Chapter 4

Computability

So far we have seen (i) how to write programs as descriptions of how to compute a function, (ii) how to unambiguously describe such computations, (iii) how to check whether a program correctly computes a function, and (iv) how to reason abstractly about the efficiency of a program. We now ask a more abstract question: what functions can be computed and what cannot.

It may seem strange to be discussing the tasks that computers cannot do. To understand why Turing and his contemporaries were so focused on this issue we need to realise – as they did – the implications of universality. This is the idea that a concept, system or machine is sufficiently powerful that it is able to capture itself. In 1931 Gödel showed how to embed formal logic into formal logic and generate contradictions. In 1936 Turing showed that his Turing Machines were also able to describe themselves.

In this part of the course we will discuss one of the most surprising results in computing: most tasks we would like to undertake cannot be performed by computers! Informally we will say that a problem has a computable solution if we could in principle write a computer program to solve the problem. If instead we show that no one could ever write a computer program to solve the problem, then the problem is uncomputable. As you might imagine, in many cases it is not yet known into which of these two classes a problem should be placed.

Before we begin we point out that our discussions are limited to computing functions assuming some (very reasonable) restrictions on the model of computation available. It is possible to build different notions of computation when relaxing these assumptions and you can find such discussions online. To understand why the first point about functions is important,
think about the following:

- Can a computer understand emotions?
- Can a computer dream?
- Can a computer create beauty?
- Can a computer predict the future?
- Can a computer fall in love?

It is very easy to see all sorts of things that computers cannot do. Perhaps we can debate some of the above points, but this is because they are poorly defined. Here we force ourselves to phrase this question in terms of computing a function. If any of the above questions can be represented by the task of computing a function then it fits into our discussion of computability.

**Learning Outcomes**

At the end of this Chapter you will be able to:

- Define what it means for a function to be *computable* or *uncomputable*
- Describe how programs can be encoded as natural numbers and how this result means that the set of programs is countable
- Recall the diagonalisation proof that there are uncountably many functions from $\mathbb{N} \to \mathbb{N}$
- Describe the notion of *decidability* and how it relates to *computability* as well as the notions of *decision procedure*, *partially decidable* and *characteristic function*
- Prove simple properties of computable/decidable functions
- Recall the definition of the *universal function* and *universal program*, what they means, and their implications to the expressiveness of a language
- Describe the *Halting Problem* and outline the proof that the problem is uncomputable
- Recall the *Church-Turing Thesis* and its implications to the expressiveness of a language
Some Exercises

I have written some exercises here that capture the previous learning outcomes. Notice that the learning outcomes here are all about explaining, describing and recalling. You are not supposed to be able to answer these exercises until after you have read the rest of this chapter.

Exercise 4.1

Explain the following concepts: computability, uncomputability, decidability, and partial-decidability. For each class of problem, give an example problem which is in the class.

Exercise 4.2

Pick your favourite programming language (if you do not have a favourite one pick Java) and argue that it cannot compute all functions.

Exercise 4.3

Pick your favourite programming language (if you do not have a favourite one pick Java) and argue that it is equally expressive to the while language. Note that equally goes in both directions. How would you demonstrate this without appealing to the Church-Turing Thesis?

4.1 Computable Functions

We begin by defining what we mean by computable functions and showing how to build computable functions from other computable functions.

4.1.1 Computable Functions $\mathbb{N} \to \mathbb{N}$

Let us simplify the set of problems to just those that are concerned with functions from natural numbers to natural numbers. In our while language we will assume that both the input argument and the output result are passed in the variable $x$. Recall our previous argument (Section 1.4) that other functions of interest (e.g. $\mathbb{Z}^n \to \mathbb{Z}^m$) can be mapped into $\mathbb{N} \to \mathbb{N}$ and therefore we can concentrate on $\mathbb{N} \to \mathbb{N}$, we repeat this argument here in a slightly different context.

\footnote{In contrast to the previous sections which were often about applying.}
CHAPTER 4. COMPUTABILITY

Definition 4.1.1

A function $f : \mathbb{N} \to \mathbb{N}$ is computable if, and only if, there is a while program $S$, such that for all states $s$, and $n \in \mathbb{N}$ with $s(x) = n$, then there is a state $s'$ such that

$$< S, s > \Rightarrow^* s' \quad \text{and} \quad s'(x) = f(n)$$

where $\Rightarrow^*$ is the transitive closure of $\Rightarrow$.

In short, a function from the natural numbers to the natural numbers is computable if we can write a while program to implement the function. To be extra-precise, we might say that such a function is while-computable. As we shall see – in Thesis 4.7.1 – there is usually no need to make this distinction.

Important

The above definition allows for computable partial functions. If $f(n)$ is undefined then the program $S$ should be non-terminating on $n$.

Example 4.1.2

The identity function $f(n) = n$ is computable. This is because skip is a while program, which leaves the value of $x$ unchanged.

In particular, in Definition 4.1.1, we take $S = \text{skip}$, and then after one step $s = s'$, and thus the value of $x$ is unchanged.

Example 4.1.3

The function $f(n) = n + 1$ is computable. We can invoke Definition 4.1.1 with the while program $S$ as $x := x + 1$ because

$$\langle S, [x \mapsto n] \rangle \Rightarrow [x \mapsto n + 1]$$

Exercise 4.4

Show that the following functions are computable by finding a program $S$ that computes them:
4.1. COMPUTABLE FUNCTIONS

- \( f(n) = 5 \)
- \( f(n) = n^2 \)
- \( f(n) = n! \)
- \( f(n) = \begin{cases} 1 & \text{if } n > 0 \\ 0 & \text{otherwise} \end{cases} \)

Exercise 4.5

Suppose \( f \) and \( g \) are computable functions. By writing a \texttt{while} program prove that \( f \circ g \) is also computable.

\textbf{Hint:} If \( f \) and \( g \) are computable, then there must be programs \( S_f \) and \( S_g \) satisfying Definition 4.1.1.

Exercise 4.6

Suppose that \( f \) is a computable function. By writing a \texttt{while} program show that for any \( k \) the function \( f^k \) is computable.

4.1.2 Computable Functions \( \mathbb{N}^m \rightarrow \mathbb{N}^n \)

We can extend the notion of computability to functions that take and return vectors of natural numbers as their argument and result instead of just a single natural number. Let us begin with a simple example: functions of type \( \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \). As before, the technique we use is to treat the two arguments as if they were a single natural number encoded using a pairing bijection \( \phi_X \). Here we assume the pairing bijection introduced in Section 1.4.1, but any bijection \( \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \) can be used.

\textbf{Definition 4.1.4}

We say that a function \( f : (\mathbb{N}, \mathbb{N}) \rightarrow \mathbb{N} \) is \textit{computable}, if, and only if, the function \( g : \mathbb{N} \rightarrow \mathbb{N} \) is computable using Definition 4.1.1, where

\[ f(x, y) = g(\phi_X(x, y)) \]

Likewise we can consider functions where the result is not a natural number.
Definition 4.1.5

We say that a function \( f : \mathbb{N} \to (\mathbb{N} \times \mathbb{N}) \) is computable, if, and only if, the function \( g : \mathbb{N} \to \mathbb{N} \) is computable using Definition 4.1.1, where
\[
 f(x) = \phi_x^{-1}(g(x))
\]

Generalizing the two Definitions 4.1.4 and 4.1.5, we can define computability on \( \mathbb{N}^m \to \mathbb{N}^n \) using iterated pairing and unpairing bijections.

Definition 4.1.6

A function \( f : \mathbb{N}^m \to \mathbb{N}^n \) – with \( n, m \geq 1 \) – is computable if, and only if, there is a function \( g : \mathbb{N} \to \mathbb{N} \) which is computable in the sense of Definition 4.1.1, such that
\[
 g(\phi_x(x_1, \phi_x(x_2, \ldots \phi_x(x_{n-1}, x_n)))) = \\
 (\phi_x(y_1, \phi_x(y_2, \ldots \phi_x(y_{m-1}, y_m) \ldots ))
\]

where,
\[
 f(x_1, x_2, \ldots, x_{n-1}, x_n) = (y_1, y_2, \ldots y_{m-1}, y_m)
\]

As we saw before, this idea can be extended to other data structures. The most simple other data structure we care about is the integers.

Exercise 4.7

Define what it means for a function \( f : \mathbb{Z} \to \mathbb{Z} \) to be computable. What kind of bijection do we need? Do the same for \( f : \mathbb{Z}^m \to \mathbb{Z}^n \) for \( n, m \geq 1 \).

4.2 Counting Programs

In this section we show that we can count while programs. This is an important step in showing that there are uncomputable functions as later we will see that we cannot count the functions we want to compute.

We introduce coding functions for the statements of the while language. Firstly we introduce bijections between arithmetic and boolean expressions and \( \mathbb{N} \), and then we show how these can be used to code statements in general.
4.2. COUNTING PROGRAMS

Recall that the data type of Arithmetic Expressions ($A\text{Exp}$) is defined by five cases:

$$a ::= x \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$$

The encoding we require for this will be called $\phi_A$ and is as follows:

$$\phi_A : A\text{Exp} \to \mathbb{N}$$

<table>
<thead>
<tr>
<th>Expression</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$5 \times n$</td>
</tr>
<tr>
<td>$x$</td>
<td>$1 + 5 \times x$</td>
</tr>
<tr>
<td>$a_1 + a_2$</td>
<td>$2 + 5 \times \phi_A(a_1, \phi_A(a_2))$</td>
</tr>
<tr>
<td>$a_1 - a_2$</td>
<td>$3 + 5 \times \phi_A(a_1, \phi_A(a_2))$</td>
</tr>
<tr>
<td>$a_1 \times a_2$</td>
<td>$4 + 5 \times \phi_A(a_1, \phi_A(a_2))$</td>
</tr>
</tbody>
</table>

This encoding has some caveats:

- We should have transformed the program text $n$ into a $\mathbb{N}$, but this is straightforward.
- Likewise, we should have translated every identifier into $\mathbb{N}$. We could do this by thinking in terms of an ASCII encoding of the string, as a (large) bit pattern, which we treat as a natural number.
- Notice that we are using our pairing bijection ($\phi$, see Section 1.4.1), as well as the recursively defined encoding function for $A\text{Exp}$ ($\phi_A$).

The existence of the bijection $\phi_A$ is important as it means that there are precisely as many arithmetic expressions as there are natural numbers. Recall that we call sets with a bijection to $\mathbb{N}$ countably infinite.

To understand how this coding function is working note that numbers are mapped into numbers of the form $5n$, variables are mapped into numbers of the form $5n + 1$, addition expressions are mapped into numbers of the form $5n + 2$, and so on. This is a similar trick to the one where we encoded $\mathbb{Z}$ by mapping negative numbers to odd numbers and positive numbers to even numbers.

We can also encode boolean expressions as natural numbers. Recall that the data type of Boolean Expressions ($B\text{Exp}$) is defined by six cases:

$$b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2$$

However, notice that the first two cases do not contain either arithmetic or boolean sub-expressions. Therefore, we only need to ‘reserve’ one space for each of them.
\( \phi_B : \mathbf{BExp} \to \mathbb{N} \)
\( \phi_B(\text{true}) = 0 \)
\( \phi_B(\text{false}) = 1 \)
\( \phi_B(a_1 = a_2) = 2 + 4 \times \phi(\phi_A(a_1), \phi_A(a_2)) \)
\( \phi_B(a_1 \leq a_2) = 3 + 4 \times \phi(\phi_A(a_1), \phi_A(a_2)) \)
\( \phi_B(\neg b) = 4 + 4 \times \phi_B(b) \)
\( \phi_B(b_1 \land b_2) = 5 + 4 \times \phi(\phi_B(b_1), \phi_B(b_2)) \)

**Exercise 4.8**

Using \( \phi \), \( \phi_A \) and \( \phi_B \), complete the following recursive function \( \phi_S \) which is a bijection from \( \mathbf{Stm} \) to \( \mathbb{N} \):

\[
\phi_S : \mathbf{Stm} \to \mathbb{N}
\]
\( \phi_S(\text{skip}) = 0 \)
\( \phi_S(\text{while } b \text{ do } S) = 1 + 4 \times \phi(\phi_B(b), \phi_S(S)) \)
\( \phi_S(x := a) = ? \)
\( \phi_S(S_1; S_2) = ? \)
\( \phi_S(\text{if } b \text{ then } S_1 \text{ else } S_2) = ? \)

Argue that this means there are only countably many programs in \( \text{while} \).

Given the result from Exercise 4.8 and the definitions from Section 1.4.2 showing that we can encode lists as natural numbers, it is possible to encode the transitions of Table 1.5 in \( \mathbb{N} \). Each transition is then a (very complicated) arithmetic transformation in \( \text{while} \). The \( \text{while} \) program which transforms the natural number of a configuration in Table 1.5 to the next one is called the **Universal Program**. We can think of it as an interpreter for \( \text{while} \) written *in* \( \text{while} \). We will return to this when we discuss universality later but I thought it worthwhile mentioning it here.

### 4.3 The Existence of Uncomputable Functions

In this section we present a proof of the existence of uncomputable functions. Note that this proof is *nonconstructive* i.e. at the end of it we do not have an example of an uncomputable function to satisfy us that such a function exists.

We start by exploring the cardinality of \( \mathbb{N} \to \mathbb{N} \).
Lemma 4.3.1

There are uncountably many functions from \( \mathbb{N} \to \mathbb{N} \).

The proof technique used in the following proof is actually as important as the result: it is an example of Cantor’s Diagonalisation Method, which is central to many results about computability. It is a proof by contradiction. From the maths course you may have learnt that such proofs are to be avoided where possible, this is a case where it is not possible.

Proof of 4.3.1

Let us assume that there are countably many functions from \( \mathbb{N} \to \mathbb{N} \). Firstly we will show that there must be infinitely many functions. Each of the constant functions \( f_0(n) = 0, \ f_1(n) = 1, \ldots f_k(n) = k, \ldots \) is different, and there are countably infinitely many of them.

If there are countably infinitely many functions of type \( \mathbb{N} \to \mathbb{N} \), then we can enumerate them. If there’s countably infinitely many items in a set \( A \) then there is a bijection \( \phi : A \to \mathbb{N} \). By an enumeration we mean that we can place the items from \( A \) in sequence from 0 upwards, using the bijection.

If the set of functions is countably infinite then we can place the functions in order \( f_0, \ f_1, \ldots, \ f_k, \ldots \). Complete details of this set of functions can be expressed by the following infinite table where each column represents the value of each function on a particular input.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_0 )</td>
<td>( f_0(0) )</td>
<td>( f_0(1) )</td>
<td>( f_0(2) )</td>
<td>( f_0(3) )</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>( f_1 )</td>
<td>( f_1(0) )</td>
<td>( f_1(1) )</td>
<td>( f_1(2) )</td>
<td>( f_1(3) )</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>( f_2 )</td>
<td>( f_2(0) )</td>
<td>( f_2(1) )</td>
<td>( f_2(2) )</td>
<td>( f_2(3) )</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>( f_3 )</td>
<td>( f_3(0) )</td>
<td>( f_3(1) )</td>
<td>( f_3(2) )</td>
<td>( f_3(3) )</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
| ...    | ...  | ...  | ...  | ...  | ...  | ...

Now, because we have assumed that we have a bijection between the functions of type \( \mathbb{N} \to \mathbb{N} \) and the natural numbers, we know that each of the functions \( f_0, \ f_1, \ldots, \ f_k, \ldots \) must be different from each of the others. This is an important point.
We will now construct another function, \( f_{\text{new}} \), which is different from all of the \( f_k \)'s, and hence show that our enumeration was incomplete.

We systematically construct the function \( f_{\text{new}} \), so that it is different to each \( f_k \). An easy way to do this is by adding 1 to the value of \( f_k(k) \), which are the diagonal terms in the above table.

To be explicit, we let \( f_{\text{new}} : \mathbb{N} \rightarrow \mathbb{N} \) be the function:

\[
 f_{\text{new}}(n) = f_n(n) + 1
\]

We can now show that \( f_{\text{new}} \neq f_k \) for any \( k \).

Suppose that \( f_{\text{new}} = f_k \) then by extensionality:

\[
 \forall (n \in \mathbb{N}). \; f_{\text{new}}(n) = f_k(n)
\]

In particular, this must be true for \( n = k \), which would mean that \( f_{\text{new}}(k) = f_k(k) \), but by our construction, \( f_{\text{new}}(k) = f_k(k) + 1 \neq f_k(k) \). Hence \( f_{\text{new}} \neq f_k \), as \( f_{\text{new}} \) is not the same as \( f_k \) for some input.

To recap: we have shown that

1. There are infinitely many functions \( \mathbb{N} \rightarrow \mathbb{N} \); and
2. If we assume that we can enumerate all of the functions in \( \mathbb{N} \rightarrow \mathbb{N} \), it turns out that we cannot, because there is a missing function.

Therefore we have uncountably many functions \( \mathbb{N} \rightarrow \mathbb{N} \).

\[ \square \]

Notice that both parts are necessary. If a set is finite then it is trivially countable. There are many different ways that we could have chosen to construct the function \( f_{\text{new}} \).

**Exercise 4.9**

Give another function which is not enumerated, and that is different to \( f_{\text{new}} \). How many such functions are there?

Now, as we have previously shown that there are only countably many programs, there are many, many, functions for which there is no \texttt{while} program. We can state this formally as the following Corollary:
4.4. DECIDABILITY

Corollary 4.3.2

There are non-computable functions of type $\mathbb{N} \rightarrow \mathbb{N}$.

Notice that although we have been expressing our problem in terms of a very limited range of functions, we have previously equipped ourselves with a way to encode and decode any data structure as a natural number using bijections. Therefore, an additionally corollary of Corollary 4.3.2 is that there are non-computable functions of any type that can be encoded in $\mathbb{N} \rightarrow \mathbb{N}$.

Exercise 4.10

Give proof sketches for the following statements:

1. There are a countably infinite number of functions of type $\mathbb{N} \rightarrow \mathbb{B}$
2. There are not a countably infinite number of functions of type $\mathbb{B} \rightarrow \mathbb{N}$

A proof sketch is something that gives enough details to construct the corresponding proof, but leaves some details out. The level of detail required is subjective. For (2) note that the above proof relied on the fact that the functions had an infinite input domain.

Exercise 4.11

Explain why there must be functions of type $(\mathbb{N}, \mathbb{N}) \rightarrow \text{Bool}$ which cannot be written in java.

4.4 Decidability

That we can partition the set of functions of type $\mathbb{N} \rightarrow \mathbb{N}$ into two distinct parts – computable and non-computable – should alert us to the fact that this partitioning is also possible for functions with other types.

We can also use simpler data structures than the natural numbers.
Definition 4.4.1
We say that a function \( P : \mathbb{N} \to \mathbb{B} \) is computable, if, and only if, the function \( f : \mathbb{N} \to \mathbb{N} \) is computable using Definition 4.1.1, where
\[
P(x) = \begin{cases} 
\text{True} & \text{if } f(x) = 1 \\
\text{False} & \text{if } f(x) = 0 
\end{cases}
\]
Functions which return boolean values are called predicates.

The functions that are computable using the above Definition 4.4.1 are given a special name: they are called decidable. In short, a decidable predicate is a computable one.

Definition 4.4.2
The predicate \( P \) is decidable if, and only if, there is a computable function \( f : \mathbb{N} \to \mathbb{N} \) such that:
\[
f(x) = \begin{cases} 
1 & \text{if } P(x) \text{ holds} \\
0 & \text{if } P(x) \text{ doesn’t hold} 
\end{cases}
\]
The total function \( f \) is called the characteristic function of \( P \), and the associated program in \texttt{while} is a decision procedure for \( P \).
Any predicate which is not decidable is undecidable.

Lemma 4.4.3
If \( P \) and \( Q \) are decidable predicates, then all of the following are also decidable:
\[ 
\begin{itemize} 
\item \( \neg P \); 
\item \( P \land Q \); 
\item \( P \lor Q \); and 
\item \( P \to Q \). 
\end{itemize} 
\]
Proof
We only consider the \( \neg P \) case and leave the rest to be proved in Exercise 4.12. If \( P \) is decidable then it has a characteristic function \( f_P \) and a decision procedure \( D_P \). Let
\[
f_{\neg P}(x) = 1 - f_P(x)
\]
be the characteristic function for \( \neg P \) and \( D_{\neg P} = D_P; x := 1 - x \) be the associated decision procedure. We could also have framed this as composing two functions and invoked the result of Exercise 4.5.
Before we go any further, it is worth reiterating that the associated function \( f \) for the predicate \( P \) in Definition 4.4.2 is called the Characteristic Function.

**Definition 4.4.4**

The Characteristic Function \( \chi_P \) of a predicate \( P \) is defined to be:

\[
\chi_P(x) = \begin{cases} 
1 & \text{if } P(x) \text{ holds}, \\
0 & \text{if } P(x) \text{ doesn’t hold}. 
\end{cases}
\]

**Exercise 4.12**

By writing the associated programs, or otherwise, prove that each of the predicates defined in Lemma 4.4.3 are also decidable.

In general, a function that is defined in terms of many other functions is computable as long as all of those functions are computable and the check required to determine which case we are in is also decidable.

**Theorem 4.4.5**

If each function \( f_i \) is computable, and each predicate \( P_i \) is decidable, then \( g \), defined as

\[
g(x) = \begin{cases} 
f_1(x) & \text{if } P_1(x) \text{ holds} \\
f_2(x) & \text{if } P_2(x) \text{ holds} \\
\vdots & \vdots \\
f_n(x) & \text{if } P_n(x) \text{ holds} 
\end{cases}
\]

is also computable. Note that we assume that we check the conditions from top to bottom, so the second case assumes that \( P_1(x) \) does not hold etc.

**Proof**

We can rewrite \( g \) using the characteristic functions of \( P_i \) as follows:

\[
g(x) = \chi_{P_1}(x)f_1(x) + \chi_{P_2}(x)f_2(x) + \cdots + \chi_{P_n}(x)f_n(x)
\]

Thus because addition and multiplication are computable, we can use substitution to conclude that \( g \) is computable.
Lemma 4.4.6

The predicate \( x \mid y \) (which tests whether \( x \in \mathbb{N} \) is divisible by \( y > 0 \)) is decidable.

Proof

We know that calculating the division and remainder is computable (see Example 1.2.1). Suppose that this program is denoted by \( D \). Then the characteristic function for this predicate is:

\[
D; \text{ if } r=0 \text{ then } 1 \text{ else } 0
\]

Exercise 4.13

Show that the predicate even : \( \mathbb{N} \rightarrow \mathbb{B} \) (that returns true when given an even number and false otherwise) is decidable.

Hint: You may find your answer to Exercise 1.7 useful here.

Exercise 4.14

Prove that the predicate prime : \( \mathbb{N} \rightarrow \mathbb{B} \) – which tests whether a natural number is prime – is decidable.

Exercise 4.15

Prove the decidability of the predicate isNFib : \( (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{B} \) that takes numbers \( i \) and \( n \) and returns whether \( n \) is the \( i \)th Fibonacci number.

Notice that the program for a decidable predicate always terminates, i.e. the predicate is a total function. There is a related concept – partially-decidable – which describes the situation where we can determine whether something is true, but where the converse cannot be determined.

Definition 4.4.7

The partial function \( P \) is partially decidable if, and only if, there exists a computable partial function \( f : \mathbb{N} \rightarrow \mathbb{N} \) with

\[
f(x) = \begin{cases} 
1 & \text{if } P(x) \text{ holds} \\
\text{undefined} & \text{if } P(x) \text{ doesn’t hold}
\end{cases}
\]

The partial function \( f \) is called the partial characteristic function of \( P \), and the associated program in while is a partial decision procedure for \( P \).
Note that the usual way to represent ‘undefined’ with a program is to fail to terminate.

**Exercise 4.16**

By writing the partial decision procedure or otherwise, prove that any decidable predicate is also partially decidable.

This notion of partial decidability (or semi-decidability as it is sometimes called) may seem unhelpful, but there are famous problems that are only partially decidable. We list some interesting undecidable and partially decidable problems below.

- The Halting Problem (see Section 4.6) is partially decidable.
- Determining whether a context-free grammar is *universal*, e.g. generates all strings, or is *ambiguous*, is undecidable.
- The problem of checking the validity of statements in first-order logic is partially decidable. This fact does not prevent us writing very effective theorem provers for first-order logic.
- Hilbert’s tenth problem, the problem of deciding whether a Diophantine equation (multivariable polynomial equation) has a solution in integers, is undecidable.

**4.5 Universality of Computation**

We now discuss what it means for a model of computation to be *universal*.

**4.5.1 The functions $\gamma$ and $\eta_i$**

We now introduce some more theoretical results that we rely on later related to the idea that programs are countable. Firstly, we capture the result from Exercise 4.8 as a theorem.
Theorem 4.5.1

*The set of while programs is effectively countably infinite.*

I have not formally introduced this notion of *effectively countable*. The term *effective* in maths just means that we have a method for doing it. So if a set is *effectively countable* then we have a procedure for counting the elements of the set.

**Proof**

We have already proved much of this in Exercise 4.8. To show the *effectivity* of the process, we need to show that we can write a *while* program to undertake this task.

The easiest way to establish this is to write a program in some other programming language to compute $\phi_S$ and appeal to the Church-Turing Thesis (introduced later in Section 4.7) which tells us that all programming languages are equivalent. In a previous iteration of the course a Haskell program was provided to achieve the second this second point and this will be available on the course website.

$\square$

(* ) Exercise 4.17

Write your own encoding of $\phi_S$ in your favourite programming language. You will need to represent *while* programs internally and handle the previous caveats about $\phi_A$.

We introduce some different terminology to refer to the number we get when we encode a *while* program.

**Definition 4.5.2**

To obtain the *index*, *code number* or *Gödel number* of a *while* program $S$, we will use $\gamma(S) = \phi_S(S)$. Note that the type of $\gamma$ is $\text{Stmt} \rightarrow \mathbb{N}$.

We call this the *index* as it is the index into the enumeration of *while* programs. Henceforth we will keep $\gamma$ and (thus) $\gamma^{-1}$ fixed for the rest of these notes\(^2\).

\(^2\)The use of $\gamma$ is a reminder that Gödel’s key insight lies in his using codings into the natural numbers in his seminal work.
Example 4.5.3

Based on the expected answer to Exercise 4.8 we can write down the code number for various programs:

- The code number for \texttt{skip} is 0
- The code number for \texttt{skip;skip} is 3
- The code number for \texttt{while true do skip} is 1
- The code number for \texttt{while false do skip} is 5
- The code number for \texttt{while true do skip;skip} is 25

Because $\gamma$ is a bijection it has an inverse $\gamma^{-1}$ which is also a bijection. Thus we have a way to turn natural numbers into programs in \texttt{while}.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Important} \\
\hline
I repeat this important point. The bijection $\gamma$ means that we have (i) a function to turn \texttt{while} programs into natural numbers, and (ii) a function to turn natural numbers into \texttt{while} programs i.e. $\gamma^{-1}$. \\
\hline
\end{tabular}
\end{center}

Exercise 4.18

After answering Exercise 4.8 we should be able to code and decode programs and code numbers.

- What are the code numbers for
  - $x := x + 1$ - what do you have to assume about $\phi_A(x)$?
  - \texttt{skip;skip;skip}
  - \texttt{if true then skip else skip}
- What are the program with code numbers 2 and 18?

Exercise 4.19

If $n \neq m$ what can you say about the two programs $\gamma^{-1}(n)$ and $\gamma^{-1}(m)$?

The other feature we will need is a way to refer to the unary function associated with a program whose index or code number is $i$. We will henceforth use the notation $\eta_i$. Recall that we can use coding arguments to lift all of these arguments to non-unary functions.
Definition 4.5.4

Suppose that the program $S$ is the value of $\gamma^{-1}(i)$. Then we define $\eta_i$, the function associated with the $i$-th program as follows.

$$\eta_i(k) = \begin{cases} 
  s'(x) & \text{if } \exists n, s'. \text{ with } <S, s[x \mapsto k] > \Rightarrow^n s' \\
  \text{undefined} & \text{otherwise}
\end{cases}$$

It is important to understand what this is doing. We explicitly define the function $\eta_i$ to be the one that takes the program $S = \gamma^{-1}(i)$ and runs it on some input, returning the output if it exists and being undefined otherwise. The undefined choice is necessary when $S$ is non-terminating.

Henceforth, we will treat $\eta_i$ as fixed for each $i$, i.e. $\eta_i$ is the unary function associated with the while program whose index is $i$.

Lemma 4.5.5

The function $h : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ defined as $h(i) = \eta_i$ is not injective.

Proof

Suppose $h(i) = h(j)$. If $h$ was injective then we should be able to show that $i = j$, but we will show that this is not true.

Suppose that $S = \gamma^{-1}(i)$, i.e. $S$ is the program with index $i$. Then the program skip; $S$ also implements the same function. But the index for the modified program is

$$\gamma(\text{skip}; S) = 2 + 4(2^{\gamma(\text{skip})}(2\gamma(S)+1) - 1) = 2 + 4((2i+1) - 1) = 8i + 2.$$ 

and $8i + 2 \neq i$. This shows that in general, $\eta_i = \eta(8i+2)$.

Important

It is important to recall the difference between a function and a program. A program computes exactly one function but a function can be computed by many functions. Therefore, whilst every program has a unique index given to it by $\gamma$, the function that this program computes is not unique.
The importance of Lemma 4.5.5 is that it shows that there are many programs which implement the same computable function. An alternative proof would have been to show that there are two inputs $i$ and $j$ such that $i \neq j$ and $h(i) = h(j)$. We already have this information from Example 4.5.3 as $0 \neq 3$ but $\gamma^{-1}(0) = \text{skip}$, $\gamma^{-1}(3) = \text{skip;skip}$, and both programs behave in exactly the same way on all inputs.

**Exercise 4.20**

How many different programs are there for the same function?

### 4.5.2 The Universal Function and Universal Program

We are now going to meet the *Universal Program*, which is a program which can simulate every other program. Recall that we already met this idea on page 102. Let us begin by defining the *universal function* $\psi_U$ for unary functions\(^3\).

**Definition 4.5.6**

We define the *Universal Function* $\psi_U : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$ for unary computable functions as:

$$\psi_U(a, b) = \eta_a(b)$$

where $a$ is the code number of function $\eta_a$.

Fairly clearly, this single function $\psi_U$ captures the behaviour of every unary computable function, because by selecting a value of $a$, we obtain the unary computable function $\eta_a$. To be more explicit, because every unary function can be encoded as a natural number we can provide this encoding $a$ along with some input $b$ to $\psi_U$ to simulate applying that function to $b$.

**Exercise 4.21**

Show that by choosing $a$ correctly, $\psi_U$ can implement the following functions.

- The identity function $f(x) = x$.
  
  **Hint:** Show that $\eta_0$ is the identity function.

---

\(^3\)The subscript $U$ is for *universal* not for *unary*
• The constant function \( f(x) = 2 \)
  **Hint:** You may assume that \( \gamma(x := 2) = 51791395714760704 \).

• The increment function \( f(x) = x + 1 \)
  **Hint:** You may assume that
  \[
  \gamma(x := 1 + x) = 707594314453383905280
  \]
  Why have I chosen \( x := 1 + x \) instead of \( x := x + 1 \)?

As you would expect by now, we can generalize the Universal function to one for \( n \)-ary computable functions

**Definition 4.5.7**

The universal function for \( n \)-ary computable functions is the \((n+1)\)-ary function \( \psi_U^{(n)} \) defined by

\[
\psi_U^{(n)}(i, x_1, x_2, \ldots, x_n) = \eta_i^{(n)}(x_1, x_2, \ldots, x_n)
\]

**Theorem 4.5.8**

The universal function \( \psi_U^{(n)} \) is computable.

**Sketch Proof of 4.5.8**

Let us consider \( n = 1 \), then the procedure for computing \( \psi_U(i, x) \) is as follows.

1. Find the **while** program associated with index \( i \), which is \( P_i = \gamma^{-1}(i) \). As we have already seen, this is a computable operation.
2. Next we simulate the execution of the program \( P_i \), step-by-step. To do this we must code the transitions of Table 1.5 using natural numbers as described on page 102.
3. If and when the computation stops, the result will be in held in the value of variable \( x \) in the final state.

For \( n > 1 \) we need to go via coding functions.

\[\square\]
This is the reason we gave the complete formal description of the \texttt{while} programming language earlier. A full version of this proof will be very long, and little further insight is to be gained. We therefore omit it.

**Exercise 4.22**

If $P$ and $Q$ are partially-decidable, which of the following are also partially-decidable?

- $\neg P$;
- $P \land Q$;
- $P \lor Q$; and
- $P \Rightarrow Q$.

Do you get the same answers if you use the Universal Program to simulate the execution of these semi-decision procedures?

## 4.6 The Halting Problem

Section 4.3 proved that uncomputable functions exist but did not give an example of an uncomputable function. In this section we meet the most famous uncomputable function: the halting problem.

The halting problem is analogous to the famous Liar Paradox, which we could state as:

“\textbf{This sentence is false.}”

The critical feature of this paradox is that we are mixing the feature being described (“this sentence”) with a description of its properties (“is false”). We are now going to follow Gödel and Turing in embedding the Liar Paradox into mathematics and computing.

### 4.6.1 The Halting Problem: An informal Argument

By now I expect that you will all have inadvertently written a program that fails to terminate. One of the most irritating features of this is that you can never be quite sure that the program is going to run for ever, rather than that the program is just taking a long time to compute something complicated. Wouldn’t it be nice to be able to detect this before we run the program? Unfortunately, as we shall see, this is \textit{not} possible (in general).
In this section we will outline an informal argument that we cannot write a program to detect when another program will terminate with a given input. To make this concrete, let us assume that the ‘halt-tester’ program, written in while is the program HALT, and that this takes the program $p$ (or to be more precise the code number of $p$) and the program’s input $n$ as inputs (as a pair in variable $x$) and outputs either 0 or 1 in variable $x$, representing false and true respectively. In other words we have assumed that the predicate $\text{halts}(p, n)$ is decidable.

We can now test any program $p$ with input $n$ and say for sure whether it terminates or not by running the program HALT. For example, to test the program for the identity function ($S = \text{skip}$) within the program:

$$x := 2; \quad \text{HALT}$$

The assignment of $x$ to 2 is because $\phi(S, 1) = 2$.

The next program to define is SELF, this takes a program $p$ as input and returns true ($x = 1$) if the program halts when its input is itself, and false ($x = 0$) otherwise. We can define SELF as:

$$z := x; \quad y := 1; \quad \text{while } 1 \leq z \text{ do } (y := y \times 2; \quad z := z - 1); \quad x := (2 \times x + 1) \times y - 1; \quad \text{HALT}$$

The first line above assigns $y$ the value $2^p$, and the second line calculates $\phi(p, p) = 2^p(2p + 1) - 1$ assigning this value to $x$ which is where HALT expects to find its input. Once more, the program above shows that if the predicate $\text{halts}(p, n)$ is decidable, then so is the predicate $\text{self}(p)$.

An alternative way to define the self predicate:

$$\text{self}(p) = \begin{cases} 
\text{True} & \text{if } \text{halts}(p, p) \\
\text{False} & \text{otherwise} 
\end{cases}$$

By Theorem 4.4.5, this is decidable provided only that the predicate halts is also decidable.

We now come to the clever bit. We define the following weird partial function:

$$\text{weird}(p) = \begin{cases} 
\text{undefined} & \text{if } \text{self}(p) \\
\text{True} & \text{otherwise} 
\end{cases}$$

\footnote{The assumption that there is a ‘halt-tester’ program will lead to a contradiction.}

\footnote{In this discussion we will often refer to a program and its code number interchangeably. Whenever we talk about passing a function to another function we implicitly mean that function’s code number.}
Inverting this, we can say that the predicate not-weird is partially decidable, as it returns true if self(p) is false, and is undefined otherwise.

Once more, the partial function weird is computable, because (assuming we could write the halt-tester program) we can write its program WEIRD as:

\[ \text{SELF}; \text{if } x = 1 \text{ then (while true do skip) else } x := 1 \]

We now come to a paradox, i.e. something that is both logically true and logically false. What happens when we supply the partial function weird with itself as input?

Using Equation 4.6.1, we see that

\[
\text{weird(weird)} = \begin{cases} 
\text{undefined} & \text{if self(weird)} \\
\text{True} & \text{if } \neg \text{self(weird)} 
\end{cases}
\] (4.1)

But

\[ \text{self(weird)} = \text{halt(weird, weird)} \]

There are now two cases:

**\text{halt(weird, weird) is true}** In this case we take the first branch of Equation 4.1, which goes into an infinite loop, i.e. it fails to terminate. But this is the effect of running the program weird using its own representation as input, and the halt-tester tells us this terminates. It is therefore a contradiction.

**\text{halt(weird, weird) is false}** In this case we take the second branch of Equation 4.1, which returns true, and thus running the program weird with itself as its input terminates. However, this contradicts the result given by the halt-tester, which is false. It is therefore also a contradiction.

Thus no matter whether the result is true or false, we have generated a contradiction. We have therefore shown that it is impossible to write a halt-tester program in our while language. We now make this a more formal argument.

4.6.2 The Halting Problem: Oracle Computing

What we have shown so far does not quite show that it is impossible to write a halt-tester, because maybe the problem lies in the expressiveness of the programming language, and perhaps using a different programming language
with extra features would have permitted us to write the halt-tester. We
will now show that even this is not possible!

Assume that we augment the while language with a new statement $\mathcal{H}$. We’ll display this is red to indicate that it is something a bit special. This means that the syntax of statements is now:

$$S ::= \mathcal{H} \mid x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S$$

(the syntax of arithmetic and boolean expressions remaining unchanged).

Because of the extra statement, we need to change the Gödel Numbering bijection from $\gamma$ to $\gamma_{\mathcal{H}}$, which we define as:

$$\gamma_{\mathcal{H}} : \text{Stm'} \to \mathbb{N}$$

$$\gamma_{\mathcal{H}}(\mathcal{H}) = 0$$

$$\gamma_{\mathcal{H}}(\text{skip}) = 1$$

$$\gamma_{\mathcal{H}}(\text{while } b \text{ do } S) = 2 + 4 \times \phi_B(b) \cdot \gamma_{\mathcal{H}}(S)$$

$$\gamma_{\mathcal{H}}(x := a) = 3 + 4 \times \phi(x, \phi_A(a))$$

$$\gamma_{\mathcal{H}}(S_1; S_2) = 4 + 4 \times \phi(\gamma_{\mathcal{H}}(S_1), \gamma_{\mathcal{H}}(S_2))$$

$$\gamma_{\mathcal{H}}(\text{if } b \text{ then } S_1 \text{ else } S_2) = 5 + 4 \times \phi_B(b) \cdot \phi(\gamma_{\mathcal{H}}(S_1), \gamma_{\mathcal{H}}(S_2)))$$

If the variables $p$ and $n$ are initialized with the Gödel Number of program $P \in \text{Stm'}$ and its input respectively, then the effect of executing the new statement is $\mathcal{H}$ is that variable $x$ is assigned the value 1 if program $P$ terminates with input $n$ and 0 otherwise. Execution of $\mathcal{H}$ always terminates.

Thus we can implement the predicate $H(p, n)$ with the program $\mathcal{H}$ directly (it accepts its arguments in $p$ and $n$, and returns a result in $x$, as usual). This predicate is total, because $\mathcal{H}$ always terminates.

We will now generate a contradiction as we did before. Let the predicate $W(p)$ be defined as:

$$W(p) = \begin{cases} \text{True} & \text{if } H(p, p) = 0 \\ \text{Undefined} & \text{otherwise} \end{cases}$$

Because there is a program for testing whether a program and input terminate ($\mathcal{H}$), there is a program for $W$, which we’ll refer to as $S_W$:

$$n := p; \mathcal{H}; \text{if } x = 0 \text{ then } x := 1 \text{ else while true do skip}$$

Now consider the result of applying the partial function $W$ to $w = \gamma_{\mathcal{H}}(W)$. There are two cases:

$H(w, w) = 1$ In this case $W(w)$ is undefined, i.e. the program $S_W$ fails to terminate with input $w$. But if $H(w, w) = 1$, then $W(w)$ is not undefined, and we have a contradiction.
4.6. THE HALTING PROBLEM

\[ H(w, w) = 0 \] In this case \( W(w) \) is \textbf{True}, i.e. the program \( S_W \) terminates with input \( w \), returning a value of 1 in variable \( x \). But if \( H(w, w) = 0 \), then \( W(w) \) does not terminate and is thus undefined, and we have a contradiction.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Important} \\
\hline
Be very careful indeed with “Oracle Computing”. Assuming that something magical happens and then concluding something astounding occurs is not convincing. Using Oracle Computing to show a contradiction is completely different; now we have shown that even if magic is permitted, it cannot add anything new to the properties of computing devices. \\
\hline
\end{tabular}
\end{center}

Using the results that we have shown so far, we now know that:

- The halting problem is a well-posed, and there is a function \( H : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N} \) corresponding to our specification.

- If we assume that a computer can implement it using an oracle \( H \), then we generate a contradiction.

- Because of the contradiction, we know that the function \( H \) cannot be computable (it has no corresponding program in \textbf{while}).

We have therefore shown that no Turing-equivalent computer can implement a halt-testing program. The notion of Turing-equivalence hasn’t been covered formally yet, but by appeal to the Church-Turing thesis we can argue that if we cannot consistently extend the \textbf{while} language with a halting construct then we cannot consistently extend any realistic programming language with one.

**Exercise 4.23**

How does the proof that the Halting Problem is undecidable rely on the Universal Program?

**Exercise 4.24**

Show that the problem of determining that a program halts is partially-decidable.
4.6.3 A Practical Look at Proving Termination

The previous part of the section might sound gloomy, perhaps we can never know if anything we run will ever halt. But I would like to stress that there is a difference between the following two statements:

1. There exists a program $P$ that decides whether every program $Q$ halts

2. For all programs $Q$ there exists a program $P$ that decides whether $Q$ halts

The first statement is the halting problem. The second statement says that given a program we can choose another program to decide whether the first program halts. This is clearly true as if $Q$ halts we can choose the program that returns true and if it doesn’t we can choose the program that returns false. This might look like cheating but the logical structure allows this.

More practically, look at the programs we have written so far in this course, do they halt? You should be reasonably confident that all of the programs that we intended to halt will halt. In Chapter 2 we met the idea of proving total correctness so we already have a technique for establishing termination (another name for halting).

The take-home message here is that whilst it is not possible in general to establish whether any program terminates. It is often possible in practice for many classes of interesting programs.

4.7 The Church-Turing Thesis

During the 1930s many different ways were found to define what we would now call the concept of computation. Amongst the better known are:

- Schönfinkel Combinators, 1924
- Church $\lambda$-Calculus, 1936
- Gödel-Kleene $\mu$-recursive functions, 1936
- Turing Turing Machines, 1936
- Post string-rewriting, 1943
- Shepherdson and Sturgiss URM, 1963
Many of these systems have fallen into obscurity, but the remarkable thing is that they all capture the same idea of computation. The technique we use to establish that each of these is defining the same concept of computation is to show that we can write an interpreter for one in the other.

To further elaborate this previous point let’s have a quick look at the Turing machine. A Turing machine is a finite device which performs operations on an infinite tape. The tape should be thought of as a set of cells indexed by \( \mathbb{Z} \), and each cell can hold a symbol from some finite alphabet \( \alpha \). The machine has a state (one of a finite number of states) and a finite set of rules that show how the machine makes transitions between states.

Provided that the Turing machine terminates then we need only represent a finite part of the tape, and we know how we can code each of these data structures as a natural number; thus we can program a Turing machine in \texttt{while}. To show how we could encode the behavior of \texttt{while} by setting up the rules and initial tape of a Turing Machine would require us to go into a considerable amount of pointless detail. It suffices to know that this is possible, and that details could be looked up on the internet.

The equivalence of all of the above definitions caused two of the above protagonists to propose the Church-Turing Thesis.

\textbf{Thesis 4.7.1 (Church-Turing)}

\begin{quote}
Any sensible definition of computation will define the same functions to be computable as any other definition.
\end{quote}

\textbf{Important}

We can paraphrase the Church-Turing Hypothesis as: “A function is computable whenever we can write a program to implement it.”

In the above we do purposefully do not specify a language in which the program should be written as this is the point of the Church-Turing Thesis, that all reasonably expressive languages can express all programs able to describe computable functions.

\textbf{Exercise 4.25}

Explain why the Church-Turing Thesis \textit{cannot} be proved.