Chapter 12

Computability

Gödel's paper has reached me at last. I am very suspicious of it now but will have to swot up the Zermelo-van Neumann system a bit before I can put objections down in black & white.

Alan Turing, letter to Max Newman, 1940

Hopefully by this point you are comfortable with programs and how they are evaluated, and should feel like with enough time and effort you would write a program to do just about anything. In this chapter, we consider the profound question of what problems can and cannot be solved by computing: are there problems that cannot be solved by *any* algorithm?

We will present an informal answer to this question here, and develop a more formal answer in Chapter **??**. Before getting to the question of computability, we introduce a similar question for declarative knowledge: are there true statements that cannot be proven by any proof?

12.1 Mechanizing Reasoning

An *axiomatic system* is a formal system consisting of a set of axioms and a set of inference rules. The goal of an axiomatic system is to codify knowledge in some domain, so that all true statements can be derived starting from the axioms and following the inference rules. A *complete* axiomatic system would be able to derive all true statements by starting from the axioms and following the inference rules. A *consistent* axiomatic system is one that can never derive a false statement by starting from the axioms and following the inference rules.

Humans have been attempting to develop axiomatic systems that codify logic for thousands of years. An early notable attempt was Aristotles *Organon* (in approximately 350BC). Aristotle developed *syllogisms*, rules of inference that codify logical deductions. For example:

$$\frac{\text{Every } A \text{ is a } P. \quad X \text{ is an } A.}{X \text{ is a } P.}$$

The statements above the line are the premises, and the statement under the line is the conclusion. The variables A, P, and X can be bound to any value. If the first two statements are true, then the inference rule states that the third statement must also be true. For example, binding A to human, P to mortal, and X to Gödel, we get:

$$\frac{\text{Every human is mortal.} \quad \text{Gödel is a human.}}{\text{Gödel is mortal.}}$$

Attempts to mechanize reasoning culminated in 1913 with the completion of Alfred North Whitehead and Bertrand Russells *Principia Mathematica*, three volumes comprising over 2000 pages that attempted to mechanize mathematical reasoning. Whitehead and Russell attempted to derive all true mathematical statements about numbers and sets starting from a set of axioms and formal inference rules.

12.1.1 Russell's Paradox

In doing this, they encountered several challenges, the most famous of which is now known as *Russell's paradox*. Suppose S is defined as the set containing all sets that do not contain themselves as members. For example, the set of all prime numbers does not contain itself as a member (since all its members are numbers), so it is a member of S. On the other hand, the set of all entities that are not prime numbers is a member of S. This set contains all sets, since a set is not a prime number, so it must contain itself. Is the set S a member of S?

There are two possible answers to consider: “yes” and “no”:

- Yes: suppose S is a member of S. Then, the set S contains itself. But, we defined the set S as the set of all sets that do not contain themselves as member. Hence, S cannot be a member of itself, and the statement that S is a member of S must be false.

- No: suppose S is not a member of S. Then, the set S does not contain itself. But, we defined the set S as the set of all sets that do not contain themselves as a member. So, if S is not a member of S, it does not contain itself, and it must be a member of set S. This is a contradiction, so the statement that S is not a member of S must be false.

This is a paradox! The “yes” answer and the “no” answer both make no sense.

Whitehead and Russell attempted to resolve this paradox by constructing their system to disallow the original definition. Their solution was to introduce types. Each set has an associated type, and a set can only contain members whose type is below the set type. A type zero set is defined as a set that contains only objects (that is, it cannot contain any sets as members). A type one set is a set that containing only objects and type zero sets. A type n set is a set that contains only objects and sets of type $n-1$ and below. With this definition, the paradox is resolved: the definition of S must now define S as a set of type k set containing all sets of type $k-1$ and below that do not contain themselves as members. Since S is a type k set, it cannot contain itself, since it cannot contain any type k sets.

Introducing types eliminates the set membership paradox (but reduces the expressiveness of the system), but it does not eliminate all self-referential paradoxes. For example, consider this paradox named for the Cretan philosopher Epimenides who was said to have said “All Cretans are liars.”. If the statement is true, than Epimenides, a Cretan, is not a liar and the statement that all Cretans are liars is false. Another version is the self-referential sentence:

> This statement is false.

If the statement is true, then it is true that the statement is false (a contradiction). If the statement is false, then it is a true statement (also a contradiction).

The type restriction eliminates the paradox regarding set self-inclusion, but it cannot eliminate all self-reference paradoxes.

12.1.2 Gödel's Incompleteness Theorem

Kurt Gödel was born in Brno in 1906, and at the age of 25 published a paper, *On Formally Undecidable Propositions of Principia Mathematica and Related Systems* that established the impossibility of completely mechanizing reasoning. Gödel proved that the axiomatic system in *Principia Mathematica* could not be complete and consistent, but more generally that *no* powerful axiomatic system could be both complete and consistent. He proved that no matter what this axiomatic system is, if it is powerful enough to express certain things, it must also be the case that there exist statements which cannot be proven either true or false in the system. Gödel's proof showed this by construction: to prove that *Principia Mathematica* contains statements which cannot be proven either true or false, it is enough to find one such statement. Gödel's statement is:

> G_{PM}: Statement G_{PM} does not have any proof in the system of *Principia Mathematica*.

If statement G_{PM} is provable in the system, then the system is inconsistent: it can be used to prove a statement that is not true. If G_{PM} is proven, then it means G_{PM} does have a proof, but G_{PM} stated that G_{PM} has no proof. On the other hand, if G_{PM} is not provable in the system, then the system is incomplete. Since G_{PM} cannot be proven in the system, G_{PM} is a true statement. But, the premise is that G_{PM} is not provable. So, we have a true statement that is not provable in the system.

The proof generalizes to *any* axiomatic system, powerful enough to express a corresponding statement *G*:

> G: Statement G does not have any proof in the system.

For the proof to be valid, it is necessary to show that statement *G* can be expressed formally in the system. To express G formally, we need to consider what it means for a statement to not have any proof in the system. A proof of the statement G is a sequence of steps, T_0, T_1, T_2, ..., T_N. Each step is the set of all statements that have been proven true so far. Initially, T_0 is the set of axioms in the system. To be a proof of G, T_N must contain G. To be a valid proof, each step should be producible from the previous step by applying one of the inference rules to statements from the previous step. So, to express statement G, an axiomatic system needs to be powerful enough to express the notion that a valid proof does not exist. Gödel showed that such a statement could be constructed using the *Principia Mathematica* system, and using any system powerful enough to be able to express interesting properties. That is, in order for an axiomatic system to be complete and consistent, it must be so weak that it is not possible to express "this statement has no proof" in the system.

12.2 Computability

Gödel established that no interesting and consistent axiomatic system is capable of proving all true statements in the system. Now we consider the analogous question for computing: are there problems for which no algorithm exists?

Recall the definitions of problem, procedure, and algorithm from Chapter 4. A *problem* is a description of an input and a desired output. A *procedure* is a specification of a series of actions. An *algorithm* is a procedure that is guaranteed to always terminate. A procedure solves a problem if that procedure produces a correct output for every possible input. If that procedure always terminates, it is an algorithm. So, the question can be stated as: is there a problem P such that there exists no procedure that produces the correct output for problem P for all inputs in a finite amount of time.

A problem is *computable* (the term *decidable* is used to mean the same thing) if there exists an algorithm that solves the problem. It is important to remember that in order for an algorithm to be a solution for a problem P, it must always terminate (otherwise it is not an algorithm) and must always produce the correct output for *all* possible inputs to P. If no such algorithm exists, the problem is *uncomputable* (also known as *undecidable*).

12.2.1 The Halting Problem

Alan Turing proved that there exist uncomputable problems. Similarly to Gödel's proof, the way to show that uncomputable problems exist is to find one the way Gödel show unprovable true statements exist by finding an unprovable true statement. The problem Turing found is the *Halting Problem*[1]:

Halting Problem	**Input:**	A specification of a procedure.
	Output:	If evaluating an application of the specified procedure would ever finish, output true. Otherwise, output false.

Suppose we had a procedure `halts?` that solves the Halting Problem. The specification of the procedure could be a quoted Scheme expression. Note that we cannot use the expression unquoted since that would mean the expression is evaluated before `halts?` is applied; if this were done, the `halts?` procedure would never be applied if the input expression would not halt.

The `halts?` procedure should work like this:

```
> (halts? '(lambda () (+ 2 3)))
#t
> (define (factorial n)
        (if (= n 0) 1
             (* n (factorial (- n 1)))))
> (halts?
     '(lambda () (factorial 10)))
#t
> (halts?
     '(lambda () (factorial -1)))
#f
```

The evaluation would not terminate since the initial value of `n` is below the base case, and each recursive application decreases the value of `n`.

```
> (define (fibo n)
```

[1]This problem is a variation on Turing's original problem, which assumed a procedure that takes one input. Of course, Turing did not define the problem using a Scheme expression since Scheme had not yet been invented when Turing proved the Halting Problem was uncomputable in 1936.

```
        (if (or (= n 1) (= n 2)) 1}
            (+ (fibo (- n 1))
               (fibo (- n 2))))))
> (halts? '(lambda () (fibo 60)))
#t
```

Note that it is not possible to implement `halts?` by evaluating the expression and outputting #t if it terminates. The problem is it is not clear when to give up and output #f. For the final example above, the evaluation of `(fibo 60)` would never actually finish because it takes too many steps. However, it would finish eventually in a finite number of steps, so the output of `halts?` should be true. This is not enough to prove that `halts?` is uncomputable. It just shows that one particular way of implementing `halts?` would not work. To show that `halts?` is uncomputable, we need to show that there is no possible way of implementing `halts?` that would produce the correct output for all inputs in a finite amount of time.

One way to prove that no `halts?` procedure could work correctly for all inputs is to find one input for which it could not possible work correctly. Consider this input procedure:

```
(define (paradox)
  (if (halts? 'paradox)
      (loop-forever)
      #t))
```

where the procedure `loop-forever` is defined as:

```
(define (loop-forever) (loop-forever)
```

The body of the `paradox` procedure is an if-expression. The predicate expression is `(halts?  'paradox)`, so if an application of the `paradox` procedure would halt it evaluates to true. If the predicate expression evaluates to true, the consequence expression, `(loop-forever)`, is evaluated. This applies the `loop-forever` procedure, whose body expression is an application of `loop-forever`. Evaluating this never terminates since it is a recursive definition with no base case. Thus, if the predicate evaluates to true, the evaluation of an application of `paradox` never halts. But, this means the result of the `(halts? 'paradox)` predicate was incorrect. Hence, it is not sensible for `(halts? 'paradox)` to evaluate to a true value, since this would cause the application of `paradox` to never terminate.

The other option is that the predicate expression evaluates to a false value. If this is the case, the alternate expression is evaluated. It is the primitive expression `#t` which evaluates to a true value. The evaluation of `paradox` terminates after the if-expression is evaluated. But, this option assumed the predicate `(halts? 'paradox)` evaluated to a false value. This means an evaluation of `paradox` does not terminate. Hence, it is not sensible for `(halts? 'paradox)` to evaluate to a false value, since this would cause the application of `paradox` to terminate.

So, `(halts? 'paradox)` cannot evaluate to a true value, and it cannot evaluate to a false value. There are no other options! The only sensible thing `halts?` could do for this input is to not produce a value. That means there is no way to define an algorithm that solves the Halting Problem. Any procedure we attempt to define to implement `halts?` must sometimes either produce the wrong result or fail to produce a result at all (that is, run forever without producing a result). Thus, the Halting Problem is uncomputable.

There is one missing step in our proof: we argued that because `paradox` does not make sense, something in the definition of `paradox` must not exist, and identified `halts?` as the component that does not exist. This assumes that everything else we used to define `paradox` does exist. This seems reasonable enough — we have been using everything else in it already (an if-expression, a quote expression, applications, and the primitive `#t`) and they seem to exist. But, perhaps the reason `paradox` leads to a contradiction is because `#t` does not really exist. Although we have been using it and it seems to always work fine, we have no proof that evaluating `#t` always terminates. Overcoming this weakness in our proof requires a more formal model of computing. In fact, Turing defined such a model to construct his proof. (Recall this was done in 1936, before anything resembling what we think of as a computer today existed.) We will examine Turing's model later in Chapter **??**.

12.2.2 Computability Proofs

Given a problem description, how do we decide if that problem is computable?

We can show that a problem is computable by describing a procedure and proving that the procedure always terminates and always produces the correct answer. It is enough to provide a convincing argument that such a procedure exists; finding the actual procedure is not necessary (but often helps to make the argument more convincing).

To show that a problem is not computable, we need to show that *no* algorithm exists that solves the problem. Since there are an infinite number of possible procedures, we cannot just list all possible procedures and show why each one does not solve the problem. Instead, we need to construct and argument showing that if there were such an algorithm it would lead to a contradiction.

The core of our argument is based on knowing the Halting Problem is uncomputable. If a solution to some new problem P could be used to solve the Halting Problem, then we know that P is also uncomputable. That is, for a given problem P, no algorithm exists that can solve P since if such an algorithm exists it could be used to also solve the Halting Problem which we already know is impossible.

The proof technique where we show that a solution for some problem P can be used to solve a different problem Q is known as a *reduction*. A problem Q is *reducible* to a problem P if a solution to P could be used to solve Q. This means that problem Q is no harder than problem P, since a solution to problem Q leads directly to a solution to problem P.

Example 12.1: Prints-Three Problem. Consider the problem of determining if an application of a procedure would ever print 3:

Prints-Three	**Input:**	A specification of a procedure.
	Output:	If evaluating an application of the specified procedure would ever print 3, output true. Otherwise, output false.

We show the Prints-Three Problem is uncomputable by showing that it is as hard as the Halting Problem, which we already know is uncomputable.

Suppose we had a procedure `prints-three?` that solves the Prints-Three

Problem. Then, we could define `halts?` as:

```
(define (halts? proc)
   (prints-three?
     '(lambda ()
        (begin
          (apply-procedure proc)
          (print "3")))))
```

where `apply-procedure` is a procedure that takes a procedure specification and applies the specified procedure to no arguments. We need this since the input to `halts?` must be quoted, so using the standard procedure application expression would not work. For now, we will assume such a procedure exists.[2]

The `prints-three?` application will evaluate to `#t` if the application of `proc` would halt, since that means the second expression in the begin expression would be evaluated and that expression prints ”3”. On the other hand, if the application of `proc` would not halt, the second expression in the begin expression would never be evaluated. As long as the procedure never prints ”3”, the application of `prints-three?` should evaluate to `#f`. Hence, the output would correctly solve the Halting Problem.

The one complexity is the input procedure might print ”3” itself. We can avoid this problem by transforming the procedure in a way that it would never print ”3” itself, without otherwise altering its halting or non-halting behavior. One way to do this would be to replace all the places where the `print` procedure appears inside `proc` with a new `do-not-print` procedure that does nothing. So, if we had a procedure `replace-prints` that does this, we could define `halts?` as:

```
(define (halts? proc)
  (prints-three?
    '(lambda ()
       (begin
         (apply-procedure (replace-prints proc))
         (print "3")))))
```

[2]In fact, the built-in procedure `eval` does exactly what we need. It takes a quoted Scheme expression and produces the value that expression would evaluate to. In the following chapter we will see how to define an `eval` procedure.

If there exists a `prints-three?` procedure that correctly solves the Prints-Three Problem — that is, it always terminates and always produces the correct true or false value indicating if the input procedure specification would ever print "3" — then, we could also define a `halts?` procedure that solves the Halting Problem. But, we know that the Halting Problem is uncomputable. Hence, the `prints-three?` procedure we used to define `halts?` cannot exist, and the Prints-Three Problem must also be uncomputable.

The Halting Problem and Prints-Three Problem are uncomputable, but do seem to be obviously important problems. It is useful to know if a procedure application will terminate in a reasonable amount of time, but the Halting Problem does not answer that question. It concerns the question of whether the procedure application will terminate in any finite amount of time, no matter how long it is. Next, we consider a problem that it would be very useful to have a solution for it one existed.

Example 12.2: Is-Virus Problem. A virus is a program that infects other programs. A virus spreads by copying its own code into the code of other programs, so when those programs are executed the virus will execute. In this manner, the virus spreads to infect more and more programs. A typical virus also includes a malicious payload so when it executes in addition to infecting other programs it also performs some damaging (corrupting data files) or annoying (popping up messages) behavior. The Is-Virus Problem is to determine if a procedure specification contains a virus:

Is-Virus	**Input:**	A specification of a procedure.
	Output:	If the procedure contains a virus (a code fragment that will infect other files) output true. Otherwise, output false.

We can demonstrate the Is-Virus Problem is uncomputable using a similar strategy to the one we used for the Prints-Three Problem. We show how to define `halts?` using an `is-virus?` procedure. Since we know `halts?` is uncomputable, this shows there is no `is-virus?` algorithm.

Assume `infect-files` is a procedure that infects files, so the result of evaluating `(is-virus? 'infect-files)` is `#t`. Then, we can define `halts?` as:

```
(define (halts? proc)
   (is-virus?
     '(lambda ()
         (begin
            (apply-procedure
                (remove-infecting-behavior proc))
            (infect-files)))))
```

The `remove-infecting-behavior` procedure is analogous to how we removed possible occurances of printing ”3” using `replace-prints` in the previous example. We need to ensure that the original procedure has no file-infecting behavior that could lead to a true result even when the procedure application does not halt. Defining `remove-infecting-behavior` is a bit trickier than `replace-prints`, and we would have to be careful to define it in a way that does not change the halting behavior of the procedure. But, it could be done. We would need to replace expressions that write to files (that is, possible infecting behaviors) with expressions that do something else.

Virus scanners such as Symantec’s Norton AntiVirus attempt to solve the Is-Virus Problem, but its uncomputability means they are doomed to always fail. Virus scanners detect known viruses by scanning files for strings that match signatures in a database of known viruses. As long as the signature database is frequently updated they may be able to detect currently spreading viruses, but this approach cannot detect a new virus that will not match the signature of a previously known virus. Sophisticated virus scanners employ more advanced techniques than signature scanning to attempt to detect complex viruses such as metamorphic viruses that alter their own code as they propagate to avoid detection. But, because the general Is-Virus Problem is uncomputable, we know that it is impossible to create a program that always terminates and that always correctly identifies an input procedure specification as either a virus or non-virus.

Exercise 12.1. Is the Launches-Missiles Problem described below computable? Provide a convincing argument why it is or why it is not computable.

Launches-Missiles	**Input:**	A specification of a procedure.
	Output:	If an application of the procedure would lead to the missiles being launched, outputs true. Otherwise, outputs false.

You may assume that the only thing that causes the missiles to be launched is an application of the `launch-missiles` procedure. $\diamondsuit$

Exercise 12.2. Is the Same-Result Problem described below computable? Provide a convincing argument why it is or why it is not computable.

Same-Result	**Input:**	Specifications of two procedures, P and Q.
	Output:	If an application of P terminates and produces the same value as applying Q, outputs true. If an application of P does not terminate, and an application of Q also does not terminate, outputs true. Otherwise, outputs false.

$\diamondsuit$

Exercise 12.3. Is the Check-Proof Problem described below computable? Provide a convincing argument why it is or why it is not computable.

Check-Proof	**Input:**	A specification of an axiomatic system, a statement (the theorem), and a proof (a sequence of steps, each identifying the axiom that is applied).
	Output:	Outputs true is the proof is a valid proof of the theorem in the system, or false if it is not a valid proof.

$\diamondsuit$

Exercise 12.4. Is the Find-Finite-Proof Problem described below computable? Provide a convincing argument why it is or why it is not computable.

Find-Finite-Proof	**Input:**	A specification of an axiomatic system, a statement (the theorem), and a maximum number of steps (max-steps).
	Output:	If there is a proof in the axiomatic system of the theorem that uses max-steps or fewer steps, outputs true. Otherwise, outputs false.

$\diamondsuit$

I am rather puzzled why you draw this distinction between proof finders and proof checkers. It seems to me rather unimportant as one can always get a proof finder from a proof checker, and the converse is almost true: the converse failse if for instance one allows the proof finder to go through a proof in the ordinary way, and then, rejecting the steps, to write down the final formula as a 'proof' of itself. One can easily think up suitable restrictions on the idea of proof which will make this converse true and which agree well with our ideas of what a proof should be like.
I am afraid this may be more confusing to you than enlightening. If so I will try again.

Alan Turing, letter to Max Newman, 1940

Exercise 12.5.($\star\star$) Is the Find-Proof Problem described below computable? Provide a convincing argument why it is or why it is not computable.

Find-Proof	**Input:**	A specification of an axiomatic system, and a statement (the theorem).
	Output:	If there is a proof in the axiomatic system of the theorem, outputs true. Otherwise, outputs false.

$\diamondsuit$

12.3 Summary

Although today's computers can do amazing things, many of which could not even be imagined twenty years ago, there are some problems that cannot be solved by computing. The Halting Problem is the most famous example: it is impossible to define a procedure that always terminates and correctly determines if an application of the procedure specified by the input would terminate. Once we know the Halting Problem is uncomputable, we can show that other problems are also uncomputable by illustrating how a solution to the other problem could be used to solve the Halting Problem, which we know to be impossible.

Uncomputable problems often come up in practice. For example, identifying viruses, analyzing program paths, and constructing proofs, are all uncomputable problems. Just because a problem is uncomputable does not mean we cannot produce useful programs that address the problem. These programs provide approximate solutions — they produce the correct results on many inputs, but on some inputs either do not produce any result, or produce an incorrect result. Approximate solutions are often useful in practice, however, even if they are not correct solutions to the problem.