n_k) = c.$$

Each of these functions is computable, because $K_0^k = \zeta_0$, and K_{c+1}^k is obtained from K_c^k and σ by composition.

Recall the predecessor function from Example 4:

Example 7 *Define the predecessor function* $\mathcal{P} : \mathbf{N} \to \mathbf{N}$ *as:*

$$\mathcal{P}(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

We have already constructed a URM program that computes this function. We now show that it can be defined by primitive recursion from ζ_0 *and* π_1^2*.*

$$\begin{aligned} \mathcal{P}(0) &= \zeta_0 \\ \mathcal{P}(n+1) &= \pi_1^2(n, \mathcal{P}(n)) \end{aligned}$$

Example 8 *The addition function is defined by primitive recursion as follows:*

$$\begin{aligned} \mathrm{add}(m, 0) &= \pi_1^1(m) \\ \mathrm{add}(m, n+1) &= \sigma(\pi_3^3(m, n, \mathrm{add}(m, n)). \end{aligned}$$

We can not define subtraction similarly, since it is not a total function on the natural numbers. But we can approximate it with the following function:

Example 9 *The dotminus function is defined as*

$$m \dot{-} n = \begin{cases} 0 & \text{if } m < n \\ m - n & \text{otherwise} \end{cases}$$

This function is primitive recursive by the following definition.

$$\begin{aligned} m \dot{-} 0 &= \pi_1^1(m) \\ m \dot{-} (n+1) &= \mathcal{P}(\pi_3^3(m, n, m \dot{-} n)). \end{aligned}$$

Example 10 *Multiplication can be defined as follows:*

$$\begin{aligned} \mathrm{mult}(m, 0) &= \zeta_1(m) \\ \mathrm{mult}(m, n+1) &= \mathrm{add}(\pi_1^3(m, n, \mathrm{mult}(m, n)), \pi_3^3(m, n, \mathrm{mult}(m, n)). \end{aligned}$$

Example 11 *Exponentiation is defined by primitive recursion over multiplication.*

$$\begin{aligned} \exp(m, 0) &= K_1^1(m) \\ \exp(m, n+1) &= \mathrm{mult}(\pi_1^3(m, n, \exp(m, n)), \pi_3^3(m, n, \exp(m, n)). \end{aligned}$$

Example 12 *Similarly, the factorial function can be defined by:*

$$\begin{aligned} \mathrm{fact}(0) &= K_1^0 \\ \mathrm{fact}(n+1) &= \mathrm{mult}(\sigma(\pi_1^2(n, \mathrm{fact}(n))), \pi_2^2(n, \mathrm{fact}(n)). \end{aligned}$$

Example 13 *Define the sign function as*

$$\mathrm{sg}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{otherwise} \end{cases}$$

This function is computable, since:

$$\begin{aligned} \mathrm{sg}(0) &= \zeta_0 \\ \mathrm{sg}(n+1) &= K_1^2(n, \mathrm{sg}(n)) \end{aligned}$$

3 Recursive Functions

In the last section, we saw that the set of computable functions is closed under the operations of composing functions and of forming functions by primitive recursion. More significantly, we saw that quite a few useful functions can be defined by starting off with just the basic functions (the zero, successor and projection functions) and repeatedly applying the operations of composition and primitive recursion. This suggests a new approach of trying to build up the computable functions "from below."

If we consider all the possible functions that can be obtained by repeated applications of composition and primitive recursion, starting with the basic functions, what is the set of functions we get? By the results of the last section, we know that any function we can get in this way is computable, so we know that the set of such functions is a subset of the set of computable functions. In fact, it is also a proper subset, since there are computable functions that are not total, while any function that can be obtained from the basic functions by composition and primitive recursion must be total. This is because the basic functions are all total, and we saw that the operations of composition and primitive recursion preserve totality. What's more, we can even show that there are *total* computable functions that can not be obtained from the basic functions through the operations of composition and primitive recursion (we will see an example of such a function in Section 3.2).

Nevertheless, the set of functions that is obtained by taking the basic functions and then closing them under composition and primitive recursion turns out to be an interesting set of functions in its own right. This is called the set of primitive recursive functions and includes almost any total function that you may wish to compute in practice.

3.1 Primitive Recursive Functions

We begin with a few definitions.

Definition 12 *The* basic functions *are the functions* $\zeta_k (k \geq 0)$, σ *and* $\pi_i^k (k \geq i \geq 0)$.

Definition 13 *The set of functions that can be obtained from the basic functions by a finite number of applications of the operation of composition is called the set of* polynomial functions.

Equivalently, the set of polynomial functions is the smallest set of functions containing the basic functions and closed under the operation of composition.

Definition 14 *The set of functions that can be obtained from the basic functions by a finite number of applications of the operations of composition and primitive recursion is called the set of* primitive recursive functions.

Equivalently, the set of polynomial functions is the smallest set of functions containing the basic functions and closed under the operations of composition and primitive recursion.

Writing $POLY_k$ to denote the set of all polynomial functions of k arguments, $PRIM_k$ for the set of all primitive recursive functions of k arguments, $COMP_k$ for the set of all computable functions of k arguments, and FN_k for the set of all (partial) functions of k arguments, we have the following inclusions:

$$POLY_k \subseteq PRIM_k \subseteq COMP_k \subseteq FN_k.$$

Each of these inclusions is strict. That is, we know that there are functions that are not computable, that there are computable functions that are not primitive recursive and it can also be shown that there are primitive recursive functions that are not polynomial.

The examples in Section 2.2.3 establish the following:

Theorem 15 *Each of the following functions is primitive recursive:*

1. *for each $k \geq 0$ and each $c \geq 0$, the constant function $K_c^k : \mathbf{N}^k \to \mathbf{N}$;*
2. *the predecessor function $\mathcal{P} : \mathbf{N} \to \mathbf{N}$;*
3. *the addition function* add $: \mathbf{N}^2 \to \mathbf{N}$;
4. *the dotminus function* $\dot{-}: \mathbf{N}^2 \to \mathbf{N}$;
5. *the multiplication function* mult $: \mathbf{N}^2 \to \mathbf{N}$;
6. *the exponentiation function* exp $: \mathbf{N}^2 \to \mathbf{N}$;
7. *the factorial function* fact $: \mathbf{N} \to \mathbf{N}$; *and*
8. *the sign function* sg $: \mathbf{N} \to \mathbf{N}$.

We can now start building further on these and show that the set of primitive recursive functions is itself a very rich class of functions. We do this essentially by looking at some closure properties of the set of primitive recursive functions. We know that the set is closed under the operations of composition and primitive recursion, since that was built into the very definition of the set. What is interesting is that this definition already guarantees a host of other closure properties. We begin with two relatively simple ones:

3.1.1 Bounded Sum and Product

Suppose $g : \mathbf{N}^{k+1} \to \mathbf{N}$ is any function, then we define the function $f_1 : \mathbf{N}^{k+1} \to \mathbf{N}$ given by

$$f_1(\bar{n}, m) = \sum_{i=0}^{m} g(\bar{n}, i)$$

as the **bounded sum** of g. Similarly, the function $f_2 : \mathbf{N}^{k+1} \to \mathbf{N}$ given by

$$f_2(\bar{n}, m) = \prod_{i=0}^{m} g(\bar{n}, i)$$

is the **bounded product** of g.

That is, for any number m, the value of $f_1(\bar{n}, m)$ is obtained by adding together the values $g(\bar{n}, 0), g(\bar{n}, 1), \ldots$ and so on until $g(\bar{n}, m)$. Similarly, the value of $f_2(\bar{n}, m)$ is obtained by multiplying this series of numbers together. Note that f_1 cannot be defined by a simple application of the addition function, since the number of values being added changes depending upon the input to f_1. In the same way, f_2 is not a simple application of the multiplication function. In order to define these functions, we need to resort to primitive recursion.

Theorem 16 *If g is a primitive recursive function, then so is its bounded sum and bounded product.*

Proof: The bounded sum is defined by:

$$\begin{aligned} f_1(\bar{n}, 0) &= g(\bar{n}, 0) \\ f_1(\bar{n}, m+1) &= \mathrm{add}(g(\bar{n}, m+1), f_1(\bar{n}, m)) \end{aligned}$$

and the bounded product by:

$$\begin{aligned} f_2(\bar{n}, 0) &= g(\bar{n}, 0) \\ f_2(\bar{n}, m+1) &= \mathrm{mult}(g(\bar{n}, m+1), f_2(\bar{n}, m)) \end{aligned}$$

Note that these definitions are not quite in the right format for a definition by primitive recursion. Take, for instance, the definition of f_1. Since it is a $(k+1)$-ary function, its base case should be given by a k-ary function. So, when we write $g(\bar{n}, 0)$, we really mean the function $g' : \mathbf{N}^k \to \mathbf{N}$ that we get by composing the $(k+1)$-ary function g with the $k+1$ functions $\pi^k_1, \ldots, \pi^k_k, \zeta_k$, each of arity k. However, it is convenient to write it as above, since the meaning is clear and to put down the exact definition would simply clutter the page with notation.

Similarly, in the recursive step of the definition of f_1, the function of arity $k+2$ that is being used is the function obtained from the composition of add : $\mathbf{N} \to \mathbf{N}$ with the two $(k+2)$-ary functions $h : \mathbf{N}^{k+2} \to \mathbf{N}$ and π^{k+2}_{k+2}, where h is itself obtained from the composition of g with the $k+1$ functions $\pi^{k+2}_1, \ldots, \pi^{k+2}_{k+1}$, each of arity $k+2$. The same remarks apply to the definition of f_2.

In future, we will often take such liberties with primitive recursive definitions. That is, we will write recursive definitions like the above, where the meaning is clear, and it is clear that a formally precise primitive recursive definition can be given by composing with the appropriate projection functions. This is, in fact, the reason for including projection functions among the basic functions we started out with. While they play this vital rôle, a definition is often much clearer if they are left out.

3.1.2 Primitive Recursive Predicates

Consider the following function $p : \mathbf{N} \to \mathbf{N}$:

$$p(n) = \begin{cases} 1 & \text{if } n \text{ is prime} \\ 0 & \text{otherwise.} \end{cases}$$

It is intuitively clear that this function is computable by the following algorithm: on input n, for each number $1 < m < n$, check to see if m divides n; if an m is found that divides n, then output the value 0, otherwise output the value 1. We could, in fact, take this informal description of an algorithm, and turn it into a URM program. However, the purpose of such a program is clearly not in the result it produces (0 or 1), but in the fact that it *decides* a certain property of its input (whether or not it is prime). Similarly, for any subset $S \subseteq \mathbf{N}$ of the natural numbers, we can ask whether or not the following function is computable (or even primitive recursive):

$$c_S(n) = \begin{cases} 1 & \text{if } n \in S \\ 0 & \text{otherwise.} \end{cases}$$

More generally, we can ask of any relation R on the natural numbers, i.e. any subset $R \subseteq \mathbf{N}^k$ (for some k), whether or not the corresponding function that decides membership is computable or primitive recursive. While the question of the computability of such decision problems will be the main focus of Section 4, here we are just going to consider those relations which are primitive recursive, and use them to construct further primitive recursive functions. To this end, we introduce the following definition.

Definition 17 *1. A predicate is any relation on the natural numbers, that is, it is a subset of $\mathbf{N}^k$ for some k.*

2. Given a predicate $P \subseteq \mathbf{N}^k$, the characteristic function of P is a function $\chi_P : \mathbf{N}^k \to \mathbf{N}$ given by:

$$\chi_P(\bar{n}) = \begin{cases} 1 & \textit{if } \bar{n} \in P \\ 0 & \textit{otherwise} \end{cases}$$

3. A predicate is said to be primitive recursive, if its characteristic function is.

For example, the equality predicate $=$ is primitive recursive, as its characteristic function eq is given by

$$\mathrm{eq}(n, m) = 1 \dot{-} \mathrm{sg}((n \dot{-} m) + (m \dot{-} n)).$$

Similarly, the less-than predicate $<$ is primitive recursive, its characteristic function being

$$l(n, m) = \mathrm{sg}(n \dot{-} m).$$

Now, since predicates are just sets, we can combine them in the standard ways. Namely, given two predicates $P, Q \subseteq \mathbf{N}$, we have:

$$P \cup Q = \{\bar{n} \mid \bar{n} \in P \text{ or } \bar{n} \in Q\};$$

and

$$P \cap Q = \{\bar{n} \mid \bar{n} \in P \text{ and } \bar{n} \in Q\}.$$

We also define the complement of P, written $\bar{P}$, as the subset of N^k consisting of all the tuples that are not in P.

$$\bar{P} = \{\bar{n} \mid \bar{n} \notin P\}.$$

It turns out that combining primitive recursive predicates in any of these ways always gives new primitive recursive predicates. This is stated and proved in the following:

Theorem 18 *The primitive recursive predicates are closed under union, intersection and complementation.*

Proof: Let $P, Q \subseteq \mathbf{N}^k$ be two primitive recursive predicates, and χ_P and χ_Q their respective characteristic functions. Then,

$$\chi_{P \cup Q}(\bar{n}) = \mathrm{sg}(\chi_P(\bar{n}) + \chi_Q(\bar{n}))$$

$$\chi_{P \cap Q}(\bar{n}) = \chi_P(\bar{n})\chi_Q(\bar{n})$$

$$\chi_{\bar{P}}(\bar{n}) = 1 \dot{-} \chi_P(\bar{n})$$

are all primitive recursive. ⋈

So, for instance, since we say that $=$ and $<$ are primitive recursive predicates, it follows from the above closure properties that so are $\leq$, $>$,$\geq$ and $\neq$. This is because $\leq$ is just the union of the two predicates $=$ and $<$; $>$ is the complement of $\leq$; $\geq$ is the union of $=$ and $>$; and $\neq$ is the complement of $=$.

3.1.3 Bounded Quantification

Another way of obtaining a new predicate from one already defined is by quantification. We first consider only bounded quantification, which is as follows. Given a predicate $P \subseteq \mathbf{N}^{k+1}$, the predicate $Q \subseteq \mathbf{N}^{k+1}$ is obtained from P by bounded existential quantification, if

$(\bar{n}, m) \in Q$ if, and only if, there is some $i \leq m$ such that $(\bar{n}, i) \in P$.

We write,

$$Q = \exists i \leq m[P].$$

Similarly, R is obtained from P by bounded universal quantification, if

$(\bar{n}, m) \in R$ if, and only if, for every $i \leq m$ such that $(\bar{n}, i) \in P$.

We write,

$$R = \forall i \leq m[P].$$

These operations also, when applied to primitive recursive predicates yield only primitive recursive predicates. This is established in the following:

Theorem 19 *If $P \subseteq \mathbf{N}^{k+1}$ is primitive recursive, then so are $\exists i \leq m[P]$ and $\forall i \leq m[P]$.*

Proof: The characteristic function f_1 of $\forall i \leq m[P]$ is given by

$$f_1(\bar{n}, m) = \prod_{i=0}^{m} \chi_P(\bar{n}, i).$$

This, being the bounded product of a primitive recursive function, is primitive recursive. Similarly, the characteristic function f_2 of $\exists i \leq m[P]$ is given by:

$$f_2(\bar{n}, m) = \mathrm{sg}(\sum_{i=0}^{m} \chi_P(\bar{n}, i)).$$

which is also primitive recursive. ⋈

Example 14 *1. Since the two place predicate $2m = n$ is primitive recursive (its characteristic function is obtained from* eq *by composition with the multiply by 2 function), so is the set of even numbers defined by:*

$$\exists m \leq n[2m = n].$$

2. The set of squares is a primitive recursive predicate, defined by

$$\exists m \leq n[m \cdot m = n].$$

3. The set of primes is also a primitive recursive predicate, defined by

$$\forall m_1 \leq n \forall m_2 \leq m_1[(m_2 = n) \vee \neg(m_1 m_2 = n)],$$

where $\vee$ is used to denote the operation of taking the union of the two predicates, and $\neg$ is the complement operation.

Note that, in order to get the arity of the predicates to conform with the definition of bounded quantification, we need to insert some projections into the above. Once again, the details are omitted in the interest of clarity.

3.1.4 Definition by Cases

While some predicates are useful in their own right, and it is of practical importance to know that they are computable (or even primitive recursive), the primitive recursive definition of predicates can also be useful in defining new functions. One way of doing this is to define a function by cases.

Suppose $g_1, \ldots, g_l : \mathbf{N}^k \to \mathbf{N}$ is a collection of functions, and $P_1, \ldots, P_l \subseteq \mathbf{N}^k$ is a collection of predicates which are mutually exclusive and exhaustive. That is, for each $i \neq j$, $P_i \cap P_j = \emptyset$ and $P_1 \cup \cdots \cup P_l = \mathbf{N}^k$ (another way of saying the same thing is to say that $P_1, \ldots, P_l$ is a partition of $\mathbf{N}^k$). Then, the function $f : \mathbf{N}^k \to \mathbf{N}$ defined by cases from $g_1, \ldots, g_l$ and $P_1, \ldots, P_l$ is given by:

$$f(\bar{n}) = \begin{cases} g_1(\bar{n}) & \text{if } \bar{n} \in P_1 \\ \vdots & \\ g_l(\bar{n}) & \text{if } \bar{n} \in P_l \end{cases}$$

Example 15 *For example, consider the function given by the definition:*

$$f(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ n^3 & \text{if } n \text{ is odd} \end{cases}$$

Here P_1 is the set of even numbers, and P_2 is the set of odd numbers. These are mutually exclusive since no number is both even and odd, and they are exhaustive since every number is either even or odd. The function $g_1 : \mathbf{N} \to \mathbf{N}$ is n^2, and $g_2 : \mathbf{N} \to \mathbf{N}$ is n^3. We can say that f is defined from g_1, g_2 and P_1, P_2 by cases.

Theorem 20 *If $g_1, \ldots, g_l : \mathbf{N}^k \to \mathbf{N}$ and $P_1, \ldots, P_l \subseteq \mathbf{N}^k$ are primitive recursive, then so is f defined as above.*

Proof:

$$f(\bar{n}) = g_1(\bar{n})\chi_{P_1}(\bar{n}) + \cdots + g_l(\bar{n})\chi_{P_l}(\bar{n}).$$

Note, that for each $\bar{n} \in \mathbf{N}^k$, at most one of the terms will be non-zero. ⋈

That is to say, that f is obtained from the above functions and the add and multiply functions by composition.

Thus, the function in Example 15 is, indeed, primitive recursive, since the functions n^2 and n^3 are primitive recursive, as are the predicates "n is odd" and "n is even."

3.1.5 Bounded Minimalisation

We now look at one final way of defining functions from predicates. This is similar to the bounded quantification of Section 3.1.3, but is used to define a function rather than a predicate.

Suppose $P \subseteq \mathbf{N}^{k+1}$ is a predicate. We define the function $f : \mathbf{N}^{k+1} \to \mathbf{N}$ by **bounded minimalization** over P as follows:

$$f(\bar{n}, m) = \begin{cases} \text{the least } i \leq m \text{ such that } (\bar{n}, i) \in P & \text{if } \exists i \leq m (\bar{n}, i) \in P \\ 0 & \text{otherwise.} \end{cases}$$

We write $f(\bar{n}, m) = \mu i \leq m[(\bar{n}, i) \in P]$.

Example 16 *As an example, consider the function* $\text{half} : \mathbf{N} \to \mathbf{N}$ *where* $\text{half}(n)$ *is the least number greater than or equal to $n/2$ (usually written as $\lceil n/2 \rceil$). This function is given by the following definition.*

$$\text{half}(n) = \mu m \leq n(2 \cdot m > n).$$

That is, it is obtained by bounded minimalization from the predicate $(2 \cdot m) \geq n$.

Similarly, consider the function $\text{sqrt} : \mathbf{N} \to \mathbf{N}$ *where* $\text{sqrt}(n)$ *is the least number greater than or equal to $\sqrt{n}$ (i.e. $\text{sqrt}(n) = \lceil \sqrt{n} \rceil$). This is given by:*

$$\text{sqrt}(n) = \mu m \leq n(m \cdot m \geq n).$$

As one might expect, any function obtained by bounded minimalisation over a primitive recursive predicate is itself primitive recursive. This is formally proved below.

Theorem 21 *If $P \subseteq \mathbf{N}^{k+1}$ is primitive recursive, then so is the function f obtained from P by bounded minimalization.*

Proof: Define the function $h : \mathbf{N}^{k+2} \to \mathbf{N}$ as follows:

$$h(\bar{n}, m, j) = \begin{cases} j & \text{if } \exists i \leq m[(\bar{n}, i) \in P] \\ m+1 & \text{if } (\bar{n}, m+1) \in P \text{ and not } \exists i \leq m[(\bar{n}, i) \in P] \\ 0 & \text{otherwise.} \end{cases}$$

Then, h is primitive recursive, being defined by cases from primitive recursive predicates and functions. Moreover, f is obtained from h by the following primitive recursion.

$$\begin{aligned} f(\bar{n}, 0) &= 0 \\ f(\bar{n}, m+1) &= h(\bar{n}, m, f(\bar{n}, m)) \end{aligned}$$

Thus f is primitive recursive. ⋈

In particular, the functions half and sqrt defined above are primitive recursive, since they are obtained by bounded minimalisation from the primitive recursive predicates $(2 \cdot m) \geq n$ and $m^2 \geq n$ respectively. As an exercise, how would one define $\lfloor n/2 \rfloor$ or $\lfloor \sqrt{n} \rfloor$?

3.2 Fast Growing Functions

Let β_0 be the addition function, and β_1 the multiplication function. Observe that:

$$\beta_1(m, n) = \underbrace{\beta_0(\cdots \beta_0(m, m) \cdots)}_{n \text{ times}}.$$

That is, as we all know, the multiplication function is just repeated addition.

In general, define the function $\beta_{k+1} : \mathbf{N}^2 \to \mathbf{N}$ by

$$\beta_{k+1}(m, n) = \underbrace{\beta_k(\cdots \beta_k(m, m) \cdots)}_{n \text{ times}}.$$

Thus, β_2 is the exponentiation function, β_3 is the tower function and so on. Each of these grows much faster than the previous function. Still, each of these can be defined by primitive recursion from the previous one. To see this, note that:

$$\begin{aligned} \beta_{k+1}(m, 0) &= \beta_k(m, 0) \\ \beta_{k+1}(m, n+1) &= \beta_k(m, \beta_k(m, n)) \end{aligned}$$

Since we know that the addition function is primitive recursive, it follows that all of the functions β_k are primitive recursive. It can be seen that each function β_{k+1} grows much faster than all the previous functions. In a sense that can be made precise, the function β_k

grows faster than any function that can be defined using at most k applications of primitive recursion. Thus, define the diagonal function $\beta : \mathbf{N}^2 \to \mathbf{N}$ given by

$$\beta(m,n) = \underbrace{\beta_n(\cdots \beta_n(m,m)\cdots)}_{n \text{ times}}.$$

That is, not only is the number of times the function is applied dependent on the input n, but the very choice of the function being applied is determined by n.

The function β defined above is total, by its very definition; it is computable again, because there is a procedure to compute it that is implicit in the definition. But, the function cannot be primitive recursive, since it grows faster than any primitive recursive function. A closely related function is Ackermann's function, which is also a total computable function that grows faster than any primitive recursive function. This function is given by the following simple rules.

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

3.3 Unbounded Minimalisation

We have now seen that the operations of composition and primitive recursion, applied to the basic functions yield a very rich class of functions, but they are not sufficient to give us all the computable functions. It turns out that we can add just one more operation, which together with the two already given suffices to yield all the computable functions. This operation is similar to the bounded minimalisation operator introduced in Section 3.1.5, but simpler. We define this operation first.

Suppose $P \subseteq \mathbf{N}^{k+1}$ is a predicate. We define the partial function $f : \mathbf{N}^k \to \mathbf{N}$ by **unbounded minimalization** over P as follows:

$$f(\bar{n}) = \begin{cases} \text{the least } i \text{ such that } (\bar{n},i) \in P & \text{if } \exists i[(\bar{n},i) \in P] \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We write $f(\bar{n}) = \mu i[(\bar{n},i) \in P]$.

If P is primitive recursive, then f is computable. That is, there is a URM which, if started on inputs $\bar{n}$ will halt with the output value $f(\bar{n})$, if this value is defined, and it will not halt otherwise.

Now, unlike with the other operations we have introduced so far, the class of primitive recursive functions is not closed under the operation of unbounded minimalisation. That is, the function f above is not necessarily primitive recursive, even if P is. In fact, that is clear from the fact that f need not be a total function. The collection of all functions that can be obtained from the basic functions using only the operations of composition, primitive recursion and unbounded minimalisation is called the class of *Partial Recursive Functions*.

However, there are also *total* functions that can be defined as the unbounded minimalisation of a primitive recursive predicate, but which are not themselves primitive recursive. The Ackermann function given in Section 3.2 above is an example of such a function. So, why is the Ackermann function definable by unbounded minimalisation? This is a consequence of the following result:

Theorem 22 *Any computable function is definable by a finite number of applications of composition, primitive recursion and unbounded minimalisation from the basic functions.*

In other words, this one additional operation, along with the two others that we have seen before is sufficient to generate, from the basic functions, all the computable functions, and the class of partial recursive functions is exactly the same as the class of computable functions.

We defer a proof of Theorem 22 until later in the course. Here, we will just observe that what we have succeeded in doing is essentially to give an algebra of the computable functions.

algebra

$$PRF$$

carriers

$$F_0, F_1, \ldots$$

constants

$$\zeta_k :\to F_k$$
$$\sigma :\to F_1$$
$$\pi^i_k :\to F_k$$

operations

$$\mathtt{comp}_{k,l} : F_k \times F_l^k \to F_l$$
$$\mathtt{pr}_k : F_k \times F_{k+2} \to F_{k+1}$$
$$\mu_k : F_{k+1} \to F_k$$

In the above, each F_k is the collection of all k-ary computable functions. That is,

$$F_k = \{f : \mathbf{N}^k \to \mathbf{N} \mid f \text{ is computable}\}.$$

The constants are the basic functions as defined earlier. The operations are the methods of defining new functions from old. Thus $\mathtt{comp}_{k,l}$ is the composition operation, which given a function in F_k, and k functions in F_l, yields a function in F_l. Similarly, the primitive recursion operation $\mathtt{pr}_k$, given a function in F_k and another function in F_{k+2} yields a function in F_{k+1}. Finally, the operator μ_k is the unbounded minimalisation operator, which given a function in F_{k+1} gives a function in F_k.

The above algebra can be seen, for instance, as the specification of a functional programming language. If the language contains symbols for each of the basic functions, and provides operations for combining programs through the operations of composition, primitive recursion and unbounded minimalisation, then we know (by Theorem 22) that any computable function can be defined in the language.

3.4 The Hierarchy of Recursive Functions

We have seen above the algebra PRF of the partial recursive functions. We have also seen in the course of this chapter, several interesting subclasses of this class of functions. We repeat their definitions below.

1. A function $f \in PRF$ is called a *polynomial function* if it can be obtained from the basic functions, using only the operation of composition. The set of all polynomial functions of n arguments is denoted $POLY_n$.

2. A function $f \in PRF$ is called a *primitive recursive function* if it can be obtained from the basic functions, using only the operations of composition and primitive recursion. The set of all primitive recursive functions of n arguments is denoted $PRIM_n$.

3. A function $f \in PRF$ is called a *recursive function* if it is a total function. The set of all recursive functions of n arguments is denoted REC_n.

It is clear from the definitions of these sets of functions that for each $n \geq 1$

$$POLY_n \subseteq PRIM_n \subseteq REC_n \subseteq PRF_n \subseteq FN_n.$$

where FN_n is the class of *all* functions on n arguments, whether computable or not.

However, it is not so clear that these containments are strict, but this is, in fact, the case for each one — that is,

1. there exists a function $f \in PRIM_n$ such that $f \notin POLY_n$;

2. there exists a function $f \in REC_n$ such that $f \notin PRIM_n$;

3. there exists a function $f \in PRF_n$ such that $f \notin REC_n$; and

4. there exists a function $f \in FN_n$ such that $f \notin PRF_n$.

These facts vary in their significance. That there exists a primitive recursive function that is not a polynomial function, and that there exists a recursive function that is not primitive recursive, is of great interest to researchers in recursive function theory, but these facts are of little interest to us here, except insofar as they are true! There are many functions that are primitive recursive but not polynomial — the addition function is one example. An example of a recursive function that is not primitive recursive is the Ackermann function (see Section 3.2). That there is a partial recursive function that is not a recursive function is obvious since we have already seen instances of partial recursive functions that are partial functions. Finally, since the partial recursive functions are exactly the computable functions, the existence of functions that are not partial recursive follows from the fact that there are functions that are not computable at all, as we saw in Section 1.5.

You may have noticed that among the classes of functions given above, REC differs from $POLY$, $PRIM$, and PRF in a fundamental way. While each of the classes $POLY$,

PRIM, and *PRF* is defined by the closure of the basic functions under some operations, *REC* is defined rather differently, as a subset of *PRF*. Thus, for each of *POLY*, *PRIM*, and *PRF*, we have an algebra, and we can construct a corresponding functional language. This is not so for *REC*. In fact, it is impossible, even in principle, to give such and algebra or language. We shall turn, in the next chapter, to a demonstration of this fact, which turns out to hinge on deep and fascinating facts about computable functions.

4 Uncomputability and Undecidability

In this section we are going to define functions that we can prove are uncomputable. We are also going to look at several examples of decision problems that are undecidable, that is to say there is no algorithm that will determine the answer in all cases. Recall from Sections 1.3 and 3.1.2 that a decision problem is a problem where we are asked to verify some property of the input and give a yes or no answer. This notion of a decision problem is formalised in Section 3.1.2 and repeated below.

4.1 Decision Problems

Consider the following function $p : \mathbf{N} \to \mathbf{N}$:

$$p(n) = \begin{cases} 1 & \text{if } n \text{ is prime} \\ 0 & \text{otherwise.} \end{cases}$$

It is intuitively clear that this function is computable by the following algorithm: on input n, for each number $1 < m < n$, check to see if m divides n; if an m is found that divides n, then output the value 0, otherwise output the value 1. This algorithm always terminates, because for each possible input n, there are only finitely many values of m to be checked, and checking each one is clearly a finite process, of division. Thus, by the Church-Turing thesis, the above function p is URM-computable.

The purpose of a program to compute the above function p is not the output of a numeric value (0 or 1), but to *decide* some property of its input (whether it is prime or not). Such a program is called a decision procedure. Let $S \subseteq \mathbf{N}$ be any set of natural numbers. The characteristic function of S, is defined to be the function $c_S : \mathbf{N} \to \mathbf{N}$ given by:

$$c_S(n) = \begin{cases} 1 & \text{if } n \in S \\ 0 & \text{otherwise.} \end{cases}$$

More generally, we can define the characteristic function of an arbitrary relation on the natural numbers. That is, suppose $R \subseteq \mathbf{N}^k$ is a k-ary relation on the natural numbers, then the characteristic function of R, is the function $c_R : \mathbf{N}^k \to \mathbf{N}$ given by:

$$c_R(n_1, \ldots, n_k)) = \begin{cases} 1 & \text{if } (n_1, \ldots, n_k) \in R \\ 0 & \text{otherwise.} \end{cases}$$

Definition If $R \subseteq \mathbf{N}^k$ is a k-ary relation on the natural numbers, and its characteristic function c_R is computable, then we say that R is *decidable*. Moreover, the program that computes c_R is called a decision procedure for R. ⋈

In general, any relation $R \subseteq \mathbf{N}^k$ or any set $S \subseteq \mathbf{N}$ defines a *decision problem* that may or may not be solvable.

Example 17 *The set $Pr \subseteq \mathbf{N}$ of prime numbers is a decision problem, and its characteristic function is the function p defined above. By the argument given above, p is computable, and so the set Pr is decidable.*

Example 18 *Consider the relation* $Ad \subseteq \mathbf{N}^3$ *defined by*

$$(x, y, z) \in Ad \quad \textit{if, and only if} \quad x + y = z.$$

It is not difficult to see that the program for adding two numbers can be modified to yield a decision procedure for the relation Ad. The relation is, therefore, decidable.

Later in this section, we will see examples of undecidable relations.

4.2 Gödel Numbers

It is not hard to imagine writing a program in Pascal (or your favourite programming language), which can act as an interpreter for URM programs. So, your program would take as input a URM program P, and the values $a_1, \ldots, a_n$, evaluate the program P on these inputs, and return the value returned by P. Of course, if P doesn't terminate on the given inputs, then neither would the interpreter.

Now, we have argued earlier that any Pascal program (or any program in your favourite language) can be equally well implemented as a URM program. But this implies that we could implement our interpreter as a URM program as well. Such an idea immediately runs into the problem of presenting programs as inputs to a URM. After all, the only inputs that a URM program accepts are natural numbers. However, we have also argued before that any datatype can be encoded as natural numbers (see Section 2.1.5). Moreover, we have also explicitly seen an encoding of strings over a finite alphabet as natural numbers (see Section 1.5). Since URM programs are certainly strings over a finite alphabet, this already gives us one encoding of URM programs into the natural numbers. One might argue that URM programs are, in fact, strings over an infinite alphabet since arbitrarily large numbers may appear in them, but these numbers can themselves be represented in decimal form, and so we only need a finite set of symbols altogether.

We also saw, in Section 1.5 that, for any infinite set A, such that there is an injection from A into $\mathbf{N}$, there is a bijection between A and N. So, there must be a bijection between the set of all URM programs and the natural numbers $\mathbf{N}$. What this means is that we can enumerate the set of all URM programs:

$$P_0, P_1, P_2, \ldots$$

However, in order to treat programs as inputs to URMs, we also want the encoding to be, in some intuitive sense "computable." The injective function $f : \Sigma^* \to \mathbf{N}$ that was defined in Section 1.5 clearly satisfies this criterion. That is, it is clear that we could write a program which, given a string in Σ^* computes the natural number that encodes it. Moreover, from f we could also define a bijection $g : \Sigma^* \to \mathbf{N}$ as follows:

$$g(\sigma) = \text{the number of } m < f(\sigma) \text{ such that there is a } \tau \in \Sigma^* \text{ such that } f(\tau) = m.$$

It is also clear that this function g is, in an intuitive sense, computable. That is, given a string σ, we first compute $f(\sigma)$ and then, in a loop we go through all numbers below the

value of $f(\sigma)$, counting those that are themselves encodings of strings in Σ^*. To do this, we have to be able to determine, given a number n, if it is, in fact, the encoding of some string in Σ^*, i.e. whether it is in the range of f. But, this only requires determining the prime factorisation of n and checking that each of the powers of primes is in the range of the size of Σ.

Thus, g is a bijection between Σ^* and $\mathbf{N}$, and g is, in an intuitive sense, computable. We cannot show that g is computable in a formal sense, that is to say, we cannot prove that g is URM-computable, because the notion of URM-computability has only been defined for functions on the natural numbers. To speak of a function on strings in Σ^* being computable we need to *first* encode such strings as natural numbers. This is exactly what the function g aims to do. Thus, we will leave the computability of g as a vague intuitive statement.

A function such as g above, which defines a bijection between some set S and the natural numbers, and is, in some intuitive sense computable or effective, is called a *Gödel numbering* of the set S, after the logician, Kurt Gödel.

The Gödel numbering of URM programs we obtain by applying the above definition of the function g is certainly not the only one possible (consult the course text for another, different numbering). For our purposes, the details of the encoding are not as important as the fact that one exists. In principle, any "reasonable" numbering of URM programs will do. From now on, we will just assume that some Gödel numbering has been fixed, and so we can speak of an enumeration of programs

$$P_0, P_1, P_2, \ldots$$

and of the first program, the second program, and so on. The following definition introduces some notation for referring to the functions defined by these programs.

Definition For each $n, k \in \mathbf{N}$ with $k \geq 1$, we write $\varphi_n^{(k)}$ to denote the k-ary (partial) function computed by the program P_n. $\bowtie$

In the following, when we are considering the unary functions, we will drop the superscript (1). Thus, we have an enumeration of the unary computable functions:

$$\varphi_0, \varphi_1, \varphi_2, \ldots$$

Similarly, we have an enumeration of the k-ary computable functions for every k. Note that this is *not* a bijection between computable functions and the natural numbers. This is because each function may appear more than once in this enumeration. There can be more than one program for computing the same function, and since any two programs are different strings of symbols, they will appear at different places in the enumeration. In fact, it is not hard to see that each computable function must appear infinitely often in the enumeration, because we can always come up with new programs for computing the same function, for instance by taking a previous proogram and adding dummy instructions to it.

We can now define the "universal computation function" as a function of two inputs. That is $\psi_U : \mathbf{N} \times \mathbf{N} \hookrightarrow \mathbf{N}$ is given by:

$$\psi_U(x, y) = \varphi_x(y)$$

More generally, the universal function for k-ary computable functions is $\psi_U^{(k)} : \mathbf{N}^{k+1} \hookrightarrow \mathbf{N}$ given by:

$$\psi_U^{(k)}(x, y_1, \ldots, y_k) = \varphi_x^{(k)}(y_1, \ldots, y_k)$$

These universal functions are the functions computed by the "interpreter" discussed at the beginning of this section. We will later see a proof that they are, in fact, computable.

4.3 Uncomputable Functions and Undecidable Problems

We are now in a position to define our first uncomputable function. Consider the function $u : \mathbf{N} \to \mathbf{N}$ given by:

$$u(n) = \begin{cases} \varphi_n(n) + 1 & \text{if } \varphi_n(n) \text{ is defined} \\ 0 & \text{otherwise} \end{cases}$$

Note the similarity between this definition and the definition of the function d in Section 1.5. The function u above is also defined by "diagonalisation." There are two main differences between the definitions of the functions d and u. The first is that the function d was defined with respect to an enumeration of *total* functions on the natural numbers. The functions φ are not all total, and we must take this into account in defining u. The second difference is that the function d was defined with respect to an *assumed* enumeration of all functions from $\mathbf{N}$ to $\mathbf{N}$ in order to show that no such enumeration exists. Thus d is not a single function, it depends on the choice of enumeration, and since we ended up showing that no such enumeration is possible, the function d also does not exist — it was a fictitious function that played a crucial role in our proof. On the other hand, an enumeration of all computable partial functions exists. We have already established that through Gödel numbering URM programs. Thus, u is defined with respect to a well-defined enumeration, and is a well-defined function. The purpose of the diagonalisation here is to show that u cannot possibly occur in the enumeration, and therefore that it must be uncomputable. This is what we prove next.

Theorem 23 *The function u defined above is not computable.*

Proof: Suppose that u is in fact computable. Then, there must be some URM program p that computes it. Now, p must occur somewhere in the enumeration of all URM programs, that is to say there is a number $m \in \mathbf{N}$ such that p is P_m. This means that $u = \varphi_m^{(1)}$. Now, either φ_m is defined at the value m or it is not. If it is defined, then, the value of $u(m)$ is $\varphi_m(m) + 1$, which means that φ_m and u are not the same function, since they differ at m. So, perhaps φ_m is undefined at m. But then, $u(m) = 0$, and so once again, u and φ_m are different at the value m. In either case u and φ_m differ at the value m, by the very definition of u. Thus, we have a contradiction, and we conclude that u is not computable. $\bowtie$

In our definition of the function u, in order to guarantee that u is uncomputable, we only had to ensure that it is different from every computable function in our enumeration.

Within this constraint, we could have defined anything. For instance, it follows from the above argument that the following function $v : \mathbf{N} \to \mathbf{N}$ is also uncomputable.

$$v(n) = \begin{cases} \varphi_n(n) + 5 & \text{if } \varphi_n(n) \text{ is defined} \\ 6 & \text{otherwise} \end{cases}$$

4.3.1 Undecidability

We can use a diagonalisation argument similar to that for Theorem 23 to show that the following set is undecidable:

$$S = \{x \in \mathbf{N} | \varphi_x(x) \downarrow\}.$$

(Note, the notation $\varphi_x(y) \downarrow$ is used to denote that φ_x is defined for input y.)

In order to show that S is undecidable, we first prove that the following function $d : \mathbf{N} \hookrightarrow \mathbf{N}$ is uncomputable:

$$d(n) = \begin{cases} 1 & \text{if } \varphi_x(x) \uparrow \\ \perp & \text{otherwise} \end{cases}$$

Lemma 24 *The function d defined above is uncomputable.*

Proof: Suppose the function d were computable. This would mean that there is some $m \in \mathbf{N}$ such that $d = \varphi_m$. We can consider two cases:

case 1: $\varphi_m(m)$ is defined, in which case, by definition of d, $d(m) = \perp$ and therefore $d \neq \varphi_m$.

case 2: $\varphi_m(m)$ is undefined, and therefore, by definition of d, $d(m) = 1$, and once again $d \neq \varphi_m$.

In either case, d is different from φ_m, contradicting the assumption that $d = \varphi_m$. We conclude that d is not computable. $\bowtie$

Now, we are ready to show that the set S is undecidable. This is the same thing as showing that its characteristic function $c_S : \mathbf{N} \to \mathbf{N}$ is uncomputable.

Theorem 25 *The set $S = \{x \in \mathbf{N} | \varphi_x(x) \downarrow\}$ is undecidable.*

Proof: Recall that the characteristic function c_S is defined by:

$$c_S(n) = \begin{cases} 1 & \text{if } n \in S \\ 0 & \text{otherwise.} \end{cases}$$

Now, suppose that c_S were computable. But, the function d can be defined as:

$$d(n) = \begin{cases} 1 & \text{if } c_S(n) = 0 \\ \perp & \text{otherwise.} \end{cases}$$

It is clear from this, that if c_S were computable, then d would also be computable. But, we have already proved, in Lemma 24, that d is uncomputable. So, we must conclude that c_S is uncomputable as well. $\bowtie$

Once we have established that S is undecidable, we can now use this fact to show that the binary relation $H \subseteq \mathbf{N} \times \mathbf{N}$ defined below is not decidable:

$$H = \{(x, y) | \varphi_x(y) \downarrow\}.$$

That is, H is the set of those pairs of numbers (x, y) such that the xth computable function is defined at the value y. Or alternatively, H is the set of pairs (x, y) such that the xth URM program halts when given input y. For this reason, the decision problem for H is called the *halting problem*. The proof that H is undecidable is straightforward.

Theorem 26 *The relation $H = \{(x, y) | \varphi_x(y) \downarrow\}$ is undecidable.*

Proof: In Theorem 25 we saw that the function c_S is uncomputable. The function c_S can be defined by

$$c_S(n) = c_H(n, n).$$

It is immediately clear from this definition that, if the function c_H were computable, then c_S would be computable as well (essentially, we could write a program for computing c_S that used the program for c_H as a "subroutine"). Since we have already proved that c_S is uncomputable, it follows that c_H is uncomputable as well. $\bowtie$

The significance of Theorem 26 lies in the fact that this is a relation that we would really like to be able to decide. Our previous examples of uncomputable functions and undecidable sets seem somewhat contrived. That is to say, the sets or functions were defined in such a way as to make them undecidable or uncomputable, typically by a process of diagonalisation. They are not necessarily functions that we might actually be interested in computing. However, it would be very nice to have a decision procedure for the halting problem. It would be nice, before we start up a program with a particular input, to be able to tell before hand whether or not the program would terminate. If we had an algorithm that could perform this task, we could build it into a compiler, which could then, perhaps, detect the presence of infinite loops at compile time!

Theorem 26 tells us that all this is, even in principle, impossible. There simply is no algorithm that can decide, looking at the code of an arbitrary program and a particular input to that program, whether the program will halt on that input. Note that, this is not to say that there is no way to decide, for some fixed program, and some fixed input, whether it will terminate. The point is, that there is no algorithm which will work for all possible programs, and decide the halting problem.

In fact, we can say somewhat more. We can come up with a single program, such that the set of inputs for which that program terminates is undecidable. This is proved below.

Theorem 27 *There is a computable function h, such that the set $\{x \in \mathbf{N} | h(x) \downarrow\}$ is undecidable.*

Define $h : \mathbf{N} \hookrightarrow \mathbf{N}$ as

$$h(x) = \varphi_x(x)$$

That h is computable follows from the computability of the universal program ψ_U, since $h(x) = \psi_U(x, x)$. Moreover, it is clear that the set $\{x \in \mathbf{N} | h(x) \downarrow\}$ is undecidable, since this is just the same as the set S. $\bowtie$

4.3.2 Reductions

The proof of Theorem 26, using Theorem 25 is an example of a reduction. That is, having proved that the set S is undecidable, we were able to prove that the halting problem — the relation H — is also undecidable by proving the conditional statement:

> If H is decidable, then S is also decidable.

Since we had already proved that S is undecidable, it follows from the above statement that H is also undecidable. Moreover, the conditional statement above was proved by transforming possible inputs for a maching deciding H into inputs for a machine deciding S. This idea is formalised in the following definition:

Definition Given relations $T \subseteq \mathbf{N}^k$ and $R \subseteq \mathbf{N}^m$, a reduction from T to R is a *computable* function $f : \mathbf{N}^k \to \mathbf{N}^m$ such that, for any $a \in N^k$:

$$f(a) \in R \quad \text{if, and only if,} \quad a \in T.$$

We say that a relation T is reducible to R, written $T \leq R$, if there is a reduction from T to R. $\bowtie$

Note that a reduction is a *total* computable function. Note also that this is a more generalised notion of a computable function then what we have been using so far. That is, the output of the function is not just a natural number, but may be a tuple of natural numbers. It should be clear how the definition of a computable function can be appropriately extended to this situation.

So, in the previous section, where S is the set $\{x \in \mathbf{N} | \varphi_x(x) \downarrow\}$, and H is the relation $\{(x, y) \in \mathbf{N} \times \mathbf{N} | \varphi_x(y) \downarrow\}$, the reduction $r : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ from S to H is given by:

$$r(x) = (x, x).$$

In order to check that the function r given above is, indeed, a reduction from S to H, we need to verify that r is computable and for any given $x \in \mathbf{N}$, $x \in S$ if, and only if, $r(x) \in H$. That r is computable is clear, either from the Church-Turing thesis, or directly by using the transfer program. For the second property of r, there are two things to check here: (1) that, if $x \in S$, then $r(x) \in H$; and (2) that, if $x \notin S$, then $r(x) \notin H$.

For (1), we need to check that if $x \in S$, then $(x, x) \in H$, since $r(x) = (x, x)$. Note that if $x \in S$, this means, by the definition of S, that $\varphi_x(x) \downarrow$, and this implies, by the definition of H, that $(x, x) \in H$. For (2), we need to check that if $x \notin S$, then $(x, x) \notin H$. If $x \notin S$, by the definition of S, $\varphi_x(x) \uparrow$, and this implies, by the definition of H, that $(x, x) \notin H$.

The main use to which we will put reductions is in showing that certain sets or relations are undecidable, by showing that some other relation, which we have already proved undecidable is reducible to it. This technique is based on the following lemma:

Lemma 28 *If $T \subseteq \mathbf{N}^k$ is reducible to $R \subseteq \mathbf{N}^m$, and R is decidable, then T is decidable.*

Proof: Let $f : \mathbf{N}^k \to \mathbf{N}^m$ be the reduction from T to R, and let $c_R : \mathbf{N}^m \to \mathbf{N}$ be the characteristic function of R. By assumption, both of these functions are computable. Consider the composition of these two functions: $(c_R \circ f) : \mathbf{N}^k \to \mathbf{N}$. This function is computable, since the composition of any two computable functions is computable (see Section 2.2.3). Moreover, $c_R \circ f$ is, in fact, the characteristic function of T. To see this, let a be any tuple in $\mathbf{N}^k$.

$$\begin{aligned}(c_R \circ f)(a) &= c_R(f(a)) \\ &= \begin{cases} 1 & \text{if } f(a) \in R \\ 0 & \text{if } f(a) \notin R \end{cases} \\ &= \begin{cases} 1 & \text{if } a \in T \\ 0 & \text{if } a \notin T \end{cases}\end{aligned}$$

Thus, we conclude that the characteristic function of T is computable, and therefore T is decidable. ⋈

Intuitively speaking, if one decision problem is reducible to another, this means that the first problem is no more difficult to compute than the second, because a procedure computing the second automatically yields a procedure to compute the first. For proving undecidability results, Lemma 28 is particularly useful in conjunction with the following theorem, called the *Parametrisation Theorem* or more often, the *s-m-n* theorem, which is given below without proof.

Theorem 29 *Let $f : \mathbf{N} \times \mathbf{N} \hookrightarrow \mathbf{N}$ be a computable function. There is a* total *computable function $s : \mathbf{N} \to \mathbf{N}$ such that:*

$$f(x, y) = \varphi_{s(x)}(y).$$

To understand the statement of the theorem, consider a particular function of two arguments, such as the addition function add : $\mathbf{N} \times \mathbf{N} \to \mathbf{N}$. Now, for any fixed number x, there is a function $\text{add}_x : \mathbf{N} \to \mathbf{N}$ of one argument, that simply adds x to its argument. For instance, add_5 is the function that adds 5 to its argument. What Theorem 29 guarantees is that there is a computable function, which given an input x, produces as a result the code for the program computing the function add_x.

We will not look at a formal proof of this theorem, but just consider an informal argument of why it is true. Suppose $f : \mathbf{N} \times \mathbf{N} \hookrightarrow \mathbf{N}$ is any computable function. This means we have a program which computes the function f. The function $f_x : \mathbf{N} \hookrightarrow \mathbf{N}$ of one argument that is obtained by fixing the first argument of f to be x, can be computed by a program P_x that does the following:

1. move the argument from register R_1 to register R_2;

2. load the constant x into register R_1;

3. start up the program computing f.

Now, we note that there is a program S that, given input x, produces the code for the above program P_x. This program S computes exactly the function s of Theorem 29.

As an example, recall the function $\zeta_1 : \mathbf{N} \to \mathbf{N}$ given by $\zeta_1(n) = 0$ for all $n \in \mathbf{N}$.

Theorem 30 *The set $Z \subseteq \mathbf{N}$ given by $Z = \{x \in \mathbf{N} | \varphi_x = \zeta_1\}$ is undecidable.*

Proof: Consider the function $f : \mathbf{N} \times \mathbf{N} \hookrightarrow \mathbf{N}$ given by

$$f(x,y) = \begin{cases} 0 & \text{if } \varphi_x(x) \downarrow \\ \perp & \text{otherwise} \end{cases}$$

f is computable, as it can be defined as the composition of two computable functions, i.e. $f = (\zeta_1 \circ \psi_U)$. Note that the value of $f(x,y)$ does not depend on y at all.

Now, by the s-m-n theorem, there is a total computable function $s_f : \mathbf{N} \to \mathbf{N}$ with the property that

$$\varphi_{s_f(x)}(y) = f(x,y).$$

Moreover, by the definition of f, it is clear that for any x, $\varphi_{s_f(x)}$ is either the everywhere undefined function, or it is the function ζ_1. The latter case occurs just in case $\varphi_x(x) \downarrow$. That is,

$$\varphi_{s_f(x)} = \zeta_1 \quad \text{if, and only if,} \quad \varphi_x(x) \downarrow .$$

But, this implies that the function s_f, which we know is computable by the s-m-n theorem, is a reduction from the set $S = \{x \in \mathbf{N} | \varphi_x(x) \downarrow\}$ to the set Z. Since we have already proved that S is undecidable, it follows that Z is also undecidable.

Now, having proved that Z is undecidable, we can show that any problem to which Z is reducible is also undecidable. One example is the following.

Corollary 31 *The relation $E \subseteq \mathbf{N} \times \mathbf{N}$ given by*

$$E = \{(x,y) \mid \varphi_x = \varphi_y\}$$

is undecidable.

Proof: Let c be the number coding the program in Section 2.2.1 which computes the function ζ_1. Then the function $f : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ given by:

$$f(n) = (n,c)$$

is a reduction from the set Z to the relation E. To check this, note first that the function f is computable, and then check that $n \in Z$ if, and only if, $(n,c) \in E$. $\bowtie$

As another example, consider the set $T \subseteq \mathbf{N}$ given by

$$T = \{x \mid \varphi_x \text{ is a total function}\}.$$

That is, T is the set of Gödel numbers of programs that compute total unary functions. Or, equivalently, T is the set of Gödel numbers of programs that halt on all unary inputs.

Theorem 32 *T is undecidable.*

Proof: We give a reduction from the set $S = \{x \mid \varphi_x(x)\downarrow\}$ to the set T. To show that such a reduction exists, consider the partial function $f : \mathbf{N} \times \mathbf{N} \hookrightarrow \mathbf{N}$ given by:

$$f(x,y) = \begin{cases} 1 & \text{if } \varphi_x(x)\downarrow \\ \bot & \text{otherwise.} \end{cases}$$

By the *s-m-n* theorem, there is a total computable function s_f such that

$$\varphi_{s_f(x)}(y) = f(x,y).$$

By the definition of f, this means that the function $\varphi_{s_f(x)} : \mathbf{N} \hookrightarrow \mathbf{N}$ is a total function if, and only if, $\varphi_x(x)\downarrow$. In other words, s_f is a reduction from the set S to the set T. $\bowtie$

The above examples have shown that the *s-m-n* theorem provides a very powerful method for constructing reductions from one undecidable set to another, and therefore for proving sets to be undecidable. This is particularly so when the sets one is dealing with are in some sense properties of computable functions. Thus, the set T in Theorem 32 is a set of numbers, but we are interested in it primarily as a decision problem on *computable functions*. That is, it is our way of formalising the decision problem:

> Given a computable function, is it total?

Similarly, the set Z of Theorem 30 is intended to formalise the decision problem:

> Given a computable function, is it the function ζ_1?

This idea of a set of numbers as representing a decision problem about functions is formalised in the definition below.

Definition A set $C \subseteq \mathbf{N}$ is an *index set* if the following is true for any numbers $x, y \in \mathbf{N}$:

> If $\varphi_x = \varphi_y$ then $x \in C$ if, and only if, $y \in C$.

$\bowtie$

Intuitively, what the definition says is that C is an index set if, whenever it contains the code of some program P, it also contains the code of every other program that computes the same function as P. We can think of an index set as representing a decision problem about some property of computable functions. That is, we can actually think of C as a set of functions.

Now, we can use the *s-m-n* theorem to prove a very general result about undecidability. This theorem, first proved by Rice in 1953 says that except for the completely trivial cases, no property of computable functions is decidable.

Theorem 33 (Rice's theorem) *If $C \subseteq \mathbf{N}$ is an index set, and $C \neq \emptyset$ and $C \neq \mathbf{N}$, then C is undecidable.*

Proof: First, observe that if C is an index set, then $\bar{C}$, the complement of C is also an index set. By the assumptions in the theorem, we know that both C and $\bar{C}$ are not empty. Let $f_\perp$ denote the function that is everywhere undefined. We consider two cases:
Case 1: There is no $a \in C$ such that $\varphi_a = f_\perp$.

Let c be any element of C, and define the function $r : \mathbf{N} \times \mathbf{N} \hookrightarrow \mathbf{N}$ by:

$$r(x,y) = \begin{cases} \varphi_c(y) & \text{if } \varphi_x(x) \downarrow \\ \perp & \text{otherwise.} \end{cases}$$

It is clear from the Church-Turing thesis that r is computable. By the s-m-n theorem, there is a total computable function s_r such that

$$\varphi_{s_r(x)}(y) = r(x,y).$$

Moreover, by the definition of the function r, the following two properties hold:

If $\varphi_x(x) \downarrow$, then $\varphi_{s_r(x)} = \varphi_c$, and therefore $s_r(x) \in C$.

and

If $\varphi_x(x) \uparrow$, then $\varphi_{s_r(x)} = f_\perp$, and therefore $s_r(x) \notin C$.

Thus, we have proved that s_r is a reduction from the set $S = \{x \in \mathbf{N} | \varphi_x(x) \downarrow\}$ to the set C, and therefore C is undecidable.
Case 2: There is an $a \in C$ such that $\varphi_a = f_\perp$.

Then, since C is an index set, there is no a in $\bar{C}$ such that $\varphi_a = f_\perp$. So, we can use the argument in Case 1 to prove that $\bar{C}$ is undecidable. Since the complement of a decidable set is always decidable, it follows that C is undecidable. $\bowtie$