Module COMP11212

Fundamentals of Computation

Computability, Correctness, and Complexity

Gareth Henshall

Semester 2, 2018/19

Part 2

Organisation and Learning Outcomes

Structure, Assessment, Coursework

All of this information is the same as for Part 1 and you should refer to those notes for the details.

Assumptions

This course requires a certain level of mathematical skill; indeed one of the goals is to show that there is a use for much of the discrete mathematics that you are taught elsewhere. Chapter 0 attempts to gather the assumed material together but you should also cross-reference material on discrete mathematics or use the web to look up unfamiliar concepts.

These Notes

This part of the course is divided up into five main chapters. Before they start there is an introduction to the Part which gives a high-level overview of the contents - I strongly suggest reading this.

Important
Read the introduction. I use these boxes to highlight things that are important that you may miss when scanning through the notes quickly. Typically the box will repeat something that has already been said. In this case, read the introduction!

Chapter 0 recaps some mathematical material from COMP11120. Then the main five chapters cover the five main concepts of *models of computation*, *coding and counting programs*, *computability*, *correctness*, and *complexity*. There are examples and exercises throughout the notes. The assessed exercises for the examples classes are outlined in question sheets at the end of the notes. Some examples are marked with a (*) indicating that they go significantly beyond the examined material.

Confused?
Great! Learning theory tells us that we do the best learning when we are confused by something and resolve that confusion. However, this only works if we manage to resolve the confusion. I try to use these boxes in the notes where I think something might be confusing. If there is somewhere where you get very confused and there is no box then let me know and I'll add one.

Reading

In the following I suggest some textbooks that you might find helpful. However these notes should contain everything you need for this course and there is no expectation for you to read any of these books.

Although not examinable, some of you may be interested in the history of computation. There are many books and resources available on this topic. For example, *Computing Before Computers* now available freely online at

`http://ed-thelen.org/comp-hist/CBC.html`

The following two books recommended from the first Part of the course are also relevant here. Hopcroft et al. cover **computability** (undecidability) and **complexity** (tractability) as well as Turing Machines (note that the 2nd Edition goes further into complexity). Parts 2 and 3 of Sipser cover **computability** and **complexity** respectively. Both books treat these topics quite theoretically, using Turing machines as models of computation and going beyond the contents of this course.

> John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000
>
> Michael Sipser. *Introduction to the theory of computation: second edition.* PWS Pub., Boston, 2 edition, 2006

The following book covers complexity in a slightly more pragmatic way (Chapter 11) and was previously the recommended text for this topic.

> Susanna S. Epp. *Discrete Mathematics with Applications.* Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 4th edition, 2010

The following book gives a more algorithmic view of complexity theory and is the recommended reading for the Algorithms course in Year 2.

> Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples.* John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2009

For **correctness** see Chapter 4 of Huth and Ryan or Chapter 15 of Ben-Ari. Both books present the same axiomatic system we explore in these notes, and also cover a lot of things you might want to know about computational logic in general.

> Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press, New York, NY, USA, 2004
>
> Mordechai Ben-Ari. *Mathematical Logic for Computer Science.* Springer Publishing Company, Incorporated, 3rd edition, 2012

The above books also contain a lot of information that you can use to revise fundamental concepts that you should already know.

Those interested more generally in the semantics of programming languages could refer to the following. This material is well outside the scope of this course and these texts are only suggested for interest not for the course.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002

Glynn Winskel. *The formal semantics of programming languages - an introduction.* Foundation of computing series. MIT Press, 1993

Matthew Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, New York, N.Y., 1990

The last of which is out of print and available online at

`https://www.scss.tcd.ie/matthew.hennessy/splexternal2015/resources/sembookWiley.pdf`

Learning Outcomes

The course-level learning outcomes are as follows. Note that each chapter also includes chapter-level learning outcomes which give a more fine-grained overview of the intended learning outcomes of that chapter. At the end of this course you will be able to:

1. Explain how we model computation with the `while` language and the role of *coding* in this modelling
2. Write imperative programs in the `while` language and use the formal semantics of `while` to show how they execute
3. Prove partial and total correctness of `while` programs using an axiomatic system
4. Explain the notion of asymptotic complexity and the related notations
5. Use Big 'O' notation to reason about `while` programs
6. Explain notions of computability and uncomputability and how they relate to the Church-Turing thesis
7. Prove the existence of uncomputable functions
8. Describe why the Halting problem is uncomputable

Aims

In addition to the above learning outcomes I have the following broader aims for this material:

1. Communicate to you that theoretical computer science is not that difficult and has real applications
2. Introduce you to some concepts that I feel are essential for any computer scientist e.g. complexity analysis
3. Have fun

Acknowledgements

Thanks to Giles Reger & Dave Lester for developing the previous version of these notes and to Joseph Razavi for providing feedback on parts of these notes.

Introduction to Part 2

Here I have chosen to introduce all of the concepts you will meet in this course at a high-level in a single (shortish) discussion. Hopefully this will give useful context and provide adequate motivation for the material presented. I suggest you reread this section a few times throughout the course and hopefully by the end of the course everything will be clear.

What is Computation?

This part of the course is about *computation*, but what is that? We are all familiar with *computers* and clearly computers are things that compute. But what does that mean? In Table 1 (see overleaf, sorry I couldn't get it on this page) I give two alternative introductions to this question, one from the practical perspective and one from the theoretical/abstract perspective. I have purposefully written this side-by-side to stress that one perspective is not necessarily more important than the other. Both perspectives conclude that we need a *model of computation* with which to reason about what computers do, or what computation is.

If a computation is a series of instruction executions or symbol manipulations then a static description of a computation is a list of the instructions or operations that we need to perform to achieve that computation. We should be familiar with such descriptions as *programs*. Independent of the model of computation we chose we can abstract computations as programs.

Now that we have decided that we are talking about programs as static descriptions of computations we can ask some interesting questions, such as:

- Are there things we cannot compute i.e. problems for which we cannot write a program?
- What does the program do? Is the program correct?
- Which programs are *better* at solving a given problem?

These are the topics covered by the later parts of the course. But before we can discuss these topics we want a concrete *model of computation* in which to write programs.

A Model of Computation

There exist various models of computation and you have probably heard of some of them. The most famous is perhaps the Turing Machine, introduced by Allan Turing in 1948. This is an abstract machine that operates on an infinite tape reading and writing symbols based on a finite table of instructions. However, Turing machines are not very friendly and, whilst they are important, we will not be concentrating on them in this course.

In this course we will use a very simple programming language, called `while`. This kind of language should be very familiar to you by now, because all of its features are available in other programming

The Practical Perspective	**The Abstract Perspective**
A computer has a processor that executes instructions and stores data in registers. But the processor in my mobile phone is different from the one in my laptop and also different from what we might find in a washing machine or supercomputer. It is clear that all of these processors are doing broadly the same thing; they are computing. But we want to be able to talk about this computation abstractly, without worrying about the exact instructions available or how registers are accessed. To achieve this we want a model that has an abstract notion of stored data and instruction execution and we want this model to be general enough that we can convince ourselves that we could use it to implement any of the concrete computational devices available.	At a high level of abstraction, computation is the process of performing some operations to manipulate some symbols to produce a result. When we add two numbers together in our head the operation is mental and the symbols abstract, when we do it on paper the operations involve a pen and the symbols are lines forming numbers, and when a computer performs the addition it applies logic gates to binary digits. In each case the result is clear. But the computation being performed was not dependent on the operations or symbols involved. There is an abstract notion of what it means to compute which is not related to the mechanisms by which the computation is carried out. To talk about this abstract notion of computation we want a model that captures this idea of performing operations on symbols.

Table 1: Two perspectives on what computation is.

languages such as `java` and `python`. The language is not that important, what is important is that it is simple and able to capture all computable functions, which it does as we find out later. Indeed, in this course we are not interested in how to write specific programs in this language (although you will be doing that) but about methods for analysing programs in this language (for their correctness and complexity) and properties of the language itself (for which we might need to write some programs to establish such properties).

The key feature informing the design of this new language (`while`) is its simplicity when compared to real programming languages. For example there is no inheritance, no data structures (other than the integers ($\mathbb{Z}$), which are not just the 32-bit variety), no functions or methods (indeed no way to structure programs at all), and not even any way to document the code with comments! This considered, this language is completely unusable for real world programming and software engineering.

However, it has some positive points:

- The `while` language is very small: there are five sorts of statements, six sorts of boolean expression, and five sorts of arithmetic expression. Despite this, it is possible to write any program in this language we could instead write in `java` or `python`.
- The above minimality means that we can define the language precisely in a reasonable amount of space and provide the rules needed to prove the correctness of programs in the language. For more complicated languages such definitions quickly become very large and difficult to explain.
- The `while` language *looks* like a proper programming language, and indeed the techniques that will be described for reasoning about the language are applicable to the more complicated programming languages which you will use throughout your career.

The key point to take away from the list above is that as we make the language more usable,

it becomes more complicated, and the simple proofs we will be doing about the language itself will become much more complicated.

One of the reasons we have chosen to use the `while` language instead of Turing machines is that the language is a reasonable abstraction of the programs you will write, meaning that it makes sense to discuss concepts such as correctness and complexity. Whilst Turing machines are the common model of computation used when discussing computability, they are not very useful for presenting these other topics. Importantly, we don't lose anything by choosing the `while` language. Due to the Church-Turing thesis (covered in Chapter 3) these models are *equivalent*. This is important so I will write it in a box.

Important
The `while` language has the same computational power as Turing machines; it is *Turing-complete*.

What Are We Computing?

So we have a language in which to write programs and this language is our model of computation. But a program is just a series of instructions, it tells us nothing about what is being computed. But we do know that a program maps some inputs to outputs. In this sense it computes a *function*[1]. The *domain* of the function is the type of inputs it expects and the co-domain (range) is the type of outputs it produces.

In this course we will mostly restrict our attention to functions on natural numbers ($\mathbb{N}$) i.e. functions in $\mathbb{N} \to \mathbb{N}$. We will do this for two reasons:

1. It is easier. Proofs about computable functions are a lot easier to perform in this domain. Programs that take one output and return one output are easier to write - although in general our programs will compute functions in $\mathbb{N}^n \to \mathbb{N}^m$ i.e. tuples of numbers to tuples of numbers, as that is easier. When trying to grasp complex concepts it is best to keep things as simple as allows for the concept to still make sense.

2. We can *code* any more interesting domain into $\mathbb{N}$. For example, we will see (Section 2.1.2) how to code pairs of numbers as numbers. This argument can be extended to any finite structure. So by only considering functions on natural numbers we do not lose anything.

You might notice that my argument that we can encode anything we are interested in using $\mathbb{N}$ does not allow us to represent functions on $\mathbb{R}$ (the real numbers) as there is no way to represent all numbers in $\mathbb{R}$ using $\mathbb{N}$ (because $\mathbb{R}$ is uncountable; if you cannot remember what uncountable means see Chapter 0). But I am happy with this - show me a computer that can represent $\mathbb{R}$ and I will update my definitions. Practially we can only represent finite approximations of real numbers and we encode any finite structure in $\mathbb{N}$.

We might also talk about programs being solutions to *problems* where a problem might have different solutions. For example, the well-known sorting problem has various well-known solutions.

[1]Some real world programs compute *relations* instead of functions i.e. given a single input multiple outputs are possible i.e. the program is non-deterministic. We do not consider such programs in this course.

So what are these problems and how do they relate to this idea that we compute functions? Talking about a problem is just an informal (lazy) way of talking about a function. The sorting problem is the function that takes a list and produces a list that is sorted. But we will still use the term *problem* as it is nice and friendly.

Confused?

At this point you might be a little confused. We have all these new concepts and how are they related? Is a computation a program, how is a model of computation related to these things? In attempt to clarify these points let us consider an analogy.

We begin by noting that *compute* is an action-word (verb) so it describes something that is active. Let us consider an analogous action of *make a cup of tea* (not a single word, but you get the idea). So if compute is the verb then the noun *computation* describes the action (or set of actions) that was performed i.e. it describes the active thing that happened. So in our analogy a computation is a description of a single tea making process.

A computation computes a *function* i.e. it transforms some start stuff to some end stuff. In the analogy we go from not having a cup of tea to having a cup of tea. The function is abstract and does not tell us anything about how we are supposed to get from the start stuff to the end stuff (i.e. how to make a cup of tea).

A *program* is a set of instructions that we can follow to perform a computation (we could replace program by algorithm but the term program tends to imply a set of well-defined instructions, whereas the notion of algorithm is more abstract). So if we follow our *Cooking for Undergraduates 101* recipe for making a cup of tea then we have used a program.

The difference between a program and a computation is that the program is the static thing and the computation is the active thing. Finally, a *model of computation* is the language in which we describe programs, we need this to make our programs unambiguous and to relate them to functions. We don't have a model of computation in our analogy because we are not that overly concerned about making sure the semantics of tea-making recipes are well-defined.

What can we compute

We now turn to a series of somewhat more abstract questions. Now that we have a model of computation, we can use it to describe the functions that we are able to compute. This is straightforward. We can compute a function if it is possible to write a `while` program to compute it. Note that the above argument that all functions of interest can be mapped to functions in $\mathbb{N} \to \mathbb{N}$ means that this argument applies to all computable functions.

What can we not compute

The above description of what a computable function suggests a very easy answer: we cannot compute functions for which we cannot write a `while` program. However, it is not immediately obvious that this set is nonempty. Here (and later) we show that there exist uncomputable functions by showing that there must be more functions than programs and giving an example of an uncomputable function.

Counting Functions and Programs

Firstly, we need to quickly remind ourselves what *countable* and *uncountable* sets are. A set is countable if there is a bijection between it and $\mathbb{N}$ (if you can't remember what a bijection is then I suggest looking it up at this point, see Chapter 0). A set is uncountable if it is not countable!

Now we argue that the set of programs is countable. If we are lazy we can argue that any program is a finite sequence of symbols and we can code any finite sequence as a natural number. If we are more imaginative then we can show how to map language constructs directly to numbers. In both approaches we take a program and assign a unique number to that program (this is called Gödel numbering). We can use this mapping to give a bijection between programs and the natural numbers, hence programs are countable.

Next we argue that the set of functions $\mathbb{N} \to \mathbb{N}$ is uncountable. We can do this via *Cantor's diagonal argument*. The details are given in Section 3.2 but the general idea is that given any enumeration of functions we can create a function not in that enumeration. Put alternatively, if you try counting functions in $\mathbb{N} \to \mathbb{N}$ we can prove that you will always miss one.

It should be obvious that if one set is countable and the other uncountable then they cannot be the same size, therefore there must be more functions than programs, and therefore functions for which no program exists. However, this proof does not give us an example of an uncomputable function, only shows us that one must exist.

An Uncomputable Function: The Halting Problem

So we have shown that uncomputable functions must exist but do we really believe that without seeing one? We should, because we proved it, but that's not as satisfying as seeing one. Section 3.5 gives a detailed introduction to a famous uncomputable function captured by the Halting Problem. As this is quite involved I don't repeat the description here. The general result is that it is not possible to write a program that, given *any* program as input, decides whether that program halts.

Computational Universality

There are various models of computation, with Turing Machines generally agreed to be the de facto standard definition. Other models of computation are then said to be *Turing-Complete* if they can be used to model Turing Machines. We could use our `while` language to model Turing Machines, thus showing that the language is Turing-Complete. Indeed, there is a broader idea that *any* sensible model of computation can be used to model Turing Machines, and thus all such models of computation compute the same functions, this is called the *Church-Turing thesis*.

At the root of the Turing-Completeness argument is a notion of *universality*. This is the idea that a model of computation is *universal* if it can model itself i.e. can be used to write a program that can execute any program that can be written in that model. If this seems odd then think about C compilers written in C, this is what we mean.

To demonstrate universality we will use a *Universal Program* that takes an encoded program and its input and runs the program on the input. Note that this relies on our previous results that we can encode programs as numbers. We don't even need to write out this Universal Program; we just need to be happy that it exists. For it to exist we just need a computable semantics (we have this) and a method for encoding/decoding programs (we have this).

Is My Program Correct?

Now we have established what our model of computation is (`while` programs) and discussed what they can and cannot compute, we can look at other important properties of these programs. The first of these is a notion of *correctness*.

Important
I am using `java` programs in this motivational introduction as these are programs you should be able to reason about already. Note that the language used in this course, and importantly in exercises, is the `while` language introduced in Chapter 1.

It shouldn't be surprising that we want the programs we write to be correct. But what does that mean? Is the following `java` program correct?

```
1 public int f(int[] array){
2   int len = array.length;
3   int sum = 1
4   for(int i=0;i<=len;i++){
5     sum = sum*array[i];
6   }
7   return sum;
8 }
```

You might immediately spot that it is not syntactically correct, on line 2 we are missing a semi-colon. Those of you used to such errors might also spot the *out-by-one* error in the loop where we use `<=` instead of `<` (this will lead to an `ArrayIndexOutOfBoundsException`. If we fix these two errors is the program correct? If your answer is *yes* then you are wrong. However, if your answer is *no* then you are also wrong because the question doesn't make sense! Sorry about that - in general I will try and avoid asking questions that don't make sense, but sometimes they can be useful to encourage us to reflect on what such questions really mean.

For something to be correct we need to know what it is *supposed to do* and then we can check whether it does this. It might appear that the code is supposed to compute the sum of the values in `array`, as we use a variable called `sum`. Does the program do this? No, it computes the product of the values in `array`. Whether this is correct or not depends on the *specification*. Although using a variable called `sum` when we're computing the product is probably a bad idea.

You should have met the idea of *testing*. Let's assume the program is supposed to compute the product, what is the specification? We can define this using a *precondition* and *postcondition*. A precondition states the inputs that we think that the program is correct for. Let us use the precondition that the length of the array is at least one, to sidestep the discussion of what the product of an empty array should be. This means that if we run the program on an input that does not satisfy the precondition we do not care what it does; in a sense we are presenting a contract e.g. *if you give me something that I want I will give you back something that you want*. The postcondition states what the output should look like, here we can informally write *the product of the values in the input*

array but later we will want to do this more formally. Notice that this pre/postcondition format also specifies the domain and co-domain of the function being implemented by the program.

Now given the above program we can test it by picking input values that make the precondition true and checking that the result is true for the postcondition. Try typing it into your computer and running it with `array = [3,2,1]` does it give the right answer? What other inputs should we try? I could have written the program

```
1  public int f(int[] array){
2    int len = array.length;
3    int sum = 1
4    for(int i=0;i<len-1;i++){
5      sum = sum*array[i];
6    }
7    return sum;
8  }
```

which would still have computed the correct product of `array = [3,2,1]`, so we should also try `array=[1,2,3]`. Are we convinced that the program will compute the correct results on *all* inputs yet? In the extreme case we cannot be convinced without trying them all[2]. What we really want to be able to do is *prove* the program is correct.

This is what we do in Chapter 4. We will introduce a small proof system that allows us to build proofs that `while` programs are correct i.e. that given any state satisfying the program's precondition, the program will transform that state into one satisfying the programs postcondition.

There is one extra complication to this puzzle. What about the following program for computing the product of an array?

```
1  public int f(int[] array){
2    int product=1; boolean isProduct = false;
3    do{
4      product = random.nextInteger();
5      isProduct=true; int check = product;
6      for(int i=0;i<array.length;i++){
7        if(product%array[i] != 0){
8          isProduct=false;
9          break;
10       }
11       check = check / array[i];
12     }
13     isProduct = isProduct && (check==1);
14   }while(!isProduct);
15   return product;
16 }
```

[2]The extreme case means large and complex programs. Note that if we have access to the source code then we can apply methods to ensure that we have covered enough of the program's control and data structures. But these methods are similar in complexity to the ideas we talk about here, and can often only give an approximation of coverage.

This program is looping around guessing some value that could be the product of the values in the array and then checking if the guessed product is correct. With some effort we can convince ourselves that *if* this method returns then it returns the product of the values in `array` but will it eventually return? It is not guaranteed that after some finite amount of time we will correctly guess the right answer, so it is possible that this method will continue trying to guess the product of the values in `array` forever!

In Chapter 4 we separate the question of whether a program is correct with respect to its specification *whenever it terminates* from the question of whether the program terminates. The former notion of correctness is called *partial correctness*, as it only applies to terminating executions of the program, and the latter is called *total correctness*. Note that this second point relates back to the *Halting Problem* described earlier. It is an important observation that the Halting Problem is a general statement about *all* programs and does not preclude being able to prove the termination of some programs.

Which Program is Best?

It should not be surprising that it is possible to write two programs that compute the same function. As a simple case consider `int x =1; int y=2` and `int y=2; int x=1`. Syntactically these programs are different but they compute the same function.

Less abstractly, let's consider the following two methods that find the maximum element in an array. They are both correct i.e. they both find the maximum value. So which program is best?

```
public int max(int[] a){
  int max = a[0];
  int len = a.length;
  for(int i=1;i<len;i++){
    if(a[i]>max){
      max = a[i];
    }
  }
  return max;
}
```

```
public int max(int[] a){
  int len = a.length;
  for(int i=0;i<len;i++){
    boolean isMax=true;
    for(int j=0;j<len;j++){
      if(a[i] <= a[j]){
        isMax=false;
      }
    }
    if(isMax){
      return a[i];
    }
  }
  // unreachable
  return 0;
}
```

Well, the one on the right is longer, does that mean it is best? Or perhaps the one on the left is easier to read so that is best. Okay, it seems that we need to define what qualities we want in a program before we can answer this question. Typically computer scientists and programmers have a keen (sometimes unhealthy) interest in how *fast* something is. So let us rephrase the question, which one is faster?

If you said the one on the left you are right, but do you understand why? Hint: it's not because the one on the right looks more complicated. Let's try running them with different length arrays and see what happens.

The following table tabulates the running times on my laptop for the two different programs on different length arrays of random integers (the contents doesn't matter as we have to look at every element).

Array length	Time in milliseconds	
	left `max`	right `max`
1,000	0	1
10,000	0	73
100,000	0	1,382
1,000,000	0	157,415 (2.5 minutes)
10,000,000	4	?? (got bored)

The left `max` mostly says 0 because it took less than 1 millisecond, and that's the shortest amount of time `java` will accurately let us measure. However, the numbers for the right `max` are pretty big, which is bad, so what's going on. The left `max` method performs n checks where n is the length of the list. It always performs exactly $n - 1$ checks as it needs to look at every element in the array (unless there is a single element). The right `max` method performs n checks when the first element is the maximum already. But if the last element is the maximum it will perform n^2 checks. Try typing $(1,000,000)^2$ into your calculator - that's a big number. This demonstrates that it is easy to write a program that solves the same problem and make it very slow[3].

We call n and n^2 the *worst-case asymptotic complexity* of each method respectively. This discussion has left out a few important points but should give a flavour of what complexity analysis is about. We discuss this further in Chapter 5.

Summary

This introduction has briefly visited all of the core concepts in this Part of the course. You still need to read the rest of the notes but hopefully this introduction serves as a map, connecting the different concepts together and giving a slightly more concise and informal introduction to the ideas.

As with any part of these notes, if anything is unclear or could be improved in general then please let me know.

[3]Note that it is probably not just time complexity that is making the right `max` slow. The left `max` looks at each element of the array once, so it is very friendly for memory cache accesses. However, for large arrays, the right `max` method will need to load the whole array into cache multiple times.

Contents

Chapter 0

Recap

This chapter recaps some of the definitions and concepts you should already know (from COMP11120) that will be used in this course. This material will not be covered in lectures in any great detail and is here to make these notes self-contained. It will not be explicitly examined but its contents will be assumed for the exam, as they are prerequisite for the rest of the course.

0.1 Functions

In Section 2.5 of the COMP11120 notes you met some properties of functions. Notably, *injectivity*, *surjectivity* and *bijectivity*. There the treatment was relatively informal, here I introduce the *same* concepts using formal logic definitions. Remember that we write $\phi : \alpha \to \beta$ as a function with domain α and co-domain β. We can also say that $\alpha \to \beta$ is the *type* of ϕ.

Definition 0.1.1

A function $\phi : \alpha \to \beta$ is *injective* if

$$\forall(x, y : \alpha).\ f(x) = f(y) \Rightarrow x = y$$

A function $\phi : \alpha \to \beta$ is *surjective* if

$$\forall(b : \beta).\ \exists(x : \alpha).\ f(x) = b$$

Definition 0.1.2

A *bijection* $\phi : \alpha \to \beta$ is a function from α to β which is both *injective* and *surjective*.

We also want the notion of an *inverse* of a function.

Definition 0.1.3

We say that for function $f : \alpha \to \beta$ the function $f^{-1}\alpha \to \beta$ is the *inverse* of f if

$$\forall(x : \alpha) : f(f^{-1}(x)) = x$$

You should recall that not all functions have an inverse (e.g. $f(x) = 0$). We can use the notion of inverse function to give an alternative definition of bijection.

Definition 0.1.4

A *bijection* $\phi : \alpha \to \beta$ is a function from α to β which has an inverse.

At this point it would be normal to prove that our two definitions of bijection are equivalent, but as this is a recap I won't do this. The reading list contains books that contain such proofs.

0.2 Countability

We will also depend on some definitions about the cardinality of sets. This was covered in Section 5.2 of COMP11120 but I give a different (but equivalent) definition of these concepts.

Definition 0.2.1

A set S is *countably infinite* if, and only, if there exists a bijection $\phi : S \to \mathbb{N}$.

I assume that you are familiar with the set $\mathbb{N}$ of natural numbers. If not, please remind yourself as it will be important in this part of the course.

Definition 0.2.2

A set S is *countable* if, and only if, it is finite or it is countably infinite.

Definition 0.2.3

A set S is *uncountable* if, and only if, it is *not* countable.

Chapter 1

The `while` Programming Language

As argued in the introduction to this part, in order to discuss computation we need a simple programming language (to provide a model of computation). As we will be discussing the *formal semantics* of this language, and later providing an axiomatic system, we need it to be small, and simple. Unfortunately most real programming languages are far too large and complicated. As we shall see, a larger language can be much easier to use.

Learning Outcomes

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

- Explain why a rigorous unambiguous approach to describing programming languages is necessary;
- Identify syntactically (in)correct `while` programs;
- Write simple programs in `while`;
- Describe how to extend the `while` language with simple features;
- Explain the notions of *state* and *formal semantics* and be able to use these to show how a `while` program executes on a given state

Additionally, you may also understand how the `while` language can be extended with *arrays* but this part of the Chapter is not examinable, and exists only so that we can have a more in-depth discussion in Chapter 5.

1.1 What is a programming language?

Before we define the `while` programming language let us briefly consider what a programming language is. In simple terms, it is a language we use to describe to a computer what it should do. To be unambiguous it is necessary, therefore, that a programming language has a well-defined (formal) *syntax* describing the *form* of valid programs. However, this is not sufficient for communicating what a computer should do, it is also necessary to know what programs written in a programming language *mean*. To achieve these we must describe the language's *semantics*. This can be done at varying levels of formality; in this Chapter we provide a *small-step structural operational semantics* for `while`, which

is really rather formal. Many real programming languages have a well-defined syntax. Fewer have a well-defined semantics. Very few have a *formal* semantics. One of the aims of this chapter is to give you a flavour of programming language design and the associated formal concepts.

1.2 Defining the syntax of `while`

The first thing we will do is define the *grammar* of our tiny programming language (if this looks unfamiliar then revisit Chapter 4 of Part A of this course):

$$\begin{array}{lcl} S & ::= & x := a \mid \texttt{skip} \mid S_1;\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \mid (S) \\ b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \mid (b) \\ a & ::= & v \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \mid (a) \end{array}$$

The variables S, b and a are non-terminals of our grammar and represent (respectively) Statements (`Stm`), Boolean Expressions (`BExp`), and Arithmetic Expressions (`AExp`). You will also notice we have variables in the programming language which we've represented by x, and numerals represented by n. Any of these symbols can be dashed (x') or subscripted (S_1). The variables x can be thought of as strings of alphanumeric characters and numerals n can be thought of as strings of digits. Although that is, perhaps cheating. We could represent the strings matching n as, for example, the following (unambiguous) grammar:

$$\begin{array}{lcl} n & ::= & s \mid -s \mid 0 \\ s & ::= & od \mid o \\ d & ::= & zz \mid z \\ z & ::= & 0 \mid d \\ o & ::= & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

But this is clearly more work than just saying that n represent numerals. The important point to remember is that these are syntactic objects which must be interpreted as numerals, rather than numerals themselves. Syntactically, the only difference between 123, 312 and *cab* is that they are all different sequences of symbols.

Once we have a grammar the first thing we want to do is use it to write a program. Here is our first `while` program:

```
x:=1; y:=5
while (y>0) do (x:=x+x; y:=y-1;)
```

Have a think about what this computes. We will return to it in a few pages and discuss it further. For now just make sure you are comfortable that this program can be generated by the above grammar. Let us note that our grammar is *ambiguous*. This means that when we write

```
x := 17; while (0 ≤ x) do x := x-1;  x := x+1
```

we could mean one of two programs. One choice is:

```
x := 17; while (0 ≤ x) do (x := x-1;  x := x+1)
```

But what we probably meant was instead:

```
x := 17; (while (0 ≤ x) do x := x-1);  x := x+1
```

To disambiguate the expressions and sentences we use the parentheses '(' and ')'. Note that the use of the brackets for statements is not usual in programming languages, but very common in mathematics.

Exercise 1.1

Why is the grammar for `while` *ambiguous*? Give one example of an ambiguous Arithmetic Expression, Boolean Expression, and Statement. You may wish to revisit Section 4.4 of Part A.

1.2.1 Extending the Language

The `while` language is missing a lot of the things we would normally want to write. However, it contains the building blocks necessary to define these things. For example, one might want to write the statement

if b **then** S

i.e. without an `else` branch. But we only have `if` b `then` S_1 `else` S_2 in the `while` language. To be able to capture the required statement we need to write an equivalent *legal* `while` program. In this case we can note that

$$[\![\texttt{if } b \texttt{ then } S]\!] \equiv [\![\texttt{if } b \texttt{ then } S \texttt{ else skip}]\!]$$

This says that the lefthand side (if-then) is equivalent to the righthand side (if-then-else-skip), for all boolean expressions b and statements S.

Whenever we find it helpful, we may mark off parts of the programming language from the other mathematics which we use to describe it; for this purpose we will use *semantic brackets* $[\![S]\!]$.

Exercise 1.2

How can we represent the boolean expression $b_1 \vee b_2$ in the `while` language? (Hint: recall DeMorgan's laws)

Exercise 1.3

How can we represent the boolean expression $b_1 \rightarrow b_2$ in the `while` language?

Exercise 1.4

How can we represent the statement

do S **while** b

in the `while` language? Note that the intended semantics is that S is executed at least once and then while b is true.

In these notes we may use shorthands such as `if` b `then` S where their definition as equivalent programs or expressions in the `while` language would be obvious. Why don't we simply add these to the language? As mentioned previously, we want the language to be small so that it is easier to define and reason about. It is common to define a small set of statements and expressions in which more interesting statements and expressions can be defined.

Important

In the rest of the notes I will generally ignore the restrictions of the grammar and use extensions (particularly of b and a) directly. In general I will use the (obvious) extensions:

- $[\![\texttt{if } b \texttt{ then } S]\!] \equiv [\![\texttt{if } b \texttt{ then } S \texttt{ else skip}]\!]$
- $[\![a_1 < a_2]\!] \equiv [\![\neg(a_2 \leq a_1)]\!]$
- $[\![a_1 > a_2]\!] \equiv [\![\neg(a_1 \leq a_2)]\!]$
- $[\![a_1 \mathrel{!}= a_2]\!] \equiv [\![\neg(a_1 = a_2)]\!]$

Exercise 1.5

Name three features not already mentioned, which are missing from the `while` language. Try to name the most significant omissions, rather than variants of the previous exercises.

1.2.2 Writing Programs in `while`

We have a programming language so let us write some programs. As a first example consider the following program:

```
x:=1; y:=5
while (y>0) do (x:=x+x; y:=y−1;)
```

What does it do? Pause here and think about it. If it is not obvious go to a computer and type

```
x=1; y=5
while y>0:
  x=x+x
  y=y−1
print x,y
```

into `python`[1] and see what happens. Notice how similar our `while` program looks to the `python`. At the end of this program the variable x will contain $2^5 = 32$ and we will say that this is what the program computes.

What if I want to compute 2^n for general n? I need input. We will generally assume that variables can contain values before executing the program and that we can read values out of variables at the end of the program. This is how input and output worked for the first computers.

Modifying the above program is straightforward. Assuming that variable y contains the value n, the following program computes 2^n:

```
x:=1;
while (y>0) do (x:=x+x; y:=y−1;)
```

Now x will contain 2^y at the end of the program.

The following five examples show how to write more complex programs in the `while` programming language. I strongly encourage you to treat these as exercises and attempt to write the corresponding

[1] Just type `python` on the command line to get the interactive mode, at least on any School machine.

programs before looking at the answers. The solutions have been included to expose you to a reasonable number of `while` programs and to give us programs to discuss later in these notes.

Example 1.2.1

Show how to write a division program. You are to accept the numerator and divisor in the variables x and y respectively, and to deliver the result of the division and the remainder in d and r respectively. You may assume that $x, y \in \mathbb{N} \wedge y \neq 0$.

Example 1.2.2

Write an integer square root program, *i.e.* a program which accepts an argument passed in in variable $x \in \mathbb{N}$, and calculates $z = \lfloor \sqrt{(x)} \rfloor$, and $r = x - z^2$, the remainder.

Example 1.2.3

Write a program to compute the nth Fibonacci number. Assume that variable $x \in \mathbb{N}$ contains n and place the result in variable z.

Example 1.2.4

Write a program to compute the *greatest common divisor* (gcd) between two numbers *i.e.* a program which accepts arguments passed in variables $x \in \mathbb{N}$ and $y \in \mathbb{N}$, and places in z the largest number that divides both x and y with no remainder. For further clarification of gcd you may wish to refer to the Wikipedia page.

Example 1.2.5

Write a program to decide whether a number is a prime number. Assume that variable $x \in \mathbb{N}$ contains the input variable and place 1 in z if x is prime and place 0 in z otherwise.

Answer to Example 1.2.1

A division program can be written as follows:

```
r := x;
d := 0;
while y ≤ r do (d := d+1; r := r−y)
```

In simple terms, this program is counting the number of times we can remove y from x. To see why how works consider $x = 10, y = 3$ and write down r and d for each iteration of the loop:

r	d
10	0
7	1
4	2
1	3

and we get the answer $r = 1, d = 3$ as expected.

□

Answer to Example 1.2.2

There are lots of very clever ways to perform this calculation, but we'll choose the very simplest.

```
z := 0;
while ((z+1)*(z+1) ≤ x) do z := z+1;
r := x − z*z
```

In effect we start with an initial guess that the square root is 0 and increment the guess at each step until a number is found that is "too big". If this program is confusing then do what we did before and write out the table of intermediate values.

□

Answer to Example 1.2.3

This is a standard programming task but here we do not have access to many of the language features we may be used to.

```
y:=1;
z:=0;
while(x>0) do (
  t := z;
  z := y;
  y := t+y;
  x := x−1
)
```

Running this on $x = 5$ will lead to the variables holding the following values at the end of the loop:

x	y	z
4	1	1
3	2	1
2	3	2
1	5	3
0	8	5

i.e. we use y to remember the previous value in the sequence.

□

Answer to Example 1.2.4

This can be solved using Euclid's algorithm as follows.

```
while (x != y) do (
  if (x>y) then
    x := x−y
  else
    y := y−x
)
z:=x
```

Note that this program is destructive i.e. it modifies its input. As an exercise you might want to try rewriting it to be nondestructive.

□

Answer to Example 1.2.5

Testing for primality is straightforward - we just check each candidate for a divisor. Normally we would start this process at the square root of x (as no factor can be greater than this) but we have kept the program as simple as possible.

```
y:=x−1;
z:=1;
while (y>1 ∧ z=1) do (
  r:=x;
  (while (y ≤ r) do r:=r−y);
  (if (r=0) then z:=0);
  y:=y−1;
)
```

This program shows us that it would be useful if we could structure our programs so that we could refer to subprograms (note that we perform division here). We will be returning to this program when we discuss *complexity*. Note that this is the only program we have seen so far that contains a *nested* while loop.

□

Important
Remember to answer these exercises using the `while` language as defined in this section. It is allowed to use simple derived statements as I have been doing and you can use complex derived statements as long as you can show how they can be defined in terms of the available constructs.

Exercise 1.6

Write a program to compute the function $f(n) = 3^n$.

Exercise 1.7

Extend the program given in Example 1.2.1 so that it works for general integers e.g. it handles the case where x and y can be negative. Think about the different cases (you may want to look up Euclidean division online).

Exercise 1.8

Extend the program given in Example 1.2.1 so that it works for general integers e.g. it handles the case where x and y can be negative.

Exercise 1.9

Write a program to calculate whether a number is even and another program to calculate whether a number is odd. Now write a program to compute x modulo y.

Exercise 1.10

Write a program to calculate the factorial of non-negative integer in variable x returning the result in variable y.

Exercise 1.11

Write a power program, *i.e.* a program which accepts two arguments passed in in variables x and y, and calculates x^y, placing the value in variable r. You may assume that both $x, y \in \mathbb{N}$. What answer do you propose to give to 0^0? **Hint:** Consider the two sequences $\{0^0,\ 0^1,\ 0^2,\ \ldots\}$ and $\{0^0,\ 1^0,\ 2^0\}$.

Exercise 1.12

Extend the power program from Exercise 1.11 to handle negative x and y.

Exercise 1.13

Write a logarithm-base2 program, *i.e.* a program which accepts an argument passed in variable $x \in \mathbb{N}$, and calculates z and r such that $x = 2^z + r$ where $0 \leq r < 2^z$. Assume that $x \neq 0$. **Hint:** You may find it useful to refer to the division program we saw earlier.

Exercise 1.14

Extend your answer to Exercise 1.13 to take an arbitrary base in variable b.

Exercise 1.15

Write a program to calculate the least common multiple of two positive integers in variables x and y returning the result in variable r.

1.3 Defining the Execution of `while` Programs

The previous section introduced the *syntax* of the `while` programming language and we discussed how to write programs in it but we only have an informal or intuitive sense of what the programs *mean*. To fully describe a programming language we also need to capture the *semantics*.

1.3.1 The Notion of State

The important feature of an imperative programming language, such as `while` is that it has a *state* in which the current values of the variables are stored. There are several alternative ways to think about the state. One is as a table

x	5
y	7
z	0

or as a "list" of the form

$$[x \mapsto 5,\ y \mapsto 7,\ z \mapsto 0]$$

In all cases we must ensure that there is only one value associated with each variable. By defining the state to be a function this requirement is trivially fulfilled, whereas for the alternatives extra conditions have to be enforced. We will use total functions to model this because we want only *one* value for each variable. The total functions for States take a variable to an integer.

Definition 1.3.1

The type State is a total function

$$\text{State} = \text{Var} \to \mathbb{Z}$$

Confused?
What is this State thing? Formally it is a *function space* i.e. a set of functions that have the same domain (here Var) and co-domain (here $\mathbb{Z}$). Informally, we can think of states as bits of memory that are infinitely indexed and hold numbers.

We will assume that any variable not mentioned when we set up a state is 0. This avoids us having to worry about which variables are *defined*.

We will also use the following notation to describe states:

$$[x \mapsto 5,\ y \mapsto 7,\ z \mapsto 0]$$

This is the function which, when applied to x returns 5, when applied to y returns 7, and when applied to anything else returns 0. It is *equivalent* to the state $[x \mapsto 5, y \mapsto 7]$.

Suppose that $\sigma = [x \mapsto 5,\ y \mapsto 7,\ z \mapsto 0]$. Then applying the function σ to variable x gives 5; we write this as:

$$\sigma(x) = 5$$

Confused?
What's this Greek symbol σ doing here? I'm going to use σ to represent states. Previously I used a lowercase s but this clashed somewhat with the uppercase S for statements. In any case, it is not unhealthy to get used to seeing and using Greek letters.

With this notation we can now define a recursive function $\mathcal{A}$ to determine the value represented by an arithmetic expression as seen in Table 1.5. We do something similar with boolean expressions in Table 1.6, except that they return a truth value in $\mathbb{B}$, which is either **tt** or **ff** (true or false, respectively). Don't worry too much about these tables, they capture the obvious semantics of arithmetic and boolean expressions. Understanding the transition system in the next section is more important.

Apart from looking up the values of variables in states, we must also be prepared to change the state when we do an assignment. For example, if we do the assignment $x := 3$ and we currently have the state σ, then the new state σ' will return 3 when s' is applied to x, and otherwise behaves exactly as σ does on every other variable.

Definition 1.3.2

We define the *state modification notation* – using square brackets – as follows. Let $\sigma' = \sigma[x \mapsto v]$; then

$$\sigma'(x) = v$$

And whenever $y \neq x$:

$$\sigma'(y) = \sigma(y)$$

Thus σ' is the same as σ except when it is applied to the variable x, in which case it returns the new value of x (which is v) rather than the old one.

Example 1.3.3

Suppose that the state $\sigma = [x \mapsto 3,\ y \mapsto 6]$. Then every value apart from x and y will be given the value 0.

The modified state $\sigma' = \sigma[x \mapsto 17]$, which we obtain after executing the assignment:

```
x := 17
```

is: $\sigma' = [x \mapsto 17,\ y \mapsto 6]$.

If it helps, what we are doing when we write $\sigma[x \mapsto v]$ is composing two functions i.e. producing a new function $\sigma \circ [x \mapsto v]$.

1.3.2 A Transition System for Statements

We have introduced the notion of a state object as a reasonable abstraction of what is happening in a program at a particular instance of time. Next we describe how programs update the state.

To do this we first introduce the notion of a *configuration* $< S,\ \sigma >$. The first component of $< S,\ \sigma >$ is a `while` program, the second is a state. Table 1.7 introduces a *rewrite system* for

$$
\begin{array}{lcl}
\mathcal{A} : \mathbf{AExp} \to \text{State} \to \mathbb{Z} & & \\
\mathcal{A}\,[\![n]\!]\;\sigma & = & \mathcal{N}\,[\![n]\!] \\
\mathcal{A}\,[\![x]\!]\;\sigma & = & \sigma\;(x) \\
\mathcal{A}\,[\![a_1 + a_2]\!]\;\sigma & = & \mathcal{A}\,[\![a_1]\!]\;\sigma + \mathcal{A}\,[\![a_2]\!]\;\sigma \\
\mathcal{A}\,[\![a_1 \star a_2]\!]\;\sigma & = & \mathcal{A}\,[\![a_1]\!]\;\sigma \star \mathcal{A}\,[\![a_2]\!]\;\sigma \\
\mathcal{A}\,[\![a_1 - a_2]\!]\;\sigma & = & \mathcal{A}\,[\![a_1]\!]\;\sigma - \mathcal{A}\,[\![a_2]\!]\;\sigma
\end{array}
$$

Table 1.5: Defining $\mathcal{A}$ for Arithmetic Expressions

$$
\begin{array}{lcl}
\mathcal{B} : \texttt{BExp} \to \text{State} \to \mathbb{B} & & \\
\mathcal{B}\,[\![\texttt{true}]\!]\;\sigma & = & \mathbf{tt} \\
\mathcal{B}\,[\![\texttt{false}]\!]\;\sigma & = & \mathbf{ff} \\
\mathcal{B}\,[\![a_1 = a_2]\!]\;\sigma & = & \begin{cases} \mathbf{tt} & \text{if} \quad \mathcal{A}\,[\![a_1]\!]\;\sigma = \mathcal{A}\,[\![a_2]\!]\;\sigma \\ \mathbf{ff} & \text{if} \quad \mathcal{A}\,[\![a_1]\!]\;\sigma \neq \mathcal{A}\,[\![a_2]\!]\;\sigma \end{cases} \\
\mathcal{B}\,[\![a_1 \leq a_2]\!]\;\sigma & = & \begin{cases} \mathbf{tt} & \text{if} \quad \mathcal{A}\,[\![a_1]\!]\;\sigma \leq \mathcal{A}\,[\![a_2]\!]\;\sigma \\ \mathbf{ff} & \text{if} \quad \mathcal{A}\,[\![a_1]\!]\;\sigma > \mathcal{A}\,[\![a_2]\!]\;\sigma \end{cases} \\
\mathcal{B}\,[\![\neg\, b]\!]\;\sigma & = & \begin{cases} \mathbf{tt} & \text{if} \quad \mathcal{B}\,[\![b]\!]\;\sigma = \mathbf{ff} \\ \mathbf{ff} & \text{if} \quad \mathcal{B}\,[\![b]\!]\;\sigma = \mathbf{tt} \end{cases} \\
\mathcal{B}\,[\![b_1 \wedge b_2]\!]\;\sigma & = & \begin{cases} \mathbf{tt} & \text{if} \quad \mathcal{B}\,[\![b_1]\!]\;\sigma \;\mathbf{and}\; \mathcal{B}\,[\![b_2]\!]\;\sigma \\ \mathbf{ff} & \text{if} \quad \mathbf{not}\;(\mathcal{B}\,[\![b_1]\!]\;\sigma \;\mathbf{and}\; \mathcal{B}\,[\![b_2]\!]\;\sigma) \end{cases}
\end{array}
$$

Table 1.6: Defining $\mathcal{B}$ for Boolean Expressions

$$
\begin{array}{ll}
[\text{ass}] & < x := a,\ \sigma > \ \Rightarrow\ s[x \mapsto \mathcal{A}\,[\![a]\!]\;\sigma] \\
[\text{skip}] & < \texttt{skip},\ \sigma > \ \Rightarrow\ \sigma \\
[\text{comp}^1] & \dfrac{< S_1,\ \sigma > \ \Rightarrow\ < S_1',\ \sigma' >}{< S_1; S_2,\ \sigma > \ \Rightarrow\ < S_1'; S_2,\ \sigma' >} \\
[\text{comp}^2] & \dfrac{< S_1,\ \sigma > \ \Rightarrow\ \sigma'}{< S_1; S_2,\ \sigma > \ \Rightarrow\ < S_2,\ \sigma' >} \\
[\text{if}^{\text{tt}}] & < \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \sigma > \ \Rightarrow\ < S_1,\ \sigma > \quad \text{if } \mathcal{B}\,[\![b]\!]\;\sigma = \mathbf{tt} \\
[\text{if}^{\text{ff}}] & < \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \sigma > \ \Rightarrow\ < S_2,\ \sigma > \quad \text{if } \mathcal{B}\,[\![b]\!]\;\sigma = \mathbf{ff} \\
[\text{while}] & < \texttt{while}\ b\ \texttt{do}\ S,\ \sigma > \ \Rightarrow \\
 & \qquad < \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ \sigma >
\end{array}
$$

Table 1.7: The **While** Language

rewriting configurations into other configurations. To rewrite a configuration we are executing the first statement within S, and perhaps making changes to the state. The $\Rightarrow$ symbol is not being used as implication in the table, instead it is defining a *relation* between either pairs of configurations, or a configuration and a final state. Of course we read the $\Rightarrow$ symbol as if the program were making progress towards some final state.

These rules can be thought of as a more complicated state transition system than the ones associated with regular expressions. Indeed there is a clue in the naming: for regular expressions we use *finite* state automata. We do not use that name here, because the transition system presented in Table 1.7 has an *infinite* number of possible statements S.

Some transitions are defined so that they only apply when a boolean condition is met (the [if$^{\text{tt}}$] case in the table). Others (*e.g.* [comp^2]) are defined by way of a horizontal line. This line is to be read as: "If the transition above the line is possible, then the transition below the line is also possible." Another way to think about these composition rules is to apply a rule to the first statement in a sequence whilst keeping the rest around e.g.

$$\langle x := 1; y := 2, [] \rangle \Rightarrow \langle y := 2, [x \mapsto 1] \rangle \Rightarrow [x \mapsto 1, y \mapsto 2]$$

The transition system given in Table 1.7 – along with Tables 1.5 and 1.6 – is an example of a *formal semantics* for a programming language. Another (less pretentious) word for semantics is "meaning".

In case you are interested (the remainder of this paragraph is non-examinable), in this particular case we have introduced a *small-step structural operational semantics*. The small-step part comes from the fact that the rules tell us how to carry out each step., the structural part comes from the fact that the structure of the statements determine exactly which rule to apply, and the operational part comes from the fact that the rules are about the operational nature of the program (as opposed to the static nature e.g. well-typedness).

Example 1.3.4

Let us look at an example execution:

$$\begin{array}{lll} < \texttt{if}\ (0 \leq x)\ \texttt{then skip else}\ x := 0 - x,\ [x \mapsto -17] > & & \\ \quad \Rightarrow \quad < x := 0 - x,\ [x \mapsto -17] > & & \text{Rule [if}^{\text{ff}}\text{]} \\ \quad \Rightarrow \quad [x \mapsto 17] & & \text{Rule [ass]} \end{array}$$

What is the effect of this program in general? **Hint:** what can you say about the value of `x` after the execution has finished?

Example 1.3.5

Let us look at another example execution:

$$\begin{array}{lll} & < x := 0; \texttt{while}\ (x > 0)\ \texttt{do}\ (x := x - 1), [] > & \\ \Rightarrow \quad & < \texttt{while}\ (x > 0)\ \texttt{do}\ (x := x - 1), [x \mapsto 0] > & \text{Rule [ass]} \end{array}$$

What just happened? We had something of the form $S_1; S_2$ and now we have S_2 so we must have applied [comp^2]. Here we are cheating and only writing the rule that took S_1 and produced an

updated state, leaving the application of [comp²] implicit. In general, the application of either [comp] rules will be implicit. Now let's continue.

$$\begin{array}{lll} \Rightarrow & < \texttt{if } (x > 0) \texttt{ then } (x := x - 1; & \\ & \quad \texttt{while } (x > 0) \texttt{ do } (x := x - 1)) \texttt{ else skip}, [x \mapsto 0] > & \text{Rule [while]} \\ \Rightarrow & < \texttt{skip}, [x \mapsto 0] > & \text{Rule [if}^{\text{ff}}\text{]} \\ \Rightarrow & [x \mapsto 0] & \text{Rule [skip]} \end{array}$$

This implicit application of [comp] rules could be captured in so-called derived rules such as

$$[\text{ass\_comp}] < x := a; S,\ \sigma > \ \Rightarrow \ < S, s[x \mapsto \mathcal{A}[\![a]\!]\ \sigma] >$$

Rules like this one can be derived from the previous ones and shown to preserve the semantics e.g. all executions produced by the derived rules could have been produced by the original rules.

Example 1.3.6

Now let us take a look at a larger example using the following program for computing 2^n defined earlier:

x:=1;
while (y>0)(x:=x+x; y:=y−1;)

This program requires that n is loaded into y before we begin so let us consider the starting state $[y \mapsto 5]$. We then apply the semantic rules as follows:

$$\begin{array}{lll} & < x := 1; \texttt{while } (y > 0) \texttt{ do } (x := x + x; y := y - 1), [y \mapsto 5] > & \\ \Rightarrow & < \texttt{while } (y > 0) \texttt{ do } (x := x + x; y := y - 1), [x \mapsto 1, y \mapsto 5] > & \text{Rule [ass]} \end{array}$$

At this point we realise that expanding the program is going to involve lots of writing so we start introducing some *definitions*. Let

$$W \triangleq \texttt{while } (y > 0) \texttt{ do } (x := x + x; y := y - 1)$$

in the following[2]. Now let's continue

$$\begin{array}{lll} \Rightarrow & < \texttt{if } (y > 0) \texttt{ then } (x := x + x; & \\ & \quad y := y - 1; W) \texttt{ else skip}, [x \mapsto 1, y \mapsto 5] > & \text{Rule [while]} \\ \Rightarrow & < x := x + x; y := y - 1; W, [x \mapsto 1, y \mapsto 5] > & \text{Rule [if}^{\text{tt}}\text{]} \\ \Rightarrow & < y := y - 1; W, [x \mapsto 2, y \mapsto 5] > & \text{Rule [ass]} \\ \Rightarrow & < W, [x \mapsto 2, y \mapsto 4] > & \text{Rule [ass]} \end{array}$$

At this point we can tell that we are going to repeat ourselves as we have returned to the program

[2] I use $\triangleq$ in these notes when I define something to be equal to something else. Such equalities only exist to introduce names for things.

we have already seen, this will happen when executing loops. Let's finish this.

$$
\begin{array}{lll}
\Rightarrow & < \texttt{if } (y > 0) \texttt{ then } (x := x + x; & \\
& \quad y := y - 1; W) \texttt{ else skip}, [x \mapsto 2, y \mapsto 4] > & \text{Rule [while]} \\
\Rightarrow & < x := x + x; y := y - 1; W, [x \mapsto 2, y \mapsto 4] > & \text{Rule [if}^{\text{tt}}\text{]} \\
\Rightarrow & < y := y - 1; W, [x \mapsto 4, y \mapsto 4 > & \text{Rule [ass]} \\
\Rightarrow & < W, [x \mapsto 4, y \mapsto 3] > & \text{Rule [ass]} \\
\Rightarrow & < \texttt{if } (y > 0) \texttt{ then } (x := x + x; & \\
& \quad y := y - 1; W) \texttt{ else skip}, [x \mapsto 4, y \mapsto 3] > & \text{Rule [while]} \\
\Rightarrow & < x := x + x; y := y - 1; W, [x \mapsto 4, y \mapsto 3] > & \text{Rule [if}^{\text{tt}}\text{]} \\
\Rightarrow & < y := y - 1; W, [x \mapsto 8, y \mapsto 3] > & \text{Rule [ass]} \\
\Rightarrow & < W, [x \mapsto 8, y \mapsto 2] > & \text{Rule [ass]} \\
\Rightarrow & < \texttt{if } (y > 0) \texttt{ then } (x := x + x; & \\
& \quad y := y - 1; W) \texttt{ else skip}, [x \mapsto 8, y \mapsto 2] > & \text{Rule [while]} \\
\Rightarrow & < x := x + x; y := y - 1; W, [x \mapsto 8, y \mapsto 2] > & \text{Rule [if}^{\text{tt}}\text{]} \\
\Rightarrow & < y := y - 1; W, [x \mapsto 16, y \mapsto 2] > & \text{Rule [ass]} \\
\Rightarrow & < W, [x \mapsto 16, y \mapsto 1] > & \text{Rule [ass]} \\
\Rightarrow & < \texttt{if } (y > 0) \texttt{ then } (x := x + x; & \\
& \quad y := y - 1; W) \texttt{ else skip}, [x \mapsto 16, y \mapsto 1] > & \text{Rule [while]} \\
\Rightarrow & < x := x + x; y := y - 1; W, [x \mapsto 16, y \mapsto 1] > & \text{Rule [if}^{\text{tt}}\text{]} \\
\Rightarrow & < y := y - 1; W, [x \mapsto 32, y \mapsto 1] > & \text{Rule [ass]} \\
\Rightarrow & < W, [x \mapsto 32, y \mapsto 0] > & \text{Rule [ass]} \\
\Rightarrow & < \texttt{if } (y > 0) \texttt{ then } (x := x + x; & \\
& \quad y := y - 1; W) \texttt{ else skip}, [x \mapsto 32, y \mapsto 0] > & \text{Rule [while]} \\
\Rightarrow & < \texttt{skip}, [x \mapsto 32, y \mapsto 0] > & \text{Rule [if}^{\text{ff}}\text{]} \\
\Rightarrow & [x \mapsto 32, y \mapsto 0] & \text{Rule [skip]}
\end{array}
$$

Exercise 1.16

What happens when you execute the following program:

while true do skip

Hint: Think about what this program does before expanding the transitions!

Exercise 1.17

Apply the semantic rules to the state $[x \mapsto 8, y \mapsto 3]$ for the division program given in Example 1.2.1. Now try applying them to the state $[x \mapsto 5, y \mapsto -2]$, what happens?

Exercise 1.18

Apply the semantic rules to the state $[x \mapsto 5, y \mapsto -2]$ for the extended division program you wrote in Exercise 1.8.

Exercise 1.19

Apply the semantic rules to the states $[x \mapsto 9]$ and $[x \mapsto 10]$ for the square root program given in Example 1.2.2.

Exercise 1.20

Apply the semantic rules to the state $[x \mapsto 5]$ for the Fibonacci program given in Example 1.2.3.

Exercise 1.21

Apply the semantic rules to the state $[x \mapsto 9, y \mapsto 3]$ for the gcd program given in Example 1.2.4.

Exercise 1.22

Apply the semantic rules to the states $[x \mapsto 3]$ and $[x \mapsto 4]$ for the primality testing program given in Example 1.2.5.

(*) Exercise 1.23

Use your favourite programming language to write an interpreter for the `while` language. The semantic definitions given in this section should be sufficient. If you are feeling very ambitious you could also write a parser. Otherwise, introduce an API that allows you to construct `while` programs programatically to be passed to your interpreter. Anybody who does this and lets me have the code to distribute in future years gets a large bar of chocolate (only available to the first solution in each programming language, modulo race conditions).

(*) Exercise 1.24

Write out all the derived rules we have been using here and prove that they are consistent with the original rules.

1.4 Adding Arrays

Important
Note that the contents of this section is not examinable (see explanation below).

As you'll have noticed, our simple programming language does not have arrays; this is because we do not need them to talk about computability. We can use natural number encodings instead (discussed in the next chapter).

However, I have chosen to include an extension of `while` with arrays, which we will call $\texttt{while}_{\texttt{Arr}}$. Note that this language is not examinable i.e. you will not be asked to recall or apply its semantics or any information relating to it from other Chapters. So why include it? In practice, most programming languages *do* have arrays and when we consider the topic of complexity all of the interesting discussions involve programs including arrays.

1.4.1 Extending the Sytax

The extended syntax for $\texttt{while}_{\texttt{Arr}}$ is as follows:

$$
\begin{array}{lcl}
S & ::= & v := a \mid \texttt{skip} \mid S_1;\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
a & ::= & v \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
v & ::= & x \mid x[a_1, \ldots, a_n]
\end{array}
$$

This adds the notion of *array subscripting*, allowing us to write expressions such as $a[i]$, $b[i, j]$, or $c[i + 1]$ wherever a variable can currently occur in a legal `while` program.

You may immediately notice that this allows us to write programs that might not make sense, for example

```
x[x[x]] = x
```

as an extreme case. To deal with this issue (which we didn't have previously) we can introduce the notion of *types* (see below). We will assume that we use things in their proper place i.e. use variables for arrays only as arrays.

You may also notice that we do not have any way of checking the size of an array to perform bounds-checking. Here we will assume that arrays are *dynamic*, i.e. they expand automatically, although we also assume that indices are positive. In essence this means that arrays are functions on the natural numbers. To facilitate this we will assume that arrays are initialised to zero. These assumptions are captured by the following semantics.

1.4.2 Extending the Semantics

The first thing we need to do is extend the notion of *state* so that it can store information about the contents of arrays. To support arrays of arbitrary dimensions we now (i) allow state objects to point to state objects, and (ii) allow states to be indexed by natural numbers. Therefore, if the state should contain an array x with values 5 and 6 at indices 1 and 3 then the state would be

$$[x \mapsto [1 \mapsto 5, 3 \mapsto 5]]$$

and if we had a two-dimensional array y with values 5 and 6 at [1][2] and [2][1] then the state would be

$$[y \mapsto [1 \mapsto [2 \mapsto 5], 2 \mapsto [1 \mapsto 6]]]$$

This is clearly difficult to read but we don't anticipate writing these down very much. Once we have done this the we can update how we evaluate arithmetic expressions with the following rule:

$$
\mathcal{A}\,[\![x[a_1, \ldots, a_n]]\!]\ s \quad = \quad \begin{cases} s(x)(\mathcal{A}\,[\![a_1]\!]) \ldots (\mathcal{A}\,[\![a_n]\!]) & \textit{if it is defined} \\ 0 & \textit{otherwise} \end{cases}
$$

We say that a value $s(x)(\mathcal{A}\,[\![a_1]\!]) \ldots (\mathcal{A}\,[\![a_n]\!])$ is defined if

$$\mathcal{A}\,[\![a_1]\!] \in \mathsf{dom}\ s(x) \wedge \ldots \wedge \mathcal{A}\,[\![a_n]\!] \in \mathsf{dom}\ s(x)(\mathcal{A}\,[\![a_1]\!]) \ldots (\mathcal{A}\,[\![a_{n-1}]\!])$$

i.e. all the links required are there.

We are also required to update the semantics of assignment to include array assignment. We give the case for two-dimensional arrays only; the extension to one or n-dimension arrays is straightforward. We can do this as follows, where we skip some of the details. In essence it is necessary to extract and update all the relevant sub-states, storing the value of a in the innermost state. Here we abuse notation and assume that retrieving a non-existent state returns the empty state.

$$[\text{ass}] \quad < x[a_1, a_2] := a_3,\ s > \ \Rightarrow \\ s[x \mapsto s(x)[\mathcal{A}[\![a_1]\!] \mapsto s(x)(\mathcal{A}[\![a_1]\!][\mathcal{A}[\![a_2]\!] \mapsto \mathcal{A}[\![a_3]\!]])] >$$

1.4.3 A Note on Types

The operational semantics is a *dynamic* semantics, telling us how the program should execute. Sometimes we also require a *static* semantics restricting the form of valid programs further than the syntactic rules. This introduces the notion of *type* and rules to tell us which programs are properly *typed* (the valid ones). An example of such a rule would be

$$\frac{\Gamma \vdash b : \mathbb{B} \qquad \Gamma \vdash S_1 : \tau \qquad \Gamma \vdash S_2 : \tau}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \tau}$$

which says that if b is a boolean and S_1 and S_2 have the same type τ then the expression $\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$ will have the type τ.

Introducing such rules in general is outside the scope of the course but there are books given in the reading list that cover this topic in depth. The important thing to note is that conformance with the syntactic rules is no longer sufficient to check whether a program is a statically valid $\texttt{while}_{\texttt{Arr}}$ program.

If you want to explore the idea of types a little further then I suggest you go and have a look at *Russell's Paradox* and see how the notion of types was used to avoid it.

1.4.4 Writing some Programs

For these programs we introduce an additional syntactic shortcut (which we could also have used earlier). Here we define a `for` construct in terms of `while`.

$$[\![\texttt{for}\ x = n\ \texttt{to}\ m\ \texttt{do}\ P]\!] \equiv [\![x := n;\ \texttt{while}(x \leq m)\ \texttt{do}\ (P; x := x + 1)]\!]$$

We will use this new construct and the array extension to define some programs. As before, I do this by first presenting them as exercises for you to do and then giving the solutions. In all of the below examples and exercises we assume that arrays are non-empty i.e. their length is > 0. As an additional exercise you could consider what needs to be changed for some programs to make them handle the case where $n = 0$.

Example 1.4.1

> Write a program in $\texttt{while}_{\texttt{Arr}}$ that finds the maximal element in an array. Assume the array is given in variable a and its length is given in variable n and places the result in variable max.

Example 1.4.2

> Write a program in $\texttt{while}_{\texttt{Arr}}$ that performs binary search on a *sorted* array to find the index of a given value. Assume the array is given in variable a, its length is given in variable n, and the value being searched for is given in variable x. Use r to indicate whether the search was successful or not and if successful use z to output the index of the value in x.

Example 1.4.3

Write a program in $\texttt{while}_{\texttt{Arr}}$ that sorts an array in variable a (with its length in variable n) from smallest to largest. The sorting is meant to be destructive and at the end of the program a should be sorted.

Example 1.4.4

Write a program in $\texttt{while}_{\texttt{Arr}}$ that computes the dot product of two arrays a and b and stores it in variable *dot*, both arrays are assumed to have the same length, given in variable n.

Answer to Example 1.4.1

This program simply performs a linear search of the values in a.

```
max=a[0];
for i=1 to n do if max>a[i] then max:=a[i]
```

We initialise *max* with the first value of a, using the above assumption that $n > 0$.

□

Answer to Example 1.4.2

The idea behind this program is that we repeatedly split the array into one half possibly containing x and one half definitely not containing x. The variable *mid* will store our current guess for the location of x and *lo*, *high* define the part of the array with think that x is in. In this program I cheat and assume that our language has a division operator, but the program could be rewritten without it.

```
lo := 0;
hi := n;
mid := n/2;
r:= 0;
while ¬(a[mid] = x) ∧ ¬(lo = hi) do
    if a[mid] ≤ x then lo := mid
    else hi := mid; mid := (hi−lo)/2;
if a[mid] = x then z := a[mid]; r:=1
```

What is the very important assumption we are making about a? As an exercise you could rewrite this program to remove the division operator by using the idea from the previous program to compute division.

□

Answer to Example 1.4.3

I have chosen to implement *bubble sort* as it is one of the shortest to write. If you are not familiar with this algorithm then the idea is to pass through the array $n-2$ times and each time compare each pair of values and swap them if they are in the wrong order.

```
if n>1 then
  for i= 0 to n−2 do
    for j = 0 to n−2 do
      if a[j] > a[j+1] then
        t = a[j]
        a[j] = a[j+1]
        a[j+1] = t
```

Later (in Chapter 5) we will spend some time discussing the efficiency (complexity) of various sorting algorithms. If the array is already sorted this program will still make the same number of iterations over the array. As an additional exercise you could consider how to update the program so that it stops as soon as it knows the array is sorted.

□

Answer to Example 1.4.4

The program for this is nice and simple

```
dot := 0; for i = 1 to n do dot := dot + (a[i] ⋆ b[i])
```

□

Exercise 1.25

Extend the solution in Example 1.4.1 to additionally place the minimum element in z.

Exercise 1.26

The solution in Example 1.4.3 provides one possible sorting algorithm but there are many different sorting algorithms. Write a $\texttt{while}_{\texttt{Arr}}$ program for a different sorting algorithm.

Exercise 1.27

Write a program in $\texttt{while}_{\texttt{Arr}}$ that takes two arrays a and b (with length in variable n) and computes the cross product of the two arrays and stores the result in an array in variable c.

Exercise 1.28

Write a program in $\texttt{while}_{\texttt{Arr}}$ that takes two *sorted* arrays a and b with lengths in variable n and m respectively and *merges* them into an array c such that c is of length $n+m$ and c is also sorted.

Exercise 1.29

Write a program in $\texttt{while}_{\texttt{Arr}}$ that takes two arrays a and b (with length in variable n) and treats them as vectors in an n-dimensional space and then computes the Euclidian distance between them, storing this in z.

Chapter 2

Coding and Counting Programs

The contents of this chapter were previously integrated into the previous and next chapters but I have chosen to separate them into their own chapter to highlight their connection.

There are two ideas in this chapter. Firstly, that we can encode complex finite structures into $\mathbb{N}$ and therefore, we can rewrite any program dealing with those complex structures into one dealing with $\mathbb{N}$ only. This justifies the simplicity of the `while` language. Secondly, that we can encode statements of the `while` language itself into $\mathbb{N}$. This means that we can *enumerate* or *count* all possible `while` programs. We will use this result in the next chapter to establish an important result about (un)computability.

Learning Outcomes

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

- Show how data structures (such as pairs, or syntax trees) can be *coded* as natural numbers; and
- Describe how programs can be encoded as natural numbers and how this result means that the set of programs is countable

2.1 Coding Data Structures

In this section we consider how to encode finite data structures into the natural numbers. But what does this mean? Our general approach will be to define a bijective (two way) function between $\mathbb{N}$ and something else such that we can move between the two presentations. See page 3 for a recap of what a bijection is (it's important in this section). Don't worry if this doesn't make sense yet - there are examples of this below.

As mentioned in various places, the fact that we can do this is important for showing that our results about `while` programs generalise to more complex programming languages.

2.1.1 Integers

The first result we show is that we do not need to consider the integers in general, but can focus directly on natural numbers. What do I mean by this? I mean that any program acting on integers can be transformed into one acting on natural numbers.

To show this let us consider the bijection $\beta : \mathbb{Z} \to \mathbb{N}$ as

$$\beta(x) = \begin{cases} 2x & if\, x \geq 0 \\ -2x - 1 & otherwise \end{cases}$$

For example,

$$\beta(0) = 0 \quad \beta(1) = 2 \quad \beta(-1) = 1 \quad \beta(5) = 10 \quad \beta(-3) = 5$$

i.e. it maps positive numbers to even numbers and negative to odd.

We can compute β i.e. using the following program which takes the input in x and puts the output in z.

```
if x >=0 then z := 2 * x else z := (-2 * x) -1
```

Importantly, we can also compute its inverse β^{-1} (on inputs in $\mathbb{N}$) using the following program

```
r := x; z:= 0;
while(2 <= r) do (z := z+1; r = r-2);
if r=1 then z := -z -1
```

This leads us to the following result

Theorem 2.1.1

The function

$$\beta : \mathbb{Z} \to \mathbb{N}$$
$$\beta(x) = \begin{cases} 2x & if\, x \geq 0 \\ -2x - 1 & otherwise \end{cases}$$

is bijective.

Proof

As β has an inverse β^{-1} (demonstrated by the above program) it is bijective. I leave it as an exercise (e.g. an application of the methods introduced in Chapter 4) to prove that the above program correctly implements β^{-1}.

□

2.1.2 Pairs

Now we will introduce a function that can be used to encode and decode pairs of natural numbers as a single natural number i.e. a *bijective* function in $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$. The function we introduce is

$$\phi(n, m) = 2^n(2m + 1) - 1$$

which takes any pair of natural numbers and produces a *unique* natural number. Before we prove that this function is bijective let's consider what it is doing. The following table gives its value for some different values of n and m.

		m			
		0	1	2	...
	0	0	2	4	...
n	1	1	5	9	...
	2	3	11	19	...
	3	...	...	...	

If we consider what these numbers would look like written in binary we will see that this produces m in binary, followed by 0 and then n 1s. For example, $\phi(2,3) = 27$, which is 11011 in binary i.e. 11 followed by 011. This example should already make us feel quite confident that ϕ is a bijection as we have, in our heads, an algorithm that could take a binary number and get n and m back.

Important
At this point let's consider why we picked the above function. It is definitely not the only bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$. For example, we could simply have picked $2^m(2n+1)-1$ to get a different function. One point to consider is why we need the final -1. To understand what this is here for consider $\phi(0,0)$ and $\phi^{-1}(0)$. Without the -1 we would not have any pair of numbers mapped to 0.

Theorem 2.1.2

The function

$$\begin{aligned} &\phi : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N} \\ &\phi(n,m) \quad = \quad 2^n(2m+1) - 1 \end{aligned}$$

is bijective.

We will show this in two stages.

Lemma 2.1.3

ϕ *is an* injection*, i.e.*

$$\forall (n, m, n', m' \in \mathbb{N}) : \ \phi(n,m) = \phi(n',m') \to (n = n' \wedge m = m')$$

Proof of 2.1.3

Suppose that $2^n(2m+1)-1 = \phi(n,m) = \phi(n',m') = 2^{n'}(2m'+1)-1$. Eliminating the -1 from each side, we can assume that:

$$2^n(2m+1) = 2^{n'}(2m'+1)$$

Suppose that $n \neq n'$ and assume that $n < n'$ (without loss of generality). Dividing both sides by 2^n we get:

$$2m+1 = 2^{n'-n}(2m'+1) \tag{2.1}$$

Now the lefthand side is odd, but the righthand side is even, so this is not possible, and we have a contradiction. It doesn't matter whether we take $n < n'$ or $n' < n$ we will run into this problem. Thus *by contradiction*, we have shown that $n = n'$. If this is so, we divide both sides of Equation (2.1) by 2^n and we get

$$2m + 1 = 2m' + 1$$

And thus $m = m'$. Hence ϕ is injective.

□

Lemma 2.1.4

ϕ *is a* surjection *i.e.*. $\forall(y : \mathbb{N}) : \ \exists(n, m : \mathbb{N}) : \ \phi(n, m) = y$.

Proof of 2.1.4

For a given natural number $y \in \mathbb{N}$ we show how to construct n and m such that $\phi(n, m) = y$. This is a relatively straightforward reversal of the function. We begin by noting that we assume $y = 2^n(2m + 1) - 1$ so the number $y + 1$ is of the form $a \times b$ where a is an even number and b is an odd number. We can find a by viewing $y + 1$ in binary and taking the longest chain of least-significant bits that are 1. Alternatively, we can find b by dividing $y + 1$ by 2 until it becomes odd. In either case we can directly get n from a as $n = \mathsf{log}_2(a)$ (or the number of bits in a), and m from b as $m = \frac{b-1}{2}$. As we see below, this is directly the method for computing the inverse of ϕ.

□

Now we return to the theorem we were proving: Theorem 2.1.2.

Proof of Theorem 2.1.2

By Lemma 2.1.3, ϕ is injective; by Lemma 2.1.4, ϕ is surjective. Thus ϕ is bijective.

□

Of course, we could also have taken the alternative approach of finding an inverse ϕ^{-1} by giving a program that computes it. This can be achieved by the following program where the loop is computing b from the above proof of surjectivity. This program takes y as input and returns values in n and m.

```
n := 0; m := y +1;
while (even(m) ∧ m > 0) do (m:= m/2; n := n+1);
m := (m−1)/2
```

This follows our previous discussion about how the result of ϕ can be viewed as a binary number.

Exercise 2.1

Why is the program given above for computing ϕ^{-1} not a legal `while` program? Give a corrected version, *i.e.* write an equivalent legal `while` program.

What this has shown is that there is an inverse function $\phi^{-1} : \mathbb{N} \to (\mathbb{N} \times \mathbb{N})$. Indeed with a few minor changes you will have written a program to calculate this function in Exercise 2.1.

Let's just recap this, shall we? We have shown how to define a bijection between a pair of natural numbers and the natural numbers.

Example 2.1.5

What is the result of $\phi(3,\ 15)$?

$$\begin{aligned} \phi(3,\ 15) &= 2^3(2 \times 15 + 1) - 1 \\ &= 8 \times 31 - 1 \\ &= 247 \end{aligned}$$

Example 2.1.6

What is the pair represented (or coded) by the number 127? This is asking 'what values of n and m make $\phi(n,\ m) = 127$?'

$$\begin{aligned} 127 &= 2^n(2 \times m + 1) - 1 \\ 128 &= 2^n(2 \times m + 1) \end{aligned}$$

Thus $n = 7$ and $m = 0$ (because $2^7 = 128$).

Exercise 2.2

What is the result of

- $\phi(0, 0)$?
- $\phi(0, 1)$?
- $\phi(5, 0)$?
- $\phi(0, 5)$?

Which is the pair presented by the number

- 0?
- 1024?
- 126?

Exercise 2.3

Define a bijective function between a triple of natural numbers and the natural numbers. Prove that it is bijective.

2.1.3 Lists

We now briefly consider how to encode lists as natural numbers. Almost all structures of interest can be decomposed into pairs and lists, so once we are done we will have tools that should allow us to encode almost any structure we would like to.

As a recap, a list is a possibly empty finite-sequence of values. We can define lists recursively using the grammar

$$list = [] \mid n :: list$$

where $n \in \mathbb{N}$, [] is the empty list and :: is the *cons* operator appending a value to the front of a list. If this presentation of lists (and the following recursive definition) is unfamiliar to you then you should recap (or review) section 6.1 of the COMP11120 notes.

We can then define a coding of lists φ recursively as follows

$$\begin{array}{lcl} \varphi([]) & = & 0 \\ \varphi(n :: l) & = & 2^n(2\varphi(l) + 1) = \phi(n, \varphi(l)) + 1 \end{array}$$

i.e. we encode a list as a pair of the head of the list and the rest of the list. However, notice that we do not use the previous pairing function directly as this possibly gives the value 0 and we reserve 0 for the empty list. Instead we omit the final -1 to shift the coding up by 1.

Exercise 2.4

Write a `while` program to compute φ.

Exercise 2.5

Prove that φ is bijective by giving a `while` program to compute the inverse function φ^{-1}.

(*) Exercise 2.6

Use your favourite programming language to write the coding functions for pairs and lists. Try and write it generally so it allows you to encode pairs or pairs, lists of pairs, lists of pairs of lists of pairs etc.

Exercise 2.7

Explain why we can encode a list of natural numbers using the following scheme:

$$\begin{array}{l} \phi_V : \mathbb{N}^k \to \mathbb{N} \\ \phi_V([n_1,\ n_2,\ \ldots,\ n_k]) = 2^{n_1+1}3^{n_2+1} \cdots {p_k}^{n_k+1} \end{array}$$

where p_k is the k-th prime number. Explain why this is an injection but is not a surjection.

Exercise 2.8

Provide a coding function that codes a binary tree as a natural number. Recall that a binary tree can be defined recursively as

$$btree = empty \mid (n, btree, btree)$$

You might find it useful to use the coding function from Exercise 2.3. Briefly argue that your function is a bijection.

2.1.4 Operations on Structures as Arithmetic

As we are able to encode pairs and lists as natural numbers we can argue that we do not need these structures in our language to be able to represent programs that use these structures. This is why we only consider numbers in our language.

The rest of the argument that we don't need these structures is that we can encode any operations on such structures as arithmetic. For example, the two main functions on pairs are first and second, retrieving the first and second values respectively. The two main functions on lists are head and tail returning the first element of a list and the rest of the list respectively. These functions related directly to part of the inverse coding functions discussed above. Other functions, such as taking the length of a list could either act directly on the encoded version, or (more straightforwardly) be implemented by applying the decoding operations.

As we see later, the transition relation of `while` can itself be coded as operations on natural numbers following a similar approach.

2.1.5 What Does This Mean?

By showing that arbitrary data structures can be coded as natural numbers we convince ourselves that we only need to consider programs on natural numbers. I've repeated myself here as **this is an important point**. If we only prove the results of the next chapter for these `while` programs (which look very different from 'real' programs) then the results are not very interesting. However, if we can also argue that any other program could be translated into a `while` program (which is what this section has been about) then the results can be lifted to more interesting programs and become themselves more interesting.

2.1.6 Removing Arrays

We previously extended `while` with arrays (in a non-examinable section). As with pairs and lists we can code arrays as natural numbers. We do not give the details here as they are tedious and not important. We just need to be happy that we can do it. One approach would be to represent an array as a list of pairs (i, a) where i is an index into the array and a is the value at that index. There can only be a finite number of places in an array where its value is non-zero, so the resulting list is finite. Once we have this representation we can use the previous coding functions for pairs and lists. Furthermore, we can extend the coding arbitrarily to nested arrays, arrays of pairs etc.

2.2 Counting Programs

We now show that we can *count* or *enumerate* `while` programs by introducing a bijection between statements of the `while` language and $\mathbb{N}$. This will become important later for defining (i) the notion of a *universal program*, and (ii) proving a result about the existence of uncomputable functions.

We introduce coding functions for the different kinds of 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.

Recall that the data type of Arithmetic Expressions (**AExp**) 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 ϕ_A and is as follows:

$$\begin{array}{lcccl}
\phi_A : \mathbf{AExp} \to \mathbb{N} & & & & \\
\phi_A(n) & = & & & 5 \times n \\
\phi_A(x) & = & 1 & + & 5 \times x \\
\phi_A(a_1 + a_2) & = & 2 & + & 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \\
\phi_A(a_1 - a_2) & = & 3 & + & 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \\
\phi_A(a_1 \times a_2) & = & 4 & + & 5 \times \phi(\phi_A(a_1), \phi_A(a_2))
\end{array}$$

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* (ϕ, see Section 2.1.2), as well as the recursively defined encoding function for **AExp** (ϕ_A).

Exercise 2.9

Introduce coding functions to handle the first two items above.

The existence of the bijection ϕ_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.

Example 2.2.1

Assuming that numerals are encoded as themselves and x, y, z are encoded as $0, 1, 2$, we get the following codings for the first 7 natural numbers.

- $\phi_A(0) = 0$
- $\phi_A(x) = 1$
- $\phi_A(0 + 0) = 2 + 5 \times \phi(0, 0) = 2$
- $\phi_A(0 - 0) = 3$
- $\phi_A(0 \times 0) = 4$
- $\phi_A(1) = 5$
- $\phi_A(y) = 6$
- $\phi_A(x + 1) = 2 + 5 \times \phi(\phi_A(x), \phi_A(1)) = 2 + 5 \times \phi(1, 0) = 7$

And as one might expect, the code number for even simple statements becomes very large:

$$\begin{array}{rcl} \phi_A(1+2) & = & 2+5\times\phi(\phi_A(1),\phi_A(2)) \\ & = & 2+5\times\phi(5,10) \\ & = & 2+5\times(32\times(20+1)-1) \\ & = & 2+5\times 671 \\ & = & 3362 \end{array}$$

We can also encode boolean expressions as natural numbers. Recall that the data type of Boolean Expressions (**BExp**) is defined by six cases:

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

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.

$$\begin{array}{lcl} \phi_B : \mathbf{BExp} \to \mathbb{N} & & \\ \phi_B(\texttt{true}) & = & 0 \\ \phi_B(\texttt{false}) & = & 1 \\ \phi_B(a_1 = a_2) & = & 2 + 4\times\phi(\phi_A(a_1),\phi_A(a_2)) \\ \phi_B(a_1 \leq a_2) & = & 3 + 4\times\phi(\phi_A(a_1),\phi_A(a_2)) \\ \phi_B(\neg b) & = & 4 + 4\times\phi_B(b) \\ \phi_B(b_1 \wedge b_2) & = & 5 + 4\times\phi(\phi_B(b_1),\phi_B(b_2)) \end{array}$$

Exercise 2.10

Firstly, use ϕ, ϕ_A and ϕ_B, complete the following recursive function ϕ_S which is a bijection from **Stm** to $\mathbb{N}$:

$$\begin{array}{lcl} \phi_S : \mathbf{Stm} \to \mathbb{N} & & \\ \phi_S(\texttt{skip}) & = & 0 \\ \phi_S(\texttt{while } b \texttt{ do } S) & = & 1 + 4\times\phi(\phi_B(b),\phi_S(S)) \\ \phi_S(x := a) & = & ? \\ \phi_S(S_1;\ S_2) & = & ? \\ \phi_S(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2) & = & ? \end{array}$$

Secondly, argue that this means there are only countably many programs in `while`.

Example 2.2.2

Based on the expected answer to Exercise 2.10 we can write down the code number for various programs:

- The code number for `skip` is 0
- The code number for `skip`; `skip` is 3
- The code number for `while true do skip` is 1
- The code number for `while false do skip` is 5
- The code number for `while true do skip`; `skip` is 25

This notion of coding a program into $\mathbb{N}$ is called different things in the literature. We will normally refer to the above coding of a `while` program as the program's *code number* but below we also introduce some alternative terminology.

Definition 2.2.3

Given a `while` program S, let $\gamma(S)$ be its *index*, *code number* or *Gödel number* given by some bijection from `while` programs to $\mathbb{N}$. Note that the type of γ is $\texttt{Stmt} \to \mathbb{N}$.

The term *index* comes from the fact that it is the index into the enumeration of `while` programs and the use of γ is a reminder that Gödel's key insight lies in his using codings into the natural numbers in his seminal work. Henceforth (in this chapter and the next) we will fix $\gamma(S) = \phi_S(S)$ as our bijection.

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

Important
I repeat this important point. The bijection γ means that we have (i) a function to turn `while` programs into natural numbers, and (ii) a function to turn natural numbers into `while` programs i.e. γ^{-1}.

Exercise 2.11

After answering Exercise 2.10 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 $\phi_A(x)$?
  - `skip`; `skip`; `skip`
  - `if true then skip else skip`
- What are the programs with code numbers 2 and 18?

Exercise 2.12

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

We finish this section with the following result.

Theorem 2.2.4

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 2.10. To show the *effectiveness* 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 ϕ_S and appeal to the Church-Turing Thesis (introduced later in Section 3.6) which tells us that all programming languages are equivalent. In a previous iteration of the course a HASKELL program was provided to achieve this second point and can be obtained by contacting me.

□

(*) Exercise 2.13

Write your own encoding of ϕ_S in your favourite programming language. You will need to represent `while` programs internally and handle the previous caveats about ϕ_A.

2.3 A Program to Compute Programs

We now have a method for coding `while` programs as natural numbers. This means that we can now code the configurations introduced in Section 1.3.2 into $\mathbb{N}$ as follows. Recall that a configuration $\langle S, \sigma \rangle$ consists of a `while` statement S and a state σ. First let us consider states to be lists of pairs of variables and numerals and we can code states as follows

$$\begin{array}{lcl} \phi_\sigma([]) & = & 0 \\ \phi_\sigma((x,v) :: l) & = & 2^{\varphi(x,v)}(2\varphi(l)+1) \end{array}$$

(again assuming that variables and numerals are coded into $\mathbb{N}$ in a sensible way) and now configurations can be simply coded as

$$\kappa(\langle S, \sigma \rangle) = \phi(\phi_S(S), \phi_\sigma(\sigma))$$

Finally, we can write `while` programs to compute the transitions of Table 1.7 as functions on coded configurations. As this is rather complicated and the details are not interesting I do not actually do this here. However, the result is interesting. By doing this we produce a very special program called the *Universal Program*. We can think of it as an interpreter for `while` written *in* `while`.

We will revisit this program later and ask ourselves whether it is really computable. As a hint, if you can solve the following exercise then the answer is *yes*.

(*) Exercise 2.14

Use your favourite programming language to write a universal program for the `while` language.

Chapter 3

Computability

So far we have seen that functions can be computed by programs but we have not considered whether this is the case for *all* functions. This is the question of which functions *can* be computed and what *cannot*. We already informally discussed this idea in the introduction to this part. Now we address it formally.

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: many tasks we would like to undertake *cannot* be performed by computers[1]! 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?

[1] Although often we can find some approximation or heuristic that gives us a good enough answer, or a method that solves a specific case but not the general case etc. So whilst we are often theoretically doomed, there are common cases where things aren't that bad!

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.

Important
The above points are just philosophical points to act as an introduction to this topic. In this chapter we consider the question of computability more formally and it is this formal discussion that is examinable. To reiterate, the points above cannot be used to argue for the existence of uncomputable functions.

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*
- 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[2]. You are not supposed to be able to answer these exercises until *after* you have read the rest of this chapter.

Exercise 3.1

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 3.2

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

[2]In contrast to other chapters which were often about *applying*.

Exercise 3.3

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?

3.1 Computable Functions

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

3.1.1 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 2.1) 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.

Definition 3.1.1

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 σ, and $n \in \mathbb{N}$ with $\sigma(\mathtt{x}) = n$, then if there is some state σ' such that

$$< S,\ \sigma > \Rightarrow^* \sigma'$$

then

$$\sigma'(\mathtt{x}) = f(n)$$

where $\Rightarrow^*$ is the *transitive closure* of $\Rightarrow$ (the transition relation for `while`).

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 3.6.1 – 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 and there is no such state σ' such that $< S,\ \sigma > \Rightarrow^* \sigma'$.

Example 3.1.2

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 3.1.1, we take $S = \texttt{skip}$, and then after one step $\sigma = \sigma'$, and thus the value of `x` is unchanged.

Example 3.1.3

The function $f(n) = n+1$ is *computable*. We can invoke Definition 3.1.1 with the `while` program S as $x := x + 1$ because

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

Exercise 3.4

Show that the following functions are computable by finding a program S that computes them:

- $f(n) = 5$
- $f(n) = n^2$
- $f(n) = n!$
- $f(n) = \begin{cases} 1 & \textit{if } n > 0 \\ 0 & \textit{otherwise} \end{cases}$
- $f(n) = \frac{10}{n}$ (what should happen for $n = 0$?)

Exercise 3.5

Suppose f and g are computable functions. By writing a `while` program prove that $f \circ g$ is also computable.

Hint: If f and g are computable, then there must be programs S_f and S_g satisfying Definition 3.1.1.

Exercise 3.6

Suppose that f is a computable function. By writing a `while` program show that for any k the function f^k is computable.

Exercise 3.7

Suppose that f is a computable function. By writing a `while` program show that the (partial) function

$$g(n) = \begin{cases} 1 & \textit{if } f(n) \textit{ is even} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

is computable.

3.1.2 Computable Functions $\mathbb{N}^m \to \mathbb{N}^n$

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 $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$. 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 ϕ. Here we assume the pairing bijection introduced in Section 2.1.2, but any bijection $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$ can be used.

Definition 3.1.4

We say that a function $f : (\mathbb{N}, \mathbb{N}) \to \mathbb{N}$ is *computable*, if, and only if, the function $g : \mathbb{N} \to \mathbb{N}$ is computable using Definition 3.1.1, where

$$f(x, y) = g(\phi(x, y))$$

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

Definition 3.1.5

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 3.1.1, where

$$f(x) = \phi^{-1}(g(x))$$

Generalizing the two Definitions 3.1.4 and 3.1.5, we can define computability on $\mathbb{N}^m \to \mathbb{N}^n$ using iterated pairing and unpairing bijections.

Definition 3.1.6

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 3.1.1, such that

$$\begin{aligned} g(\phi_x(x_1,\ \phi_x(x_2,\ \ldots \phi_x(x_{n-1},\ x_n)))) = \\ (\phi_x(y_1,\ \phi_x(y_2,\ \ \ldots\ \phi_x(y_{m-1},\ y_m)\ \ \ldots\ )) \end{aligned}$$

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 3.8

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 $n, m \geq 1$.

3.2 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 3.2.1

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 3.2.1

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 is 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.

	0	1	2	3	4	...
f_0	$\boxed{f_0(0)}$	$f_0(1)$	$f_0(2)$	$f_0(3)$	...	
f_1	$f_1(0)$	$\boxed{f_1(1)}$	$f_1(2)$	$f_1(3)$	...	
f_2	$f_2(0)$	$f_2(1)$	$\boxed{f_2(2)}$	$f_2(3)$	...	
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$\boxed{f_3(3)}$	...	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		

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_{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_{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 $f_{new} : \mathbb{N} \to \mathbb{N}$ be the function:

$$f_{new}(n) = f_n(n) + 1$$

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

Suppose that $f_{new} = f_k$ then by extensionality:

$$\forall(n \in \mathbb{N}).\ f_{new}(n) = f_k(n)$$

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

To recap: we have shown that

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

Therefore we have *uncountably* many functions $\mathbb{N} \to \mathbb{N}$.

□

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_{new}.

Exercise 3.9

Give another function which is not enumerated, and that is different to f_{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 `while` program. We can state this formally as the following Corollary:

Corollary 3.2.2

There are non-computable or uncomputable functions of type $\mathbb{N} \to \mathbb{N}$.

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 3.2.2 is that there are non-computable functions of any type that can be encoded in $\mathbb{N} \to \mathbb{N}$.

Exercise 3.10

Give proof sketches for the following statements:

1. There are an uncountable number of functions of type $\mathbb{N} \to \mathbb{B}$

2. There are a countable number of functions of type $\mathbb{B} \to \mathbb{N}$

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 3.11

Explain why there must be functions of type $(\mathbb{N} \times \mathbb{N}) \to$ `Bool` which cannot be written in `java`.

3.3 Decidability

That we can partition the set of functions of type $\mathbb{N} \to \mathbb{N}$ 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. In fact, there is a special class of functions you should already be familiar with: boolean functions or predicates e.g. those with a co-domain of $\mathbb{B}$.

Definition 3.3.1

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 3.1.1, where

$$P(x) = \begin{cases} \textbf{True} & \text{if } f(x) = 1 \\ \textbf{False} & \text{if } f(x) = 0 \end{cases}$$

Functions which return boolean values are called *predicates*.

The functions that are computable using the above Definition 3.3.1 are given a special name: they are called *decidable*. In short, a decidable predicate is a computable one.

Definition 3.3.2

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 `while` is a *decision procedure* for P.

Any predicate which is *not* decidable is *undecidable*.

Lemma 3.3.3

If P and Q are decidable predicates, then all of the following are also decidable:

- $\neg P$;
- $P \wedge Q$;

- $P \vee Q$; *and*
- $P \rightarrow Q$.

Proof

We only consider the $\neg P$ case and leave the rest to be proved in Exercise 3.12. 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 3.5.

□

Before we go any further, it is worth reiterating that the associated function f for the predicate P in Definition 3.3.2 is called the *Characteristic Function*.

Definition 3.3.4

The *Characteristic Function* χ_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 3.12

By writing the associated programs, or otherwise, prove that each of the predicates defined in Lemma 3.3.3 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 3.3.5

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

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

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

If P_i are mutually exclusive 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. If P_i are not mutually exclusive then we can write a set of predicates $Q_1, \ldots, Q_n$ that capture the ordering e.g. $Q_i(x) \Leftrightarrow P_i(x) \wedge \neg P_{i-1} \wedge \ldots \wedge \neg P_1(x)$.

□

Lemma 3.3.6

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 divisor 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; if r=0 then 1 else 0
```

□

Exercise 3.13

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

Hint: You may find your answer to Exercise 1.9 useful here.

Exercise 3.14

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

Exercise 3.15

Prove the decidability of the predicate isNFib : $(\mathbb{N} \times \mathbb{N}) \to \mathbb{B}$ that takes numbers i and n and returns whether n is the ith 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 3.3.7

The partial function P is *partially decidable* if, and only if, there exists a computable partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ 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 progam in `while` is a *partial decision procedure* for P.

Recall that the usual way to represent 'undefined' with a program is to fail to terminate. The above description is a sufficient but not necessary one. There may exist a program for a partially-decidable predicate P that returns 1 whenever $P(x)$ is true and 0 some of the time when $P(x)$ is false and fails to terminate otherwise.

Exercise 3.16

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

- $\neg P$;
- $P \wedge Q$;
- $P \vee Q$; and
- $P \Rightarrow Q$.

Exercise 3.17

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 3.5) is partially decidable.
- Determining whether a context-free grammar is *universal*, e.g. generates all strings, or is *ambigious*, 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

3.4 Universality of Computation

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

3.4.1 The Functions that we can Count

Previously (see page 34) we introduced the function γ as a bijection between `while` programs and $\mathbb{N}$. We are also interested in the unary function associated with the program whose index or code number is i. We will use the notation η_i for this. Recall that we can use coding arguments to lift all of these arguments to non-unary functions.

Definition 3.4.1

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

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

It is important to understand what this is doing. We explicitly define the function η_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 η_i as fixed for each i, *i.e.* η_i is the unary function associated with the `while` program whose index is i.

Lemma 3.4.2

The function $h : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ 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

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

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 γ, the function that this program computes is not unique.

The importance of Lemma 3.4.2 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 **??** as $0 \neq 3$ but $\gamma^{-1}(0) =$ `skip`, $\gamma^{-1}(3) =$ `skip;skip`, and both programs behave in exactly the same way on all inputs.

Exercise 3.18

How many different programs are there for the same function?

3.4.2 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 35. Let us begin by defining the *universal function* ψ_U for unary functions[3].

Definition 3.4.3

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

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

where function η_a is the function computed by the program with code number a.

Fairly clearly, this single function ψ_U captures the behaviour of every unary computable function, because by selecting a value of a, we obtain the unary computable function η_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 ψ_U to simulate applying that function to b.

Exercise 3.19

Show that by choosing a correctly, ψ_U can implement the following functions.

- The identity function $f(x) = x$.
  Hint: Show that η_0 is the identity function.
- 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 3.4.4

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 3.4.5

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

[3]The subscript U is for *universal* not for *unary*

Sketch Proof of 3.4.5

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 1.7 using natural numbers as described on page 35.
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.

□

This is the reason we gave the complete formal description of the `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.

3.5 The Halting Problem

Section 3.2 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:

"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.

3.5.1 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 *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*[4] that the 'halt-tester' program, written in `while` is the program HALT, and that this takes the value $\gamma(p)$[5] (p's code number) and the program's input n as inputs (as a pair in variable x) and outputs either 0

[4]The assumption that there is a 'halt-tester' program will lead to a contradiction.

[5]Some discussions we will often refer to a program and its code number interchangeably i.e. whenever they talk about passing a program to another program they implicitly mean that program's code number. I try and avoid this here by explicit use of the coding function γ.

or 1 in variable x, representing false and true respectively. In other words we have assumed that the predicate halts($\gamma(p),\ n$) 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 ($S = \texttt{skip}$) with input 1 we run the program:

```
x := 2; HALT
```

The assignment of x to 2 is because $\phi(\gamma(\texttt{skip}),\ 1) = 2$.

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; y := 1; while 1 ≤ z do (y := y⋆2; z := z−1);
x := (2⋆x+1)⋆y−1;
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 halts(p, n) is decidable, then so is the predicate self(p).

An alternative way to define the self predicate:

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

By Theorem 3.3.5, 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:

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

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:

```
SELF; if x = 1 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 3.5.1, we see that

$$\text{weird}(\text{weird}) = \begin{cases} \textbf{undefined} & \text{if self}(\text{weird}) \\ \textbf{True} & \text{if } \neg\text{self}(\text{weird}) \end{cases} \tag{3.1}$$

But

$$\text{self}(\text{weird}) = \text{halt}(\text{weird},\ \text{weird})$$

There are now two cases:

halt(weird, weird) is true In this case we take the first branch of Equation 3.1, 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.

halt(weird, weird) is false In this case we take the second branch of Equation 3.1, 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.

3.5.2 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:

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

(the syntax of arithmetic and boolean expressions remaining unchanged). Because of the extra statement, we need to change the Gödel Numbering bijection from γ to $\gamma_{\mathcal{H}}$, which we define as:

$$\begin{array}{lcl}
\gamma_{\mathcal{H}} : \textbf{Stm'} \to \mathbb{N} & & \\
\gamma_{\mathcal{H}}(\mathcal{H}) & = & 0 \\
\gamma_{\mathcal{H}}(\texttt{skip}) & = & 1 \\
\gamma_{\mathcal{H}}(\texttt{while}\ b\ \texttt{do}\ S) & = & 2 + 4 \times \phi(\phi_B(b), \gamma_{\mathcal{H}}(S)) \\
\gamma_{\mathcal{H}}(x := a) & = & 3 + 4 \times \phi(x, \phi_A(a)) \\
\gamma_{\mathcal{H}}(S_1;\ S_2) & = & 4 + 4 \times \phi(\gamma_{\mathcal{H}}(S_1), \gamma_{\mathcal{H}}(S_2)) \\
\gamma_{\mathcal{H}}(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2) & = & 5 + 4 \times \phi(\phi_B(b), \phi(\gamma_{\mathcal{H}}(S_1), \gamma_{\mathcal{H}}(S_2)))
\end{array}$$

If the variables p and n are initialized with the Gödel Number of program $P \in$ **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 contradition as we did before. Let the predicate $W(p)$ be defined as:

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

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; H; if x = 0 then x := 1 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.

$H(w, w) = 0$ In this case $W(w)$ is **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.

Important
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.

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 $\not{\mathcal{H}}$, then we generate a contradiction.
- Because of the contradiction, we know that the function H cannot be computable (it has no corresponding program in `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 `while` language with a halting construct then we cannot consistently extend any realistic programming language with one.

Exercise 3.20

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

Exercise 3.21

Show that the problem of determining that a program halts is partially-decidable.

3.5.3 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 4 we will meet the idea of proving *total correctness*, which will give us 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.

3.6 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** λ-Calculus, 1936
- **Gödel-Kleene** μ-recursive functions, 1936
- **Turing** Turing Machines, 1936
- **Post** string-rewriting, 1943
- **Markov** 1954
- **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 α. 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 `while`. To show how we could encode the behavior of `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 online.

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

Thesis 3.6.1 (Church-Turing)

> *Any sensible definition of computation will define the same functions to be computable as any other definition.*

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.

Chapter 4

Proving `while` Programs Correct

In this chapter we will show that it is possible to *prove* that a `while` program is correct. To do this we must *specify* what the program is supposed to do. The technique we will use involves specifying the state before and after executing a program and proving that whenever we start in an acceptable state we end in one. We will explore what this means by way of examples.

It is worth briefly considering what this means and why it is important. What this means is that we do not just *test* a program over a sample of possible inputs and outputs, put instead check mathematically that it will *always* deliver the right answers for all possible inputs. We will expand this point later.

The typical use-cases justifying the use of program correctness are safety critical systems and security systems. If you take a look at some of the following websites:

`https://en.wikipedia.org/wiki/List_of_software_bugs`
`http://www.cse.psu.edu/~gxt29/bug/softwarebug.html`
`https://www5.in.tum.de/~huckle/bugse.html`

you will see numerous examples of software bugs that led to the loss of millions of dollars and even the loss of life. Many of these bugs were due to seemingly simple mistakes i.e. out by one errors[1]. Arguably these mistakes could not have been made if the techniques introduced in this chapter had been used.

Learning Outcomes

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

- Write simple specifications in the form of pre and post conditions
- Describe the meaning and role of *loop invariants* and *loop variants*
- Describe the difference between *partial* and *total* correctness
- Apply axiomatic rules to establish the *partial correctness* of `while` programs
- Apply axiomatic rules to establish the *total correctness* of `while` programs

[1] Some are clearly more complex, for example *race conditions* in concurrent software cannot be dealt with using techniques in this chapter, but there are formal methods that can be applied to detect such bugs.

4.1 What does Correctness Mean?

We have alluded to this both above and in the introduction. **It does not make sense to talk about correctness without knowing what the intended outcome is.** There are a number of things we could mean by correctness.

Syntactic Wellformedness. It is clear that `if while x` is not a valid program in `while` as it is not accepted by the grammar of the language. In this sense it is incorrect. Other kinds of syntactic errors you have probably met are missing semicolons or misspelled keywords. These errors are typically easy to detect as one just needs to check the program against the grammar.

Properly Typed. Another (usually) static property of a program is whether it type-checks. For example, in `java` we cannot write `String x = 5;` as `5` is not of type `String`. Type-checking is usually designed to be easy to check, although some languages have more complex type systems. Note that `while` does not have types as they are not needed to discuss computation.

Well-Behaved With Respect to Language Properties. A key example of deviation from this kind of correctness is an `ArrayOutOfBoundsException` in `java`. The `java` language tells us that we should not access arrays outside of their defined bounds and doing so is an error. Another example is `NullPointerException`. In C and similar languages we have a host of more complicated errors involving *memory-safety*. There are many tools that can be used to check these properties.

Semantic Correctness. The program `x:=5` is syntactically correct, assumedly properly typed and doesn't seem to break any sensible language rules. But if I meant to type `x:=6` then the program is not correct. This is the first time we meet an *explicit* notion of intended outcome. In the above cases the intended outcome was *implicit* in the programming language. Now we need to make it explicit by writing it down. This is the kind of correctness we return to shortly.

Termination. Sometimes we want our programs to finish running. Sometimes we don't. A program to calculate the 1,000,000th prime number is supposed to terminate, this might be after a long time, but it is supposed to give an answer. A web-server is supposed to run continuously and react to requests. So the idea of whether a program should or should not terminate is also part of a specification. We don't cover the idea in this course, but often for programs that don't terminate we want to establish other properties, such as *progress* and *fairness*.

In all of the above cases there is a clear notion of what the intention was, whether it was implicit or explicit. Here is a quick reminder.

Important
A well-formed program is not inherently correct or incorrect, it is only correct or incorrect with respect to a given specification. A program P can be correct for a specification A but incorrect for a specification B.

So how do we capture a reasonable notion of semantic correctness for `while` programs? Recall that programs are functions on states i.e. there is a start state and an end state. The *actual* behaviours of a program P can be collected as the set

$$\mathsf{Actual}(P) \equiv \{\langle \sigma_{start}, \sigma_{end} \rangle \mid \langle P, \sigma_{start} \rangle \Rightarrow \sigma_{end}\}$$

The *intended* or *desired* behaviours can be captured similarly by using two predicates: Pre for a *pre*condition on the start state and Post for a *post*condition on the end state. The intended meaning is that if the start state satisfies Pre then the end state should satisfy Post. The assumption is we only care about what happens to programs starting in states satisfying the precondition. The set of behaviours for this specification is then

$$\mathsf{Specification}(\mathsf{Pre}, \mathsf{Post}) \equiv \{\langle \sigma_{start}, \sigma_{end} \rangle \mid \mathsf{Pre}(\sigma_{start}) \rightarrow \mathsf{Post}(\sigma_{end})\}.$$

The problem of checking the correctness of P with respect to Pre and Post is then (theoretically) the problem of checking whether

$$\mathsf{Specification}(\mathsf{Pre}, \mathsf{Post}) \subseteq \mathsf{Actual}(P).$$

However, both of the above sets are normally infinite (at least in theory) so checking this inclusion by testing is not a reasonable approach. This chapter introduces a set of rules that allow us to carry out the check.

Important
The intention here is to provide an intuition for what comes later. In exercises you will be expected to use the presented axiomatic system rather than make these kinds of informal arguments.

This is not the full picture as we have only considered *terminating* runs of a program i.e. those that have an end state. Non-terminating runs will not be captured in either of the above sets. This is such an important point that we separate the two cases into two kinds of correctness. If we do not consider non-terminating runs then this is *partial* correctness (partial because we only consider some of the runs). Otherwise, it is necessary to ensure that all runs terminate, this is called *total* correctness. Often we want to establish total correctness but sometimes this is not possible (see Section 3.5) or desired (recall the web-server example).

Important
The problem of checking whether a program satisfies its specification **if** it terminates is called *partial* correctness. The problem of checking whether a program satisfies its specification **and** it always terminates is called *total correctness*.

4.2 Proving the Division Program is Correct

In this section we will show that the division program from Example 1.2.1 (page 9) is correct. As a reminder this program can be written as follows:

```
r := x;
d := 0;
while y ≤ r do (d := d+1; r := r−y)
```

Important
Note that the purpose of this section is to give you an intuition of how we aim to prove these things. The next sections describe how I actually want you to do it i.e. the approach you should follow in exercises.

4.2.1 Writing the Specification

Before we can continue with the proof we need to establish the specification. You may recall that in this exercise we were only concerned with the case where $x, y \in \mathbb{N} \wedge y \neq 0$. To generalise to the integers (e.g. $\mathbb{Z}$) we can state this as $x \geq 0 \wedge y > 0$. This is our precondition. Our postcondition is that the result of the division is in d and the remainder in r. We could write this as $(y \times d) + r = x$, but this is not specific enough as this could just set $d = 0 \wedge r = x$ and be true. Furthermore, r should be positive, otherwise we could add 1 to d and subtract y from r and continue to get further valid solutions. Therefore, our postcondition is $x = (y \times d) + r \wedge 0 \leq r < y$ (what would happen if our program changed x and y?).

Exercise 4.1

Write pre and post conditions for the extended division program given in Exercise 1.8 (page 12).

Exercise 4.2

Write pre and post conditions for the logarithm base 2 program given in Exercise 1.13 (page 12).

4.2.2 Proving Partial Correctness

The first thing we are going to tackle is the `while`-loop and then build the rest of the proof around this. This is a generally good strategy as proving the correctness of `while`-loops is generally the difficult bit. Dealing with assignments and conditionals is relatively straightforward as it is clear how these update the state. The problem with `while`-loops is that we cannot tell *statically*[2]. how many times they will execute, so there is no concrete relation between states before and states after the loop.

To deal with this issue we will find an expression that is always true of states before and after the `while`-loop. If we could do this then we can replace the `while`-loop by this expression and everything would become nice again i.e. the case we know how to solve. Such an expression is called a *loop-invariant*.

[2]This is a term that means 'without running it' as opposed to *dynamically*. Static program analysis is the field of (formally) analysing source code.

Definition 4.2.1

A *loop-invariant* for the statement

while b **do** S

is a predicate P which is true of the state at the start and end of each execution of a `while`-loop body S.

Confused?
A common mistake is to think that P must hold *during* the loop body S. It only needs to hold before and after S executes.

It is important to note that a loop-invariant does not mean that the values of variables do not change. It is assumed that they do change, but the expression should be true for every combination of values that they can take when the loop is executed. Hopefully this will become clearer soon.

There are many possible loop-invariants for our program. For example,

- $d \geq 0$
- $r \leq x$
- $z = 0$

But to be *useful* it must involve all of the variables that change. Remember that implicitly we want to replace the `while`-loop by this expression and if some variables are not mentioned then their effect on the rest of the program will not be captured.

Confused?
In case what I am saying causes confusion. We will not actually replace the `while`-loop by a loop invariant. It is just a way of thinking about the role of loop invariants.

In the case of the division program, we see that variable d increases by 1 every time the loop is executed, whilst variable r has y subtracted each time. The value of y does not change. Indeed, only d and r are changed by the loop. One clue that we need to multiply d by y is that one variable increases by 1 and the other decreases by y. So, if we add r to $d \times y$ we have a value that should be constant, let's call it K:

$$K = d \times y + r$$

How can we check that it is constant? Let's look at what K will be on the next iteration of the loop i.e. when d is 1 larger and r is y smaller:

$$(d+1) \times y + (r - y) = d \times y + y + r - y = d \times y + r = K$$

So what is the value of K? We know that initially r is set to the value of variable x and d is 0. Substituting these into the above gives us

$$K = d \times y + r = 0 \times y + x = x$$

So a possible loop invariant is

$$x = d \times y + r$$

which should not be surprising. However, to be strong enough to establish our postcondition we also need to know that r remains non-negative throughout the loop, so that it is non-negative at the end. Note that this is dependent on x being non-negative at the beginning, which is part of our precondition. This gives a loop invariant of

$$x = d \times y + r \wedge r \geq 0$$

which can easily be shown to hold on every iteration of the loop.

The rest of the proof is quite straightforward. It is easy to see that given our precondition and the assignments that the loop invariant will hold at the beginning of the loop (this is by construction). It only remains to consider what happens when the loop terminates.

The `while`-loop terminates when $\neg(y \leq r)$, *i.e.* $r < y$. Thus, when the program finishes the following condition is true

$$d \times y + r = x \wedge 0 \leq r < y$$

and as this is the postcondition we wanted to hold we have established that the program is (partially) correct.

4.2.3 Proving Total Correctness

To complete the proof that our division program is correct, we must show that each while-loop will always terminate. To do this we must find a quantity that strictly decreases every time we execute the loop and is bounded below i.e. eventually stops decreasing.

One simple scheme that ensures this is to provide an expression v which is a natural number quantity (thus $v \geq 0$) and where each time we execute the loop body this quantity always becomes smaller. Thus whatever value v has when we start, we know that the maximum number of times we can go around the loop is v (why?), and thus the loop must terminate. This quantity v is known as the *loop variant*.

Definition 4.2.2

A *loop-variant* for the statement

while b **do** S

is a quantity v which strictly decreases on every iteration of the `while`-loop and is bounded below by b i.e. when it reaches a certain value the `while`-loop will terminate.

Once more let us consider our division algorithm. There is one obvious quantity that looks suitable and that is r. Each time we execute the loop body this quantity appears to be reduced by y. In fact r is only being reduced if the quantity y is positive; if $y = 0$, we are calculating $x \div 0$, and the program will not terminate, which is arguably mathematically reasonable, although extremely poor practical

programming! The important feature is that the loop will terminate when r gets *small enough*, so by making sure r is strictly decreasing we know that we will eventually stop looping.

Given our precondition $x \geq 0 \wedge y > 0$ we can show that the division algorithm is *totally correct* using a combination of our partial correctness result and the above argument that it is always terminating.

Important
Note that without the precondition $y > 0$ the program is not totally correct but is still partially correct as it still computes division correctly whenever it terminates but there are cases where it will not terminate.

4.3 An Axiomatic System for Partial Correctness

We can be more systematic about proving programs are correct, by using a system to ensure we do not miss out important features or forget to prove something important. Do not be misled: in the same way that double-entry book-keeping keeps account records straight despite still needing arithmetic; following an axiomatic schema will not magically eliminate the need for mathematics. Instead, it gives us a structured way in which to present our reasoning about programs.

Pre- and Post-Conditions and Hoare Triples

Recall that we use predicates on states as pre and post conditions. We say that a state s satisfies a condition P if $P(s)$ holds. That is: P is a predicate (or boolean function) on states. We can give P a type: State $\to \mathbb{B}$. Formally, such a predicate should look something like

$$P(\sigma) = \sigma(x) > \sigma(y)$$

but we will *always* write these as

$$x > y$$

where the state is implicit. We will be making use of logical symbols such as conjunction $\wedge$, disjunction $\vee$, and implication $\to$ and I will assume that you understand what they mean.

Example 4.3.1

The following are examples of predicates on states and states that are true and false in them.

Predicate	True state	False state
$x > y$	$[x \mapsto 2, y \mapsto 1]$	$[x \mapsto 1, y \mapsto 1]$
$x > 0$	$[x \mapsto 2, y \mapsto 1]$	$[y \mapsto 3]$
$2^a > x$	$[a \mapsto 1, x \mapsto 1]$	$[a \mapsto 0, x \mapsto 1]$
$(x + y) \geq 0 \wedge z = 0$	$[x \mapsto 0, y \mapsto 0, z \mapsto 0]$	$[z \mapsto 1]$

We will also talk about implications between predicates, e.g. $P \to P'$ for predicates P and P', where such implications mean that for all states σ we have $P(\sigma) \to P'(\sigma)$. Again, we will abuse notation and write such impliciations as, for example, $x > 0 \to x \geq 0$.

We will specify the behaviour of a program S in terms of the conditions that are true before and after the execution of the program.

Definition 4.3.2

A *Hoare Triple* for partial correctness is a triple consisting of two predicates, and a statement, which we will write as:

$$\{P\}\ S\ \{Q\}$$

We will refer to P as the *pre-condition*, and Q as the *post-condition* for the statement[3] S.

The Hoare triple $\{\ P\ \}\ S\ \{\ Q\ \}$ says that if we start in a state satisfying P and execute S then we will end in a state satisfying Q, or we will not terminate (as this is partial correctness).

Example 4.3.3

Here are some examples of Hoare triples that are valid:

- $\{\ \mathit{true}\ \}\ x := 0\ \{\ x = 0\ \}$
- $\{\ x = 0\ \}\ x := x + 1\ \{\ x = 1\ \}$
- $\{\ P\ \}$ `if` $x > y$ `then` $z := x$ `else` $z := y\ \{\ P \wedge z = y\ \}$ where P is $(x = 1 \wedge y = 2)$. This is an example of how we sometimes introduce names to make things more readable.
- $\{\ x = 0 \wedge y = 1\ \}\ t := x; x := y; y := t\ \{\ x = 1 \wedge y = 0\ \}$
- $\{\ \mathit{false}\ \}\ S\ \{\ Q\ \}$ for any program S and postcondition Q
- $\{\ P\ \}$ `while` *true* `do` $S\ \{\ Q\ \}$ for any program S, precondition P and postcondition Q

Here are some examples of Hoare triples that are not:

- $\{\ x = 1\ \}\ x := 0\ \{\ x = 1\ \}$
- $\{\ x = 1\ \}$ `skip` $\{\ x = 0\ \}$
- $\{\ x = 1\ \}$ `if` $x > 1$ `then` $x := x - 1\ \{\ x = 0\ \}$

To check the validity of such statements we introduce a set of rules that allow us to break down the program S and check each statement individually. We introduce these rules next. But first let us spend a little more time considering how we write specifications as Hoare triples

Example 4.3.4 (Specifying Division)

We can now write the pre and postconditions we generated in the previous section for Example 1.2.1 as the following Hoare triple

$$\{\ x \geq 0 \wedge y > 0\ \}\ S\ \{\ z = (x * d) + r \wedge 0 \leq r < y\ \}$$

Note that this is a specification and any program S satisfying it meets our requirements. The specification is separate from the proof of a particular program.

[3] You will have noticed that I am using the term statement and program interchangeably.

Example 4.3.5 (Initial Values)

Imagine we wanted to specify a program that swapped the values of variables x and y. How should we specify this? We might be tempted to write[4]

$$\{ \quad \} \ S \ \{ \ x = y \wedge y = x \ \}$$

but this does not mean what we want. The program $x := 1; y := 1$ satisfies this specification. Notice that we need to allow for the program updating the values of x and y. To do this we introduce *auxiliary variables* to record their values at the beginning of the program. The solution using auxiliary variables is the following Hoare triple

$$\{ \ x = a \wedge y = b \ \} \ S \ \{ \ x = b \wedge y = a \ \}$$

where a and b record the initial values of x and y respectively. This is a general issue with specification and if we are being very strict we should always record the initial values of variables if we want to refer to them again later. This observation might make us revisit our previous specification of division and update it to be more defensive, but we will keep this previous specification as it is easier to work with.

Exercise 4.3

Write Hoare triples capturing the pre and post conditions for the problems described in the following previous Examples and Exercises. Make sure that you capture all implicit requirements (e.g. a restriction to $\mathbb{N}$) and think carefully about whether you need to make use of auxiliary variables.

- The square root problem in Example 1.2.2 (page 9)
- The primes problem in Example 1.2.5 (page 9)
- The extended division problem in Exercise 1.8 (page 12)
- The factorial problem in Exercise 1.10 (page 12)
- The power problem in Exercise 1.11 (page 12)
- The logarithm problem in Exercise 1.13 (page 12)

If you introduce any additional mathematical functions (e.g. max) then they should be defined.

Inference Rules

We introduce a proof system that allows us to establish the validity of Hoare triples. This is *sound* (i.e. it does not allow us to prove false things valid) but due to the undecidable[5] nature of the problem the system is necessarily *incomplete* (i.e. there are true things we cannot prove are valid).

The inference rules for the proof system are given in Table 4.8. Two of the rules have no premise and the reset are of the form

$$\frac{\textit{premise}}{\textit{conclusion}}$$

[4] I will sometimes use the empty precondition instead of *true* as it represents the case where we assume nothing.

[5] Recall what undecidability means from Section 3.3. Here the undecidability stems from the use of arithmetic.

$$\text{ass}_p \quad \{\ P[x \mapsto \mathcal{A}[\![a]\!]]\ \}\ x := a\ \{\ P\ \}$$

$$\text{skip}_p \quad \{\ P\ \}\ \texttt{skip}\ \{\ P\ \}$$

$$\text{comp}_p \quad \frac{\{\ P\ \}\ S_1\ \{\ Q\ \},\ \{\ Q\ \}\ S_2\ \{\ R\ \}}{\{\ P\ \}\ S_1;\ S_2\ \{\ R\ \}}$$

$$\text{if}_p \quad \frac{\{\ P \wedge \mathcal{B}[\![b]\!]\ \}\ S_1\ \{\ Q\ \},\ \{\ P \wedge \neg\mathcal{B}[\![b]\!]\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\ Q\ \}}$$

$$\text{while}_p \quad \frac{\{\ P \wedge \mathcal{B}[\![b]\!]\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\ P \wedge \neg\mathcal{B}[\![b]\!]\ \}}$$

$$\text{cons}_p \quad \frac{\{\ P'\ \}\ S\ \{\ Q'\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}} \quad \text{if } P \to P' \text{ and } Q' \to Q$$

Table 4.8: Inference System for Partial Correctness of `while`

which can be read as *if the premise is true then the conclusion is true*. I believe you met similar inference rules in COMP11120.

With the exception of one rule (the rule of consequence) these tell us how to deal with each kind of statement in our language i.e. they are *structural* in a similar way to the rewrite rules (although no longer deterministic). The rules form Hoare's Axiomatic Semantics for partial correctness. It is called a semantics as, like our previous operational semantics, it can be used for defining the meaning of programs. In the following we introduce and explain the rules but I leave fully worked examples to the end of the section. Sections 4.4 and 4.6 offer more advice on how to construct proofs of partial correctness.

`skip`

This is the most straightforward rule. If the pre-condition P is true for the initial state σ, and we execute the `skip` statement, then P will also be true for the final state (since it is also σ).

Sequences with ;

Sequences are the glue that hold programs together. I explain this rule next as they are also the glue that hold our proofs together. Sequences are where the postcondition of one statement becomes the precondition of the next.

If we have a way of establishing Hoare triples for S_1 and S_2, then we can produce a rule for $S_1;\ S_2$, by treating the post-condition of S_1 as the pre-condition of S_2. This can be written out as

$$\begin{array}{ll} \text{if} & \{P\}\ S_1\ \{Q\} \\ \text{and} & \{Q\}\ S_2\ \{R\} \\ \text{then} & \{P\}\ S_1;\ S_2\ \{R\} \end{array}$$

Notice how this is similar in structure to the operational view of sequencing. This should not be surprising as P, Q and R represent sets of states. Another way of reading this is *if when we have a state σ satisfying P and execute S_1 we get a state σ' satisfying Q and when we execute S_2 on σ' we get a state σ'' satisfying R then executing $S_1; S_2$ on a state σ satisfying P will give us a state σ'' satisfying R.*

`if-then-else`

Suppose that the pre-condition P is true for the initial state σ, and that we execute the statement

$$\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2.$$

Further suppose that Q is the post-condition. Then there are two cases to consider:

$\mathcal{B}[\![b]\!]\sigma$ is true In this case we execute statement S_1, and as pre-condition we have one more fact: that $\mathcal{B}[\![b]\!]\sigma$ is true. Thus we require:

$$\{P \wedge \mathcal{B}[\![b]\!]\}\ S_1\ \{Q\}$$

$\mathcal{B}[\![b]\!]\sigma$ is false In this case we execute statement S_2, and as pre-condition we have one more fact: that $\mathcal{B}[\![b]\!]\sigma$ is false.

$$\{P \wedge \neg\mathcal{B}[\![b]\!]\}\ S_2\ \{Q\}$$

Puting this all together, we can say that:

$$\begin{array}{ll} \text{if} & \{P \wedge \mathcal{B}[\![b]\!]\}\ S_1\ \{Q\} \\ \text{and} & \{P \wedge \neg\mathcal{B}[\![b]\!]\}\ S_2\ \{Q\} \\ \text{then} & \{P\}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\} \end{array}$$

As we see later, adding the condition to the precondition can never make the proof harder as $\{\ P\ \}\ S\ \{\ Q\ \}$ implies $\{\ P \wedge A\ \}\ S\ \{\ Q\ \}$ for any predicate A.

`while-do`

The `while`-loop construct is not too tricky to understand, but is the only one which cannot be automated as it requires some creativity.

As we discovered earlier, there is a special name for the pre-condition for this statement: a *loop-invariant*. This is the key to writing correct imperative code. It is called an invariant because every time we start and finish the body S of the `while`-loop $\texttt{while}\ b\ \texttt{do}\ S$ this loop-invariant condition is true.

However, we should take account of the relevant boolean conditions at each stage of the execution. If the loop-invariant is an invariant, then at the beginning of each execution of the loop body S, we know in addition that the boolean condition b must be true. At the end of the execution of the loop, the invariant must also be true. Thus

$$\{P \wedge \mathcal{B}[\![b]\!]\}\ S\ \{P\}$$

At the beginning of the execution of `while` b `do` S, the loop-invariant must be true, and it is also true at the end. In addition, if the loop terminates then the boolean condition must be false.

$$\{P\}\ \texttt{while}\ b\ \texttt{do}\ S\ \{P \wedge \neg \mathcal{B}[\![b]\!]\}$$

Puting it all together:

$$\begin{array}{ll} \text{if} & \{P \wedge \mathcal{B}[\![b]\!]\}\ S\ \{P\} \\ \text{then} & \{P\}\ \texttt{while}\ b\ \texttt{do}\ S\ \{P \wedge \neg \mathcal{B}[\![b]\!]\} \end{array}$$

Note, as we will see later, that $\{\ P\ \}\ S\ \{\ Q \wedge A\ \}$ implies $\{\ P\ \}\ S\ \{\ Q\ \}$ i.e. if we establish this for the negated condition then we can drop this if it is not needed later in the proof. However, it is often the case that we need the negated condition in the remaining steps.

Assignment $x := a$

You'll notice that so far we have not mentioned assignment $x := a$. This might seem strange as it is seemingly one of the more simple constructs. However, its inference rule is somewhat counterintuitive.

Before we continue let us quickly introduce some extra notation. We will write $P[a \mapsto b]$ for the predicate P with all expressions a replaced by the expression b. For example:

- $(x > 0)[x \mapsto 1] = (1 > 0)$
- $(x > 0)[y \mapsto 1] = (x > 0)$
- $(x > y)[x \mapsto y, y \mapsto x] = (y > x)$

Now let us consider what the rule for assignment should be. At first it would seem that we might want to write

$$\{\ P\ \}\ x := a\ \{\ P[x \mapsto \mathcal{A}[\![a]\!]]\ \}$$

i.e. assignment updates the value of x after the assignment. However, let us consider the following Hoare triple that can be shown to be valid using this rule

$$\{\ x = 1\ \}\ x := 2\ \{\ 2 = 1\ \}$$

as $(x = 1)[x \mapsto 2] = (2 = 1)$. This is clearly wrong. An alternative might be

$$\{\ P\ \}\ x := a\ \{\ P[a \mapsto x]\ \}$$

i.e. replace a by x. But this allows us write false things such as

$$\{\ x = 1\ \}\ x := 0\ \{\ x = 1\ \}$$

due to the fact that 0 does not appear in $x = 1$.

So what should the correct rule be? Let us suppose that we have an assignment $x := a$ that takes state σ to state σ' and that we have a postcondition P that is true of σ'. The state s' will have modified σ to set the value of x to $\mathcal{A}[\![a]\!]$. The original value of x in σ has been lost; indeed, it no longer matters. This suggests a way to find a suitable precondition. We know that

$$P(\sigma[x \mapsto a])$$

is true, from the above. The trick is noticing that this is the *composition* of three functions

$$P \circ ([x \mapsto a] \circ \sigma)$$

as the predicate P is a function from states to boolean values. Due to *associativity of function composition* this can be rewritten as

$$(P \circ [x \mapsto a]) \circ \sigma$$

which gives us a function that, when applied to σ, gives *true*. So the precondition is the postcondition P applied to the state where x has its new, modified, value.

This gives us the following rule:

$$\{\ P[x \mapsto \mathcal{A}[\![a]\!]]\ \}\ x := a\ \{\ P\ \}.$$

As this might still seem a little counter-intuitive let's check it on some examples. Firstly, to find a precondition P for

$$\{\ P\ \}\ x := 1\ \{\ x = 1\ \}$$

we can apply the rule to give

$$\begin{array}{rl} & \{\ 1 = 1\ \}\ x := 1\ \{\ x = 1\ \} \\ \equiv & \{\ \mathit{true}\ \}\ x := 1\ \{\ x = 1\ \} \end{array}$$

which is what we want, as $x = 1$ after assigning 1 to x after starting in any state. Next consider finding a precondition P for

$$\{\ P\ \}\ x := x + 1\ \{\ x = 2\ \}$$

by applying the rule we get

$$\begin{array}{rl} & \{\ x + 1 = 2\ \}\ x := x + 1\ \{\ x = 2\ \} \\ \equiv & \{\ x = 1\ \}\ x := x + 1\ \{\ x = 2\ \} \end{array}$$

which is clearly what we want. This is encouraging. Finally, let us check that we don't have false consequences. Consider finding a precondition P for

$$\{\ P\ \}\ x := 1\ \{\ x = 2\ \}$$

by applying the rule we get

$$\begin{array}{l} \{\ 1 = 2\ \}\ x := 1\ \{\ x = 2\ \} \\ \equiv \{\ \mathit{false}\ \}\ x := 1\ \{\ x = 2\ \} \end{array}$$

which says that there are no states such that starting in them and assigning 1 to x takes us to a state where $x = 2$.

Notice that in all of the above cases I have applied the assignment rule *backwards*. This is typically the easiest way to apply this rule, suggesting that our approach to proof construction should in general be backwards.

As an aside, in the original presentation of this approach there was a forward version of the rule written as

$$\{\ P\ \}\ x := a\ \{\ \exists y : x = a[x \mapsto y] \wedge P[x \mapsto y]\ \}$$

but this is rather difficult to use and we don't consider it here.

Rule of Consequence

There is one further rule, which involves logical operators. This is a very important rule but also a rule that is difficult to use because it is not guided by the *structure* of the program i.e. for the other rules when we see assignment we need the assignment rule, and similarly for other constructs. But this rule can be applied at any stage.

To motivate the need for this rule consider checking whether

$$\{\ x > y\ \}\ y := y + 1\ \{\ x \geq y\ \}$$

holds. We can work backwards from the postcondition and use the assignment rule to show that

$$\{\ x \geq y + 1\ \}\ y := y + 1\ \{\ x \geq y\ \}$$

holds. But this doesn't directly establish what we want to show. Somehow we want to conclude

$$\frac{\{\ x \geq y + 1\ \}\ y := y + 1\ \{\ x \geq y\ \}}{\{\ x > y\ \}\ y := y + 1\ \{\ x \geq y\ \}}$$

i.e. given something we know is true, the thing we want to be true is true. Here this is quite straightforward as we can use basic mathematical reasoning to show that whenever a state satisfies $x \geq y + 1$ it necessarily also satisfies $x > y$, and vice versa, i.e. the two predicates are equivalent (for the integers, not the reals). Therefore, these two preconditions specify the same set of states.

But now consider

$$\{\ x > y\ \}\ x := x + 1\ \{\ x > y\ \},$$

here this collapses to checking that

$$\frac{\{\ x + 1 > y\ \}\ x := x + 1\ \{\ x > y\ \}}{\{\ x > y\ \}\ x := x + 1\ \{\ x > y\ \}}$$

holds and it is no longer the case that the two preconditions are true for the same states (consider $[x \mapsto 0, y \mapsto 1]$). However, we can make the observation that we only need $x + 1 > y$ to hold for all states satisfying $x > y$ and not the other direction. This is because in the bottom Hoare triple we are only considering states where $x > y$ is true, so we will never need to check states where this is not true for any subexpressions.

A symmetric argument can be made for postconditions. Here we can argue that we go in the other direction as any state true at the top must also be true at the bottom. This is all captured in the rule of consequence's use of implication.

Important
Notice that the implications go in different directions for the pre and post conditions.

We will generally need to use the rule of consequence where we are forced to have one side of the

Hoare triple in a different form from the one we want. This can happen when dealing with a given pre or post condition and will often be encountered when dealing with loop invariants.

Confused?
I acknowledge that the rule of consequence is confusing but don't worry. There are plenty of examples in the rest of this section and Sections 4.4 4.6 go into more detail on how to construct proofs using this rule.

4.3.1 Some Small Example Proofs

The previous section introduce our rules but we haven't seen many examples of them being used. To help get us started let's look at some small proofs.

Max Value. As a first example consider the following program that finds the maximum of two variables

$$\texttt{if } x > y \texttt{ then } z := x \texttt{ else } z := y$$

Firstly, let us check that it does indeed find the maximum of 1 and 2 by checking

$$\{\ x = 1 \wedge y = 2\ \}\ \texttt{if } x > y \texttt{ then } z := x \texttt{ else } z := y\ \{\ z = 2\ \}$$

In the following derivation let $P \triangleq (x = 1 \wedge y = 2)$, we then present a partial correctness proof as a proof tree:

$$\dfrac{\dfrac{\{\ \mathit{false}\ \}\ z := x\ \{\ z = 2\ \}}{\{\ P \wedge x > y\ \}\ z := x\ \{\ z = 2\ \}} \quad \dfrac{\{\ P\ \}\ z := y\ \{\ z = 2\ \}}{\{\ P \wedge \neg(x > y)\ \}\ z := y\ \{\ z = 2\ \}}}{\{\ P\ \}\ \texttt{if } x > y \texttt{ then } z := x \texttt{ else } z := y\ \{\ z = 2\ \}}$$

The leaves of each branch use the assignment rule, the next step down uses the rule of consequence and the first step applies the rule for `if then else`. Check that you are happy that $(P \wedge x > y) \rightarrow \mathit{false}$. Notice that whenever we have a false precondition, e.g. $\{\ \mathit{false}\ \}\ S\ \{\ Q\ \}$, the Hoare triple will always hold as there are no states such that the precondition holds and we can say anything we want about them (e.g. this is quantification over an empty set, which is always true).

Important
The triple $\{\ \mathit{false}\ \}\ S\ \{\ Q\ \}$ is always true for any program S and postcondition Q. For any precondition $P \neq \mathit{false}$, the triple $\{\ P\ \}\ S\ \{\ \mathit{false}\ \}$ is (i) always false for terminating programs S, but (ii) always true for non-terminating programs S.

Now let us prove the program correct with respect to the intended specification, which can be

written as follows, assuming that we have a function max, which is a reasonable assumption to make.

$$\{ \quad \} \ \texttt{if}\ x > y\ \texttt{then}\ z := x\ \texttt{else}\ z := y\ \{\ z = \mathsf{max}(x, y)\ \}$$

In the following derivation let $M \triangleq (z = \mathsf{max}(x, y))$. I am naming certain expressions to make the proofs fit nicely in the page width in tree form. Later we will see a better way of writing proofs that handles larger proofs and has the space to make the rules used more explicit (although we will still name parts). The proof tree is then

$$\dfrac{\dfrac{\{\ x = \mathsf{max}(x, y)\ \}\ z := x\ \{\ M\ \}}{\{\ x > y\ \}\ z := x\ \{\ M\ \}} \quad \dfrac{\{\ y = \mathsf{max}(x, y)\ \}\ z := y\ \{\ M\ \}}{\{\ \neg(x > y)\ \}\ z := y\ \{\ M\ \}}}{\{ \quad \} \ \texttt{if}\ x > y\ \texttt{then}\ z := x\ \texttt{else}\ z := y\ \{\ M\ \}}$$

The structure is the same. The leaves are instances of the assignment rule but this time we had to be a bit clever about how we applied the rule of consequence. We made use of the fact that $x > y \to x = \mathsf{max}(x, y)$ to produce something in the form that we needed to apply the assignment rule. It is perhaps worth pointing out that it would have been better to use auxiliary variables above or included an argument that the values of x and y are not updated.

Swapping Values. Another very simple program is the one that swaps the values of x and y. The specification is that x and y should have the other values at the end. As we saw earlier, this involves adding some auxiliary variables into the specification, which we can write as

$$\{\ x = a \wedge y = b\ \}\ t := x; x := y; y := t\ \{\ x = b \wedge y = a\ \}$$

The proof of this Hoare triple can then be given as the following proof, presented in *linear* form[6] where a single application of a proof rule is given on a numbered line and the tree structure is indicated by referring to the rule used and the relevant lines.

1. $\{\ y = b \wedge x = a\ \}\ t := x\ \{\ y = b \wedge t = a\ \}$ by ass_p
2. $\{\ y = b \wedge t = a\ \}\ x := y\ \{\ x = b \wedge t = a\ \}$ by ass_p
3. $\{\ x = b \wedge t = a\ \}\ y := t\ \{\ x = b \wedge y = a\ \}$ by ass_p
4. $\{\ y = b \wedge x = a\ \}\ t := x; x := y\ \{\ x = b \wedge t = a\ \}$ by $\text{comp}_p(1, 2)$
5. $\{\ x = a \wedge y = b\ \}\ t := x; x := y; y := t\ \{\ x = b \wedge y = a\ \}$ by $\text{comp}_p(4, 3)$

This proof makes simple use of the assignment rule and the sequencing rule.

A `while`-loop. To illustrate the `while`-loop rule and the use of loop invariants let us consider the Hoare triple

$$\{\ x \geq z \wedge y \geq 0 \wedge z \geq 0\ \}\ \texttt{while}\ x > 0\ \texttt{do}\ x := x - 1; y := y + 1\ \{\ y \geq z\ \}$$

for a program that adds y to x using a `while`-loop. Let us briefly consider what the pre and post conditions mean here. The precondition restricts us to states with positive x, y, z and $x \geq z$ and we

[6] You should be familiar with this from Natural Deduction proofs in COMP11120.

are to show that we end up with $y \geq z$. Given that the loop adds y to z we immediately get a feeling that the triple is probably correct.

The first thing to do in this proof (and in general for programs with loops) is to work out what our loop invariant should be. We can look at the pre and postconditions to see what kind of thing might be helpful. These are both in terms of inequalities so we should look for an inequality containing both x and y. It is a fair assumption that we need to do something with $x+y$ as this value will be constant due to the semantics of the loop. At the end of the loop we want to know that $y \geq z$ and we know that x will be zero, so $x+y \geq z$ seems like a good candidate. We can also check that this holds at the beginning, which it does. However, we cannot use $x + y \geq z$ by itself to show that $y \geq z$ as x needs to be non-negative. So our loop invariant is $x + y \geq z \wedge x \geq 0$. Coming up with loop invariants can be difficult, and we discuss this further later.

In the following linear proof we will use $P \triangleq (x + y \geq z \wedge x \geq 0)$ as our loop invariant.

1. $\{\ x - 1 + y + 1 \geq z \wedge x - 1 \geq 0\ \}\ x := x - 1\ \{\ x + y + 1 \geq z \wedge x \geq 0\ \}$ by ass_p
2. $(x > 0) \rightarrow (x - 1 \geq 0)$ by arithmetic
3. $\{\ P \wedge x > 0\ \}\ y := y + 1\ \{\ x + y + 1 \geq z \wedge x \geq 0\ \}$ by $\text{cons}_p(1, 2)$
4. $\{\ x + y + 1 \geq z \wedge x \geq 0\ \}\ y := y + 1\ \{\ P\ \}$ by ass_p
5. $\{\ P \wedge x > 0\ \}\ x := x - 1; y := y + 1\ \{\ P\ \}$ by $\text{comp}_p(3, 4)$
6. $\{\ P\ \}$ `while` $x > 0$ `do` $x := x - 1; y := y + 1\ \{\ P \wedge \neg(x > 0)\ \}$ by $\text{while}_p(5)$
7. $(x \geq z \wedge y \geq 0) \rightarrow (x + y \geq z)$ by arithmetic
8. $(x \geq z \wedge z \geq 0) \rightarrow (x \geq 0)$ by arithmetic
9. $(x \geq z \wedge y \geq 0 \wedge z \geq 0) \rightarrow P$ by 7,8
10. $(P \wedge x \leq 0) \rightarrow (P \wedge x = 0) \rightarrow (y \geq z)$ by arithmetic
11. $\{\ x \geq z \wedge y \geq 0 \wedge z \geq 0\ \}$ `while` $x > 0$ `do` $x := x - 1; y := y + 1\ \{\ y \geq z\ \}$ by $\text{cons}_p(6, 9, 10)$

Notice that this makes heavy use of the rule of consequence. As mentioned earlier, this is typical for proofs using the `while`-rule where the pre and post conditions differ from the loop invariant.

You can view the above approach as a general framework for producing such proofs. We first found the loop invariant and then (i) showed that it was invariant in the loop via the rule for *while*, and (ii) connected the loop invariant to the initial pre and post conditions via the rule of consequence.

4.3.2 The Division Program Again

Let us now use this system to prove the partial correctness of our division algorithm. We already have the Hoare triple we want to establish. The program can be proved correct as follows, using our previous loop invariant of $P \triangleq (x = d \times y + r \wedge r \geq 0)$. Below I also use the definition $Q \triangleq (x = d \times y + r - y \wedge r - y \geq 0)$ for readability reasons.

1. $\{\ x = (d + 1) \times y + r - y \wedge r - y \geq 0\ \}\ d := d + 1\ \{\ Q\ \}$ by ass_p
2. $x = (d + 1) \times y + r - y = (d \times y) + y + r - y = d \times y + r$ by arithmetic

3. $y \leq r \rightarrow r - y \geq 0$ by arithmetic
4. $\{\ P \wedge y \leq r\ \}\ d := d + 1\ \{\ Q\ \}$ by $\text{cons}_p(1, 2, 3)$
5. $\{\ x = d \times y + r - y \wedge r - y \geq 0\ \} := r - y\ \{\ P\ \}$ by ass_p
6. $\{\ P \wedge y \leq r\ \}\ d := d + 1; r := r - y\ \{\ P\ \}$ by $\text{comp}_p(4, 5)$
7. $\{\ P\ \}$ `while` $y \leq r$ `do` $S_3\ \{\ P \wedge \neg(y \leq r)\ \}$ by $\text{while}_p(6)$
8. $\{\ x = r \wedge r \geq 0\ \}\ d := 0\ \{\ P\ \}$ by ass_p
9. $\{\ x \geq 0\ \}\ r := x\ \{\ x = r \wedge r \geq 0\ \}$ by ass_p
10. $\{\ x \geq 0\ \}\ r := x; d := d - 1\ \{\ P\ \}$ by $\text{comp}_p(8, 9)$
11. $\{\ x \geq 0 \wedge y > 0\ \}\ S\ \{\ P \wedge r < y\ \}$ by $\text{comp}_p(7, 10)$ (and cons_p)

Note how in this proof I have introduced names for subparts of the program and for expressions that will be reused. This is used to generally improve the readability of the proof. When doing this make sure you are careful that you use names consistently. It is also okay to skip some obvious steps. For example, in the last step we implicitly rely on $(x \geq 0 \wedge y > 0) \rightarrow (x \geq 0)$ and the cons_p rule. But again be careful that they are obvious and if in doubt include everything.

4.3.3 A More Complicated Example

Let us now consider a much more complicated program i.e. the GCD program introduced in Example 1.2.4. To write a specification for this function we will use a mathematical function gcd that computes the greatest common divisor of two numbers characterised by the following axioms (which we will also need to use in the proof):

A_1. $\mathsf{gcd}(a, a) = a$ for $a > 0$

A_2. $\mathsf{gcd}(a, b) = \mathsf{gcd}(b, a)$ for $a, b > 0$

A_3. $b < a \Rightarrow \mathsf{gcd}(a, b) = \mathsf{gcd}(a - b, b)$ for $a, b > 0$

Using this function we can then write the specification of the problem as follows:

$$\{\ x > 0 \wedge y > 0\ \}\ C\ \{\ z = \mathsf{gcd}(x, y)\ \}$$

To make things friendlier here we are going to update the program slightly so that it copies the input variables into auxiliary variables and then operates on these[7]. Here is the updated program:

```
a := x
b := y
while a != b do (
  if (a>b) then
    a := a–b
```

[7] I really should have introduced the auxiliary variables into the specification rather than the program. But at this point I do not feel it is worth the effort to change this example. Hopefully it is not difficult to convince yourselves that the effect is the same.

```
  else
    b := b−a
)
z:=a
```

We now need to find a loop invariant. This is quite straightforward here as we can use $\mathsf{gcd}(x, y) = \mathsf{gcd}(a, b)$. To convince ourselves that this is a loop invariant we can consider axiom A_3 which tells us that $\mathsf{gcd}(a, b) = \mathsf{gcd}(a - b, b)$ given certain restrictions on a and b. As before, we need to strengthen the loop invariant slightly for it to work, and here we have to use $a > 0 \wedge b > 0$ as this helps us apply the above axioms.

To make the proof a bit more readable we introduce a list of definitions:

- $P_{xy} \triangleq (x > 0 \wedge y > 0)$
- $P_{ab} \triangleq (a > 0 \wedge b > 0)$
- $I \triangleq \mathsf{gcd}(x, y) = \mathsf{gcd}(a, b)$
- $Q_1 \triangleq (a > 0 \wedge y > 0 \wedge \mathsf{gcd}(x, y) = \mathsf{gcd}(a, y))$
- $Q_2 \triangleq (a - b > 0 \wedge b > 0 \wedge \mathsf{gcd}(x, y) = \mathsf{gcd}(a - b, b))$
- $Q_3 \triangleq (a > 0 \wedge b - a > 0 \wedge \mathsf{gcd}(x, y) = \mathsf{gcd}(a, b - a))$

These are then used in the following linear proof of partial correctness:

1. $\{P_{xy} \wedge \mathsf{gcd}(x, y) = \mathsf{gcd}(x, y)\}\ a := x\ \{Q_1\}$ by ass_p
2. $P_{xy} \rightarrow (P_{xy} \wedge \mathsf{gcd}(x, y) = \mathsf{gcd}(x, y))$ by logic
3. $\{P_{xy}\}\ a := x\ \{Q_1\}$ by $\mathrm{cons}_p(1, 2)$
4. $\{Q_1\}\ b := y\ \{P_{ab} \wedge I\}$ by ass_p
5. $\{P_{xy}\}\ a := x; b; = y\ \{P_{ab} \wedge I\}$ by $\mathrm{comp}_p(3, 4)$
6. $\{Q_2\}\ a := a - b\ \{P_{ab} \wedge I\}$ by ass_p
7. $(P_{ab} \wedge I \wedge a > b) \rightarrow Q_2$ by A_3
8. $\{P_{ab} \wedge I \wedge a > b\}\ a := a - b\ \{P_{ab} \wedge I\}$ by $\mathrm{cons}_p(6, 7)$
9. $a > b \rightarrow a \neq b$ by arithmetic
10. $\{P_{ab} \wedge I \wedge a > b \wedge a \neq b\}\ a := a - b\ \{P_{ab} \wedge I\}$ by $\mathrm{cons}_p(8, 9)$
11. $\{Q_3\}\ b := b - a\ \{P_{ab} \wedge I\}$ by ass_p
12. $(P_{ab} \wedge I \wedge b > a) \rightarrow Q_3$ by A_2, A_3
13. $\{P_{ab} \wedge I \wedge b > a\}\ b := b - a\ \{P_{ab} \wedge I\}$by $\mathrm{cons}_p(11, 12)$
14. $b > a \rightarrow (\neg(a > b) \wedge a \neq b)$ by arithmetic

15. $\{P_{ab} \wedge I \wedge \neg(a > b) \wedge a \neq b\}\ b := b - a\ \{P_{ab} \wedge I\}$ by $\text{cons}_p(13, 14)$
16. $\{P_{ab} \wedge I \wedge a \neq b\}$ `if` $a > b$ `then` X_2 `else` $X_3\ \{P_{ab} \wedge I\}$ by $\text{if}_p(10, 15)$
17. $\{P_{ab} \wedge I\}$ `while` $a \neq b$ `do` $X_1\{P_{ab} \wedge I \wedge a = b\}$ by $\text{while}_p(16)$
18. $(a > 0 \wedge a = b) \rightarrow \mathsf{gcd}(a, b) = a$ by A_1
19. $(\mathsf{gcd}(x, y) = \mathsf{gcd}(a, b) \wedge a = b) \rightarrow a = \mathsf{gcd}(x, y)$ by 18
20. $\{P_{ab} \wedge I\}$ `while` $a \neq b$ `do` $X_1\{a = \mathsf{gcd}(x, y)\}$ by $\text{cons}_p(17, 19)$
21. $\{a = \mathsf{gcd}(x, y)\}\ z := a\ \{z = \mathsf{gcd}(x, y)\}$ by ass_p
22. $\{P_{xy}\}\ GCD\ \{z = \mathsf{gcd}(x, y)\}$ by $\text{comp}_p(5, 20, 21)$

Again I implicitly name subparts of the program. Notice the non-trivial use of the above axioms about gcd. To construct the proof I started with the loop and then worked backwards from the end of the loop to the beginning, applying the conditional and assignment rules. Notice that the structure of the proof is very similar for each branch of the conditional. This is quite common and I could possibly have structured the proof more nicely to demonstrate this.

4.3.4 Final Thoughts and Exercises

It is worth stressing a point that was made earlier, that post conditions only apply to terminating programs (we see how to prove termination in Section 4.5). This important so it gets an extra box.

Important
The *partial correctness property* is: 'If the pre-condition is true of the initial state, *and* the statement S terminates, *then* the post-condition *must* be true of the final state. **Note very carefully** that if the program does *not* terminate, the postcondition may be true or false.

Exercise 4.4

Explain why we can use *any* post-condition with a non-terminating program such as

```
while true do skip
```

And the partial correctness property will hold.

Before attempting the following exercises you might find it helpful to read Section 4.4.

Exercise 4.5

Prove partial correctness of

$$\{\ x \neq y \wedge x = a \wedge y = b\ \}\ S\ \{\ (a > z \vee b > z) \wedge (z = a \vee z = b)\ \}\}$$

where S is the following program for finding the minimum of two variables

$$\texttt{if } x < y \texttt{ then } z := x \texttt{ else } z := y$$

Why did I introduce the auxiliary variables a and b?

Exercise 4.6

Write a program to compute the function $f(x) = x^2 + x + 1$ and prove that it is correct e.g. formulate an appropriate partial correctness specification and prove that it holds.

Exercise 4.7

Write a program to compute the function

$$f(x) = \begin{cases} 1 & \textit{if} \quad x = 0 \\ 2 & \textit{if} \quad x > 0 \wedge x < 10 \\ x + 1 & \textit{otherwise} \end{cases}$$

and prove that it is correct e.g. formulate an appropriate partial correctness specification and prove that it holds.

Exercise 4.8

Prove the partial correctness of the extended division program you wrote for Exercise 1.8 with respect to the specification you wrote for Exercise 4.3. You will find it useful to refer to the above proof of the unextended division program.

Exercise 4.9

Complete the inference tree for partial correctness for the following program which raises 2 to the power of x (provided $x \geq 0$).

p := 1; **while** 1 ≤ x **do** (p := p⋆2; x := x−1)

Exercise 4.10

Prove the partial correctness of the logarithm program you wrote for Exercise 1.13 with respect to the specification you wrote for Exercise 4.3.

Exercise 4.11

Consider the programs capturing coding functions (and their inverses) in Section 2.1. Write specifications for them and prove their correctness.

As further exercises you should take the other programs introduced in Examples and Exercises in Section 1.2.2 and have a go writing their specifications and proving their partial correctness.

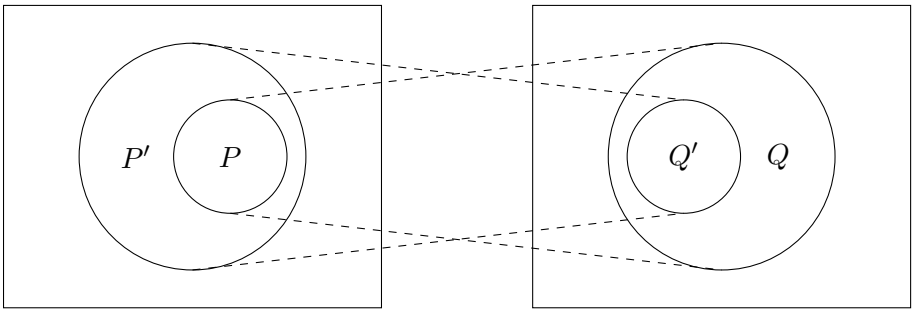

Figure 4.1: A picture to help us think about the rule of consequence.

4.4 Some More Hints

Proving partial correctness requires two kinds of creative steps:

1. Generating loop invariants; and
2. Applying the rule of consequence to put things together.

The rest of the steps should be relatively straightforward as they are driven directly by the *structure* of the program. In this section we discuss these two steps in further detail and give some hints on how to apply them. More practical details on how to approach the construction of proofs can be found in Section 4.6.

4.4.1 The Rule of Consequence Again

Applying the rule of consequence can be tricky so I want to give you some hints and some derived rules that might make things clearer. But first I want to revisit what

$$\{\ P\ \}\ S\ \{\ Q\ \}$$

means. If this Hoare triple holds then for **any** state σ if $P(\sigma)$ is true then executing S on σ will produce **some** state σ' such that $Q(\sigma')$ is true. I have highlighted **any** and **some** because they highlight which states we care about. The important thing to note is that executing S on states that satisfy P will not necessarily result in **all** states that satisfy Q. Another important point is that if we show that the triple holds for more states than those satisfying P then we are still okay, as those more states include those that satisfy P.

Figure 4.1 attempts to clarify this intuition. P and Q represent the sets of states we care about initially i.e. those that satisfy P and Q. We can safely replace P by a *larger* set P' (where $P \subseteq P'$) due to the above argument that establishing the Hoare triple for a more general precondition is okay. We can safely replace Q by a *smaller* set Q' (where $Q' \subseteq Q$) as if we start in a state s satisfying P and execute S on σ and end in a state σ' in Q' then we also have a state that is in Q.

Often we don't need to both enlarge the set of sates satisfying the precondition and shrink the set of sates satisfying the postcondition. This suggests that we can split the rule of consequence into two separate *derived* rules:

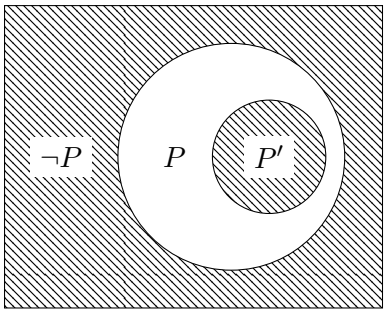

Figure 4.2: A picture to remind us about implication.

$$\frac{\{\ P'\ \}\ S\ \{\ Q'\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}}\ \text{if } P \to P' \qquad \frac{\{\ P'\ \}\ S\ \{\ Q'\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}}\ \text{if } Q' \to Q$$

The rule on the left is called *precondition strengthening* and the rule on the right is called *postcondition weakening*. We use the term strengthening because the assumed direction of the proof is downwards. In mathematics A is *stronger* than B if anything that follows from B also follows from A i.e. $A \to B$. So in the precondition strengthening rule P is stronger than P' and we replace P' by P as we move downwards. A symmetric motivation is used for the name of the postcondition weakening rule.

Confused?
I like to remember the above as going 'up' on the left and 'down' on the right.

So our notion of bigger and smaller is captured by implication. If you struggle to see why $P \to P'$ means that P must be included in P' then consider the picture in Figure 4.2. The shaded bits are the things that are true for $P \to P'$ (recall that this is equivalent to $\neg P \vee P'$). If P' does not include P then $P \to P'$ cannot be true, as for it to be true we need to be able to shade the full box.

There is a common case that deserves a special rule. Consider the proof

$$\frac{\{\ x = x\ \}\ x := 1\ \{\ x = 1\ \}}{\{\ \mathit{true}\ \}\ x := 1\ \{\ x = 1\ \}}$$

where we use the rule of consequence to go from the top application of the assignment rule to the Hoare triple we want to hold at the bottom. Here it is not just the case that $P \to P'$ but $P \leftrightarrow P'$ i.e. P and P' represent the same set of states. This happens most often when we remove equivalences,

and we can define the following rules for this:

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \}}{\{\ P \wedge E = E\ \}\ S\ \{\ Q\ \}} \qquad \frac{\{\ P\ \}\ S\ \{\ Q \wedge E = E\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}}$$

$$\frac{\{\ P \wedge E = E\ \}\ S\ \{\ Q\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}} \qquad \frac{\{\ P\ \}\ S\ \{\ Q\ \}}{\{\ P\ \}\ S\ \{\ Q \wedge E = E\ \}}$$

Here we are allowed to delete equivalences in directions that the rule of consequence does not normally allow, this is because the resulting triples are true for an equivalent set of states. In fact, one could introduce the more general rule

$$\frac{\{\ P_1\ \}\ S\ \{\ Q_1\ \}}{\{\ P_2\ \}\ S\ \{\ Q_2\ \}} \text{ if } P_1 \leftrightarrow P_2 \textit{ and } Q_1 \leftrightarrow Q_2$$

(*) Exercise 4.12

> Prove that the derived rules given here are sound i.e. only allow us to derive things that are true. Hint: consider which states they apply to.

As applications of these rules often require you to find valid implications and equivalences here are some that might be useful:

- $(a \wedge b) \rightarrow a$
- $a \rightarrow (a \vee b)$
- $a > b \rightarrow a \geq b$
- $a > b \rightarrow a + 1 > b$
- $a > b \leftrightarrow a \geq b + 1$ (for integers)
- $a \geq b \leftrightarrow (a > b \vee a = b)$
- $(a \geq 0 \wedge b \geq 0) \rightarrow (a + b) \geq 0$
- $(a \geq 0 \wedge \neg(a < 0)) \leftrightarrow a = 0$

You should replace a and b by useful things and note that many of these have symmetric versions.

4.4.2 Finding Loop Invariants

I introduced the above derived rules before this short discussion of how to find loop invariants because the most common way to find useful loop invariants is to start with something we already have and either *strengthen* it or *weaken* it.

In general, whilst constructing a proof we will reach a point where we have a `while`-loop and there will be an expression that is currently true before the loop and an expression that is currently true

after the loop and we have to find something that connects the two. In other words, we are faced with the challenge of proving

$$\{\ A\ \}\ \texttt{while}\ b\ \texttt{do}\ C\ \{\ B\ \}$$

using the rule

$$\frac{\{\ P \wedge \mathcal{B}[\![b]\!]\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\ P \wedge \neg\mathcal{B}[\![b]\!]\ \}}$$

If we find an appropriate loop invariant P we will then need to show that $A \to P$ and $(P \wedge \neg\mathcal{B}[\![b]\!]) \to B$ i.e. we will need to weaken A to get P and strengthen $(P \wedge \neg\mathcal{B}[\![b]\!])$ to get B. This gives us a strategy for finding P i.e. start with either A or B (we might only have one at this stage of the proof, probably B if we are going backwards) and strengthen/weaken it until things work.

Notice, also that we are going to have to use P to establish

$$\{\ P \wedge \mathcal{B}[\![b]\!]\ \}\ S\ \{\ P\ \}$$

which might suggest that if we are going backwards from P to $P \wedge \mathcal{B}[\![b]\!]$ that we need to make $\mathcal{B}[\![b]\!]$ hold at the start of the loop. However, note that

$$\frac{\{\ P\ \}\ S\ \{\ P\ \}}{\{\ P \wedge \mathcal{B}[\![b]\!]\ \}\ S\ \{\ P\ \}}$$

by precondition strengthening as $(P \wedge \mathcal{B}[\![b]\!]) \to P$. Although often b is implied by P anyway. A common example of this is where P includes $x \geq 0$ and the condition b is $x < 0$ then after the loop we have $x \geq 0 \wedge \neg(x < 0)$ which can be used to establish $x = 0$.

4.5 An Axiomatic System for Total Correctness

If, in addition to being partially correct, we prove that every loop must terminate then we say that the program S is *totally correct* with respect to the pre and postconditions.

We use a different syntax for Hoare triples to indicate that we are talking about *total* correctness.

Definition 4.5.1

> A *Hoare Triple* for total correctness is a triple consisting of two predicates, and a statement, which we will write as:
>
> $$[\ P\ ]\ S\ [\ Q\ ]$$
>
> We will refer to P as the *pre-condition*, and Q as the *post-condition* for the statement S, just as we did for partial correctness.

The only difference between the axiom system for partial correctness and the new one for total correctness lies in the rule for `while`-loops. For the other parts of the language we just add rules such as

$$\frac{\{\ P\ \}\ S_1; S_2\ \{\ Q\ \}}{[\ P\ ]\ S_1; S_2\ [\ Q\ ]}$$

but I won't write these all out because that would be boring.

Instead, let's consider the `while` rule. As we have already seen (Section 4.2), the idea is to have an expression (called E here) that (i) gets smaller after executing the loop, and (ii) will be bounded below whenever whenever we execute the loop. What do I mean by bounded-below? That there is some value k such that it is always the case that $E \geq k$. Commonly we use $k = 0$ i.e. that the expression is non-negative (this is what we do below). It should be clear that the combination of these two facts implies that there are a bounded number of iterations of the loop e.g. the starting value of the expression E. If this is not clear then note that (ii) ensures that we cannot keep decreasing E forever. The rule is given as:

$$\frac{[\ P \wedge \mathcal{B}\,[\![b]\!] \wedge E = n\ ]\ C\ [\ P \wedge E < n\ ]}{[\ P\ ]\ \texttt{while}\ b\ \texttt{do}\ C\ [\ P \wedge \neg\mathcal{B}\,[\![b]\!]\ ]}\ \text{if } P \wedge \mathcal{B}\,[\![b]\!] \rightarrow E \geq 0$$

Notice that this just gives us something else to establish as well as the loop invariant. The expression E what was previously defined as a *loop variant*. We can introduce a weaker ruler that is often more useful where we assume that E is decremented by 1 on each step:

$$\frac{[\ P \wedge \mathcal{B}\,[\![b]\!] \wedge E = n\ ]\ C\ [\ P \wedge E = n - 1\ ]}{[\ P\ ]\ \texttt{while}\ b\ \texttt{do}\ C\ [\ P \wedge \neg\mathcal{B}\,[\![b]\!]\ ]}\ \text{if } P \wedge \mathcal{B}\,[\![b]\!] \rightarrow E \geq 0$$

As an exercise it should not be difficult to show that this rule can be derived from the one above.

Example 4.5.2

Let us consider the program

while x $>$ 0 **do** x := x$-$1

and the total correctness property $[\ \mathit{true}\ ]\ S\ [\ \mathit{true}\ ]$ i.e. we want the program to terminate on all inputs. The first step is finding a loop variant. Here this is trivial as x strictly decreases on each step. The proof can then be given as:

1 $[\ x - 1 < n\ ]\ x := x - 1\ [\ x < n\ ]$ by ass_t
2 $(x > 0 \wedge x = n) \rightarrow (n \geq x) \rightarrow (n > x - 1)$ by arithmetic
3 $[\ x > 0 \wedge x = n\ ]\ x := x - 1\ [\ x < n\ ]$ by $\text{cons}_t(1, 2)$
4 $[\ \mathit{true}\ ]\ S\ [\ \mathit{true}\ ]$ by $\text{while}_t(3)$

Whilst it may have been more straightforward to use the second rule above, I have used the first rule.

4.5.1 Division Again

In this course we will not consider complicated total correctness proofs. It is sufficient to see a proof for a relatively simple program that has a simple loop variant. For this we will use the division algorithm again. Remember that earlier we noticed that it has a loop variant r. We can use this to establish

$$[\ x \geq 0 \wedge y > 0\ ]\ S\ [\ P \wedge r < y\ ]$$

where S is our division program, which is

r := x; d := 0; **while** y ≤ r **do** (d := d+1; r := r−y)

in case we had forgotten! Again we define some useful predicates to help with our proof

- $P \triangleq (x = (d \times y) + r \wedge r \geq 0 \wedge y > 0)$
- $Q_1 \triangleq (x = (d \times y) + r - y \wedge r - y \geq 0 \wedge r - y = n - y)$
- $Q_2 \triangleq (x = ((d+1) \times y) + r - y \wedge r - y \geq 0 \wedge r - y = n - y)$

noting that $Q_1 \equiv Q_2$ but we use both to show the effect of the assignment rule at one point. The linear proof can then be given as follows.

1. $[\ Q_1\ ]\ d := d+1\ [\ Q_2\ ]$ by ass_t
2. $x = ((d+1) \times y) + r - y = (d \times y) + y + r - y = d \times y + r$ by arithmetic
3. $r = n \rightarrow r - y = n - y$ by arithmetic
4. $[\ P \wedge y \leq r \wedge r = n\ ]\ d := d+1\ [\ Q_2\ ]$ by $\text{cons}_t(1,2,3)$
5. $[\ Q_2\ ] := r - y\ [\ P \wedge r = n - y\ ]$ by ass_t
6. $[\ P \wedge y \leq r \wedge r = n\ ]\ d := d+1; r := r - y\ [\ P \wedge r = n - y\ ]$ by $\text{comp}_t(4,5)$
7. $(r = n - y \wedge y > 0) \rightarrow r < n$
8. $[\ P \wedge y \leq r \wedge r = n\ ]\ d := d+1; r := r - y\ [\ P \wedge r < n\ ]$ by $\text{cons}_t(6,7)$
9. $(P \wedge y \leq r) \rightarrow y > 0$
10. $[\ P\ ]$ `while` $y \leq r$ `do` $S_3\ [\ P \wedge \neg(y \leq r)\ ]$ by $\text{while}_t(8,9)$
11. $[\ x = r \wedge r \geq 0\ ]\ d := 0\ [\ P\ ]$ by ass_p
12. $[\ x \geq 0 \wedge y > 0\ ]\ r := x\ [\ x = r \wedge r \geq 0\ ]$ by ass_t
13. $[\ x \geq 0 \wedge y > 0\ ]\ r := x; d := d - 1\ [\ P\ ]$ by $\text{comp}_t(11,12)$
14. $[\ x \geq 0 \wedge y > 0\ ]\ S\ [\ P \wedge r < y\ ]$ by $\text{comp}_t(10,13)$ (and cons_p)

Notice that the main way in which this differs from the partial correctness proof is that we now need the precondition $y > 0$, which wasn't required before. This should not be surprising, as this is the condition that ensured termination. Notice that instead of using $r < n$ directly I used the expression $r = n + y$ that resolves to $r = n$ when applying the assignment rule and then separately showed that this expression implied $r < n$.

Exercise 4.13

Prove the total correctness of the extended division program you wrote for Exercise 1.8 with respect to a total version of the specification you wrote for Exercise 4.3.

Exercise 4.14

Prove total correctness for the program given in Exercise 4.9

Exercise 4.15

Prove the total correctness of the logarithm program you wrote for Exercise 1.13 with respect to a total version of the specification you wrote for Exercise 4.3.

4.6 Guidance on Proofs of Correctness

This section gives some practical advice on how to approach constructing proofs of correctness. The section is structured around questions I received in examples classes last year.

4.6.1 How do I start?

I will say some more general things before explicitly answering this question, so please bear with me. Firstly, it is important to note that

There is no general algorithm

as the correctness problem is *undecidable*. Remember that this tells us that there is (provably) no algorithm that can always produce the right answer to this question. It is important that you are familiar with this idea that there may not be an algorithm to follow. Many real-world problem solving tasks are of this form.

However, all is not lost as

Many parts are algorithmic

and usually a proof just requires a few small creative steps to set things up and then mechanical application of the rules.

There are only two rules that require a creative step:

- The rule for `while` requires a *loop invariant* for partial correctness and additionally a *loop variant* for total correctness. The rule doesn't tell us what this should be but it gives hints of how it should fit into the proof. I discuss a method for finding suitable invariants and variants below.

- The rule of consequence doesn't tell you when to apply it. This rule is the glue so in general you should only use it when you need to stick bits of your proof together.

Everything else can be mechanical, but what does this mean? With the exception of the rule of consequence, the inference rules tell us what the structure of the proof should look like. If we need to prove something of the form

$$\{\ P\ \}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\ Q\ \}$$

then the related rule tells us that we need to prove two further partial correctness statements:

$$\{\ P \wedge b\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P \wedge \neg b\ \}\ S_2\ \{\ Q\ \}$$

even the `while` rule defines the *structure* of what we need to prove, it just has something else we need to fill in.

So, where do we start? Let us run through two examples where I describe the steps in detail.

Example One

Let us consider the partial specification statement

$$\{\ x > 0\ \}\ \texttt{if}\ y > 0\ \texttt{then}\ z := x + y\ \texttt{else}\ z := x - y\ \{\ z > 0\ \}$$

how do we start producing a proof for this statement? We can immediately apply the if_p rule to decompose this proof into two subproofs. How do we know this? Because of the structure of the program, the top-level construct is `if then else` and this is the construct targeted by the if_p rule. If we are constructing a linear proof we can now write

1. $\{\ x > 0 \wedge y > 0\ \}\ z := x + y\ \{\ z > 0\ \}$ by ?
2. $\{\ x > 0 \wedge \neg(y > 0)\ \}\ z := x - y\ \{\ z > 0\ \}$ by ?
3. $\{\ x > 0\ \}\ \texttt{if}\ y > 0\ \texttt{then}\ z := x + y\ \texttt{else}\ z := x - y\ \{\ z > 0\ \}$ by $\text{if}_p(1, 2)$

but now we need to produce proofs for statements 1 and 2. In a sense we can start again. Given the partial correctness statement

$$\{\ x > 0 \wedge y > 0\ \}\ z := x + y\ \{\ z > 0\ \}$$

how should we prove this? The top-level construct is assignment so we should use the assignment rule. But the assignment rule places restrictions on the precondition and postcondition. We can try applying it backwards from the postcondition $z > 0$ i.e. the assignment rule tells us that this partial correctness statement

$$\{\ x + y > 0\ \}\ z := x + y\ \{\ z > 0\ \}$$

is universally true. But the precondition we have is $x > 0 \wedge y > 0$. So where to go from here? The rule of consequence tells us that if $(x > 0 \wedge y > 0) \to (x + y > 0)$ then we can glue these together. Thankfully this implication does hold. These two steps may not appear very intuitive. Instead of doing this we could introduce the following *derived rule* that follows directly from the given rules:

$$\frac{P \to Q[x \mapsto \mathcal{A}[\![a]\!]] \qquad \{\ Q[x \mapsto \mathcal{A}[\![a]\!]]\ \}\ x := a\ \{\ Q\ \}}{\{\ P\ \}\ x := a\ \{\ Q\ \}}$$

If you want you can use this rule in your proofs and call it $\text{ass}_p + \text{cons}_p$. We can now update our linear proof to be

1. $\{\ x + y > 0\ \}\ z := x + y\ \{\ z > 0\ \}$ by ass_p
2. $(x > 0 \wedge y > 0) \to (x + y > 0)$ by maths
3. $\{\ x > 0 \wedge y > 0\ \}\ z := x + y\ \{\ z > 0\ \}$ by $\text{cons}_p(1, 2)$
4. $\{\ x > 0 \wedge \neg(y > 0)\ \}\ z := x - y\ \{\ z > 0\ \}$ by ?
5. $\{\ x > 0\ \}\ \texttt{if}\ y > 0\ \texttt{then}\ z := x + y\ \texttt{else}\ z := x - y\ \{\ z > 0\ \}$ by $\text{if}_p(3, 4)$

I will leave filling in the remaining ? as an exercise. Note that it is completely symmetric to what we have just done but with a slightly less trivial implication. I discuss how to deal with this implications further below, in case they are troubling you.

It is important to reflect on what we just did. We iteratively did the following things

- Applied the structural rules to decompose the proof into smaller proofs
- If the partial correctness statement is an assignment, apply the derived $\text{ass}_p + \text{cons}_p$ rule

This doesn't cover the `while` rule but is a method for algorithmically producing proofs of correctness for programs without loops. This is important as proofs of correctness for programs with loops will eventually turn into proofs of correctness for programs without loops as the rules are applied.

Note that this isn't the only way we could prove this program correct. We could have applied the rule of consequence and assignment rule more creatively. For example, we could have produced this (partial) proof.

1. $\{\ x > 0 \wedge y > 0 \wedge x + y = x + y\ \}\ z := x + y\ \{\ x > 0 \wedge y > 0 \wedge z = x + y\ \}$ by ass_p
2. $(z = x + y \wedge x > 0 \wedge y > 0) \rightarrow (z > 0)$ by maths
3. $\{\ x > 0 \wedge y > 0\ \}\ z := x + y\ \{\ z > 0\ \}$ by $\text{cons}_p(1, 2)$
4. $\{\ x > 0 \wedge \neg(y > 0)\ \}\ z := x - y\ \{\ z > 0\ \}$ by ?
5. $\{\ x > 0\ \}$ `if` $y > 0$ `then` $z := x + y$ `else` $z := x - y\ \{\ z > 0\ \}$ by $\text{if}_p(3, 4)$

Here we notice that the precondition $x > 0 \wedge y > 0$ and the equality $z = x + y$ are strong enough together to imply the required postcondition $z > 0$ so we can apply the following derived rule

$$\frac{\{\ P \wedge a = a\ \}\ x := a\ \{\ P \wedge x = a\ \} \qquad (P \wedge x = \mathcal{A}[\![a]\!]) \rightarrow Q}{\{\ P\ \}\ x := a\ \{\ Q\ \}}$$

This is relying on the other *direction* of the rule of consequence. Previously we were *weakening* the precondition but now we are *strengthening* the postcondition. Note that this is *sound* as everything follows from the original set of rules but it is not in itself *complete* in the sense that it relies on P not containing x for it to be directly applied. If you are not familiar with the notions of sound and complete then look them up online.

You might wonder where these derived rules are coming from. This is just me writing down the intuition captured by multiple applications of the existing rules. You don't need derived rules but sometimes they can help give shortcuts in proofs.

Example Two

Now let us consider an example containing a loop. We will use a very simple program and think about the steps we need to go through. Let us prove the following partial correctness statement:

$$\{\ x = a \wedge y = b \wedge x \geq 0\ \}\ \texttt{while}\ x > 0\ \texttt{do}\ (y := y + 1; x := x - 1)\ \{\ y = a + b\ \}$$

The program is *destructive*, in the sense that it destroys the initial values of x and y, so we need to use auxiliary variables in the specification.

We cannot proceed as before as the while_p rule doesn't fit with the pre and postconditions we have. We need a loop invariant. I discuss more tricks for how to find a loop invariant below. For now let us look at the loop body

$$y := y + 1; x := x - 1$$

and think about an *expression* that will have the same value at the *start* and *end* of this loop body (**not during**). This should be straightforward; the value of $x + y$ goes up by 1 and down by 1 during the loop body so its value stays the same. But this isn't a predicate. To make it a predicate we can consider the value of $x + y$ at the *start* of the loop (before it executes). This is $a + b$ so we will try

$$x + y = a + b$$

as a loop invariant. I phrase it like this (*try*) as this loop invariant might not be strong enough for what we need.

This tells us that we will be trying to produce a proof of

$$\{\ x + y = a + b\ \}\ \texttt{while}\ x > 0\ \texttt{do}\ (y := y + 1; x := x - 1)\ \{\ x + y = a + b \wedge \neg(x > 0)\ \}$$

but before we do that we should check that proving this allows us to prove the thing we wanted to prove in the first place. To do this we should connect this statement to our original statement via the rule of consequence, which requires us to establish

1. $(x = a \wedge y = b \wedge x \geq 0) \rightarrow (x + y = a + b)$
2. $(x + y = a + b \wedge \neg(x > 0)) \rightarrow y = a + b$

The first implication holds trivially but the second does not; we are missing something. What we really want is to show that at the end of the loop $x = 0$. We know this is true as $x \geq 0$ at the beginning but we cannot use this information here as we haven't established that this is preserved by the loop. This tell us that we need to strengthen the loop invariant by adding $x \geq 0$. So our loop invariant is now

$$R \triangleq (x + y = a + b \wedge x \geq 0)$$

Our application of the rule of consequence now holds as

$$(x + y = a + b \wedge x \geq 0 \neg(x > 0)) \rightarrow (x + y = a + b \wedge x = 0) \rightarrow y = a + b$$

At this point our linear proof looks like

1. $\{\ R\ \}\ \texttt{while}\ x > 0\ \texttt{do}\ (y := y + 1; x := x - 1)\ \{\ R \wedge \neg(x > 0)\ \}$ by ?
2. $(x = a \wedge y = b \wedge x \geq 0) \rightarrow (x + y = a + b)$ by maths
3. $(x + y = a + b \wedge x \geq 0 \neg(x > 0)) \rightarrow (x + y = a + b \wedge x = 0) \rightarrow y = a + b$ by maths
4. $\{\ x = a \wedge y = b \wedge x \geq 0\ \}\ \texttt{while}\ x > 0\ \texttt{do}\ (y := y + 1; x := x - 1)\ \{\ y = a + b\ \}$ by $\text{cons}_p(1, 2, 3)$

where I have already named the loop invariant to make things look neater.

We can now apply the `while` rule to produce the following partial correctness statement without loops that we have to prove

$$\{\ R \wedge x > 0\ \}\ y := y + 1; x := x - 1\ \{\ R\ \}$$

When faced with a sequence of assignments I tend to apply the assignment rule in sequence backwards. Here that would look like

$$\{x-1+y+1 = a+b \wedge x-1 \geq 0\}\ y := y+1\ \{x-1+y = a+b \wedge x-1 \geq 0\}\ x := x-1\ \{x+y = a+b \wedge x \geq 0\}$$

and now we can apply the rule of consequence to check that

$$(x + y = a + b \wedge x \geq 0 \wedge x > 0) \rightarrow (x - 1 + y + 1 = a + b \wedge x - 1 \geq 0)$$

which holds as $x > 0 \rightarrow x - 1 \geq 0$ (this is the important bit, the rest is trivial).

This might have gone quite fast and it might help to break this down. Notice that this corresponds to an extension of above derived rule $\text{ass}_p + \text{cons}_p$ to n assignments. What I just did corresponds to the linear proof:

1. $\{\ x-1+y+1=a+b \wedge x-1 \geq 0\ \}\ y := y+1\ \{\ x-1+y=a+b \wedge x-1 \geq 0\ \}$ by ass_p
2. $\{\ x-1+y=a+b \wedge x-1 \geq 0\ \}\ x := x-1\ \{\ x+y=a+b \wedge x \geq 0\ \}$ by ass_p
3. $\{\ x-1+y+1=a+b \wedge x-1 \geq 0\ \}\ y := y+1; x := x-1\ \{\ R\ \}$ by $\text{comp}_p(1,2)$
4. $(x+y=a+b \wedge x \geq 0 \wedge x > 0) \rightarrow (x-1+y+1=a+b \wedge x-1 \geq 0)$ by maths
5. $\{\ R \wedge x > 0\ \}\ y := y+1; x := x-1\ \{\ R\ \}$ by $\text{comp}(3,4)$
6. $\{\ R\ \}$ `while` $x > 0$ `do` $(y := y+1; x := x-1)\ \{\ R \wedge \neg(x > 0)\ \}$ by $\text{while}_p(5)$

and this fills in the ? in the above proof. Notice that I could have organised this slightly differently as I could have applied the rule of consequence in different places. I find this the neater way of organising things but the proof system allows you to be creative. However, it is very important that you make non-trivial applications of the rule of consequence explicit. An example of a trivial application is one that applies very basic arithmetic reasoning to an expression $a = b$ so that it is in the form $a = a$ and then removes the equivalence. Another example is of the form $x > 0 \wedge x < 0$ being resolved to *false*.

A summary of how to start

Let me finish this section with a rough guide to how to approach constructing a proof of correctness for the partial correctness specification

$$\{\ P\ \}\ S\ \{\ Q\ \}$$

For total correctness one just needs to add the loop variant in the appropriate place. Let us assume the program is of the form

$$S_1; (\texttt{while}\ b\ \texttt{do}\ S_2); S_3$$

Now follow these steps:

1. If there is a loop find a loop invariant R; if it is a total correctness specification find a loop variant also. See below for hints on how to find these
2. Deal with the bits of the program before and after the loop i.e. produce proofs for $\{\ P\ \}\ S_1\ \{\ R\ \}$ and $\{\ R \wedge \neg b\ \}\ S_3\ \{\ Q\ \}$. If S_1 and S_3 are empty then you just need to show that $P \rightarrow R$ and $(R \wedge \neg b) \rightarrow Q$. If you cannot produce this proofs (or the implications do not hold) then you might need to strengthen the loop invariant.
3. Now produce a proof for the loop body i.e. $\{\ R \wedge b\ \}\ S_2\ \{\ R\ \}$ (or with a loop invariant if it is total correctness). To do this follow the above steps of
   (a) applying the structural rules until you get an assignment (or sequence of assignments)
   (b) applying the assignments backwards from the postcondition to produce some new precondition
   (c) applying the rule of consequence to connect the new precondition to the existing one

4.6.2 What are we proving?

This question is asking what does it mean if we have produce a proof of a correctness statement i.e. if we have a proof such as

$$\frac{\{\ x > 0 \wedge x > 0\ \}\ y := x\ \{\ y > 0\ \} \qquad \{\ x > 0 \wedge \neg(x > 0)\ \}\ \texttt{skip}\ \{\ y > 0\ \}}{\{\ x > 0\ \}\ \texttt{if}\ x > 0\ \texttt{then}\ y := x\ \texttt{else skip}\ \{\ y > 0\ \}}$$

what have we achieved?

There are two approaches to this question:

- If the inference system is *sound* then we have shown that the statement at the bottom is *true*. What does sound mean? It means that whenever we have a rule of the form $\frac{premise}{conclusion}$ if the premise is true then the conclusion is necessarily true. Soundness means that we cannot prove things that are false. For rules without a premise this means that they are *axioms* i.e. always true. Note that our rules have metavariables in them, like P, Q and S, this means that they should hold for *any* instantiation of those metavariables.

- If the statement at the bottom is *true* what does that mean? This returns to the meaning of the triple $\{\ P\ \}\ S\ \{\ Q\ \}$ which is that given any state σ if $P(\sigma)$ is true then when executing S on σ (recall the operational semantics) to produce σ' (recall that it is deterministic and terminating) we have $Q(\sigma)$ being true. More mathematically we might say

  $$\forall \sigma, \sigma' \in State : (P(\sigma) \wedge \langle S, \sigma \rangle \Rightarrow \sigma') \rightarrow Q(\sigma')$$

  Perhaps the meaning of $P(\sigma)$ and $Q(\sigma')$ is unclear. When we write

  $$\{\ x > 0\ \}\ \texttt{if}\ x > 0\ \texttt{then}\ y := x\ \texttt{else skip}\ \{\ y > 0\ \}$$

  we are implicitly defining the precondition predicate

  $$P(\sigma) = \sigma(x) > 0$$

  and the postcondition predicate

  $$Q(\sigma) = \sigma(y) > 0$$

  so our notation is a bit lazy in this respect. A special case of this is when we write $\{\ true\ \}\ S\ \{\ Q\ \}$ or $\{\ \ \}\ S\ \{\ Q\ \}$ where *true* stands for the predicate

  $$true(\sigma) = true$$

  i.e. the predicate that always returns true and ignores the state, and we take the *empty* predicate to be *true* as it places no restrictions on the states.

4.6.3 What direction does it go in?

This is a very similar question to *how do I start* as the question is really asking whether we start with the goal or axioms. Sometimes in mathematical proof there seems to be a moral obligation to start

from axioms and derive the thing you want to derive. Perhaps more pragmatically some inference systems work like that, for example the *modus ponens* rule

$$\frac{P \to Q \qquad P}{Q}$$

starts from knowing $P \to Q$ and P and deriving Q. It is designed to go from top to bottom as from the bottom it is not clear what should go on the top.

However, many proof systems, this one included, are *goal-directed* which means that we start from the thing we want to prove and decompose this statement into other things we want to prove. So to answer the original question, we go *'up'*.

Although this is not the full story. As suggested earlier, I tend to

- Go 'up' if the program is an if statement or a sequence
- Go 'up and backwards' if the program is an assignment or sequence of assignments i.e. use the derived rule introduced earlier to go up on the right hand side, backwards through the assignment rule and then join up with the rule of consequence
- Start form the 'middle' if the program is a while i.e. find the loop invariant and produce an upwards proof from that and a downwards proof to join it up with the initial pre and post conditions

4.6.4 Does a loop invariant or a loop variant always exist?

No, for the simple reason that the program might not be correct and might not be terminating. As a more complicated answer, the proof system is *not complete*. Which means that there are correctness statements we can write down but cannot prove in the system. A simple example is

$$\{\ \mathit{true}\ \}\ S\ \{\ \mathit{false}\ \}$$

in general as this collapses to solving the Halting Problem. As another argument consider

$$\{\ \mathit{true}\ \}\ x := x\ \{\ P\ \}$$

for some predicate P, this collapses to determining whether P is valid. We haven't discussed the specification language we use for predicates but generally we have used first-order logic with arithmetic. We know that first-order logic is partially decidable and arithmetic is undecidable. So for any reasonably useful specification language for predicates the proof system is clearly incomplete.

4.6.5 How do I find loop invariants?

The first thing to point out is that there is no general algorithm that given a triple

$$\{\ P\ \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\ Q\ \}$$

always find a loop invariant R such that

- $P \to R$

- $(R \wedge \neg b) \rightarrow Q$
- $\{\ R \wedge b\ \}\ S\ \{\ R\ \}$

which is what we need for R to be a usable loop invariant. A lot of research has gone into finding good ways of solving this problem but due to the general undecidability of arithmetic it is not even in general possible to check if the above implications hold for a given R.

So now I have told you there is no algorithm what should you do? This is one of the creative parts of this course but there are some hints you can follow:

- The loop invariant should contain all of the variables that the loop body updates and are referred to either after the loop body or in the postcondition. If it doesn't then it is almost certain that the loop invariant will not be strong enough.
- It is common to have a loop condition of the form $x > 0$ and to assume at the end of the loop that $x = 0$. For this to hold you need the loop invariant to include $x \geq 0$ (or similar). Just because this holds before the loop doesn't mean you can assume it holds afterwards, anything you want to be true after the loop has to go in the loop invariant.
- Sometimes you might have definitions that are invariant across the whole program. These still need to go in the loop invariant but you can ignore them (just carry them around) during the proof.
- Look at the postcondition, what do you need to be true at the end? If the postcondition is an inequality then you probably want an inequality as the loop invariant. Possibly start with the postcondition as the loop invariant and work out what you need to change. Remember that you will need to show $(R \wedge \neg b) \rightarrow Q$.
- The loop invariant should be invariant to the *effect* of the loop body. So a good starting point is working out what this effect and then use this to produce an expression invariant to that effect. As some examples:
  - The loop body $x := x - 1; y := y + 1$ has the effect of making x smaller by 1 and y bigger by 1, it is then simple to see that those 1s cancel out in $x + y$
  - The loop body $x := x - 1; y := y + 2$ has the effect of making x smaller by 1 and y bigger by 2, we can cancel out the effect of the 2 by multiplying 1 by 2, giving us an invariant expression of $2x + y$.
  - The loop body $x := x + y$ has the effect of making x bigger by y but this doesn't immediately give us an invariant expression. If we also know that $y > 0 \wedge x > 0$, perhaps from the loop condition, then the expression $x > 0$ is invariant.

  Once we have an invariant expression we can easily find an invariant by finding the expressions value at the start of the loop.
- If the loop body contains conditional statements then the loop invariant has to hold for each path through the loop. One approach would be to consider each path separately and then guard them. For example, given a loop body of the form `if` b `then` S_1 `else` S_2 one could attempt to build a loop invariant of the form $(b \rightarrow R_1) \wedge (\neg b \rightarrow R_2)$.

The best practice is to look at examples and try it out. As a last bit of advice, if you write python programs for the `while` programs you can add assert statements to check if an expression really is invariant.

4.6.6 How do I find loop variants?

Due to the Halting Problem we know that it is not possible to produce an algorithm that finds suitable loop variants. But for the kinds of programs we will be looking at loop variants will be easier to find. In most cases the `while` loops are really `for` loops with a loop counter decreasing on each step and the loop terminating when that counter becomes zero. Here I will point out that for such cases we can use the derived rule for total correctness of the form (introduced earlier):

$$\frac{[\ P \wedge \mathcal{B}[\![b]\!] \wedge E = n\ ]\ C\ [\ P \wedge E = n - 1\ ]}{[\ P\ ]\ \texttt{while}\ b\ \texttt{do}\ C\ [\ P \wedge \neg\mathcal{B}[\![b]\!]\ ]}\ \text{if } P \wedge \mathcal{B}[\![b]\!] \rightarrow E \geq 0$$

where we explicitly show that the expression E gets smaller by 1 on each iteration.

Here I discuss a few special alternative cases:

- Nothing gets smaller and the loop condition bounds something from above. For example, in `while` $x < 100$ `do` $x := x + 1$ the value of x just gets bigger but the loop terminates when x is 'big enough'. But there are immediately a few things that decrease. Simply the value of $-x$, as long as $x > 0$ to begin with, and we can show that $(x > 0 \wedge -x = n) \rightarrow (1 - x < n)$, which is what we would need. Additionally, the value $100 - (x + 1)$ is going to tend towards 0 (for $x \geq 0$).
- Either E_1 or E_2 will get smaller. In this case you can just use $E_1 + E_2$ as a loop variant.
- Either E_1 gets smaller or E_2 gets bigger and is always positive. In this case you can use $E_1 - E_2$ as a loop invariant as this decreases when E_1 gets smaller and E_2 gets bigger (depending on E_2 being strictly positive).
- The loop condition does not use an inequality so there is no inherent notion of bounding from above or below. In this case you will need to infer the bound. For example, if the loop condition is $x \neq 0$ and the loop invariant contains $x \geq 0$ then we can show that x is bounded from below. However, note that $x \neq 0$ is not enough as the loop body might be $x := x - 2$.

In general you should look at the loop and ask *why should this terminate*? If you can work that out the you should be able to phrase the reason as an expression which strictly decreases.

4.6.7 How do I know what implications I need for the rule of consequence?

I would argue that these appear in well-defined places when constructing a proof, even though you're allowed to apply the rule of consequence wherever you want. In fact, what I haven't told you is that there is a way of automating all of this, assuming loop invariant annotations, via the generation of *verification conditions* that can be passed to a theorem prover[8]. These verification conditions are the implications that appear whilst constructing a proof. The places they occur when I produce proofs are in my derived assignment rule, i.e. attaching the initial precondition to the new one, and when relating the loop invariant to the initial pre and post conditions.

[8] Chapter 3 of these notes describes how this is done: `http://www.cl.cam.ac.uk/~mjcg/Teaching/2015/Hoare/Notes/Notes.pdf`. These notes also discuss the soundness and completeness of the proof system. Other good references exist and if you are interested in this topic do ask me about them.

4.6.8 How do I solve the implications?

This is mostly straightforward maths and logic. There are some hints in Section 4.4. Remember that a proof is an argument that something is true and you just need to write down enough for the argument to be convincing. You don't need to break down $(a > 0 \wedge b > 0) \rightarrow (a + b > 0)$ any further, it is clearly true. But something very complex involving a number of reasoning steps should be broken down to show what those steps are.

4.7 What about Arrays?

I am not going to give you reasoning rules for programs with arrays because they are outside the scope of this course[9]. But I will quickly point out why things are tricky. We cannot just write

$$\{\ P[b[\mathcal{A}[\![a_1]\!]] \mapsto \mathcal{A}[\![a_2]\!]]\ \}\ b[a_1] := a_2\ \{\ P\ \}$$

i.e. use the assignment rule, as this would allow us to prove

$$\{\ x = y \wedge b[y] = 0\ \}\ b[x] = 1\ \{\ x = y \wedge b[y] = 0\ \}$$

which is clearly false. The problem is that one variable might *alias* another. This problem is often called the *frame* problem as we need to know what is allowed to change outside of our frame of reference. To solve this problem we need to add a special rule and some extra axioms that allow us to reason about the *equivalence* of arrays.

4.8 Practical Correctness

Important
The contents of this section is non-examinable.

This chapter has introduced some fundamental yet powerful techniques for proving the correctness of sequential imperative programs (i.e. most of the programs we write). At this point I point out that there are other powerful techniques for proving the correctness of non-sequential and non-imperative programs, but these are out of scope for a first-year undergraduate course.

However, these techniques need quite a bit of tool support to be practically useful. Such tools do exist, and if you are interested in playing with one then contact me directly and I'll send you some links, however, I think it would also be useful to show you some practical tools that you can use to assert similar notions of correctness in programs you are already writing.

The tool that I am referring to is the programming language concept of *assertions*. Let's look at a practical $7 billion bug: which caused the loss of Ariane-5 Flight 501.

```
// Calculates the sum of a (int) + b (int)
// and returns the result (int).
```

[9] If you are very interested then I believe the third year course on Hoare Logic at Cambridge includes arrays and the notes are available online.

```
int sum(int a, int b) {
  return a + b;
}
```

What could possibly go wrong with such simple code? By adding an assertion we can check.

```
// Calculates the sum of a (int) + b (int)
// and returns the result (int).
int sum(int a, int b) {
  assert (Integer.MAX_VALUE - a >= b);
  return a + b;
}
```

The assertion is checking that there is not an integer overflow. With the integers used in `java` an overflow is likely to return a negative value. In the particular code used in Ariane-5, the issue was the use of 16-bit unsigned integers in the control program, and the new rocket having significantly more powerful engines.

The real beauty of assertions lies in the fact that by default they are *not* compiled into your code. In `java` you can switch them on and force the testing of all assertions by using the compile-time flag `-enableassertions`. So, to recap:

- Assertions are as efficient as comments (in that they are not compiled-in unless you force them to be);
- They are an excellent guide to documenting your code (you are adding in-line unit tests to your code); and
- During development, you can switch them on and discover every case where your assumptions about your own code are wrong!

.

My recommendation is that you use them to document the permitted input values to functions, the expected range of outputs of functions and for the loop variants and invariants for complicated loops. For example, I'd use an assertion to check that the input to a factorial function was non-negative, and another one to check that the calculated answer was a positive integer.

However, you should be careful. Assertions are turned off by default and if you want to report the bad behaviour in production code you should use exceptions and normal error handling mechanisms.

As well as being dynamically checkable, there exist tools for *statically* checking `java` assertions. Again, if you are interested in these then contact me.

Chapter 5

Complexity and Asymptotic Analysis

In this part of the course we will look at measures of program execution efficiency. In other words, how to answer the question *which program is better?* We will begin by discussing how we can analyse algorithms directly and then use this to motivate a technique for approximating their behaviours as the problem size increases. This discussion will focus on *upper bounds* on the amount of *time* required. We will extend the discussion to also consider *lower bounds* and *space complexity*. We then meet the concept of *complexity classes*, which are a way in which we can group functions together based on their complexity. The chapter will finish with a discussions about practical applications of these concepts.

Learning Outcomes

At the end of this Chapter you will:

- Be able to explain the 'Big-Oh' notation (and its relatives Ω and Θ);
- Be able to analyse and compare the asymtotic complexities of imperative programs; and
- Be able to describe the relationship between time complexity and space complexity and the notion of a *complexity class*

5.1 Setting the Scene

To set the scene we answer a few clarifying questions about what this chapter is focussing on.

What are we analysing?

We are analysing programs (or algorithms, they mean the same thing here) to measure their *complexity*. This is usually taken to mean predicting the resources required by the algorithm. Sometimes this is memory, communications bandwidth, or computer hardware, but usually we are concerned with the time taken to execute the algorithm. And for the first part of this chapter this is what we will focus on.

Which version of a program are we analysing?

If we wanted to write a program to check if an input is larger than 1000 we could write the following program:

```
result := 0
while(n>0 & result=0) do
  (if n != 1000 then n := n-1 else result:=1)
```

If we give this program 1000 it terminates immediately with the correct answer. If we give the program 999 it will perform 999 iterations before it terminates with the correct answer. If we give the program 10000 it will perform 9000 iterations before it terminates with the correct answer. If we assume that the program could receive numbers in the range 1 to 10000 with uniform probability then the average number of iterations performed will be

$$\frac{1}{10000} + \left(\sum_{n=1}^{999} \frac{1}{10000} n\right) + \left(\sum_{n=1001}^{10000} \frac{1}{10000}(n - 1000)\right)$$
$$=$$
$$\frac{1}{10000}\left(1 + \frac{999 \times 1000}{2} + \left(\frac{10000 \times 10001}{2} - \frac{1000 \times 1001}{2}\right) + (10000 \times -1000)\right)$$
$$=$$
$$\frac{1}{10000}\left(1 + 499500 + (50005000 - 500500) - 10000000\right) = 4000.4$$

Which took some non-trivial calculations to find out. Which behaviour are we interested in? The best case? The worst case? The likely case? How do we know what the likely case is? Above I assumed that the inputs were drawn uniformly from some range but we rarely know such details. If we are implementing a sorting function are the inputs likely to be already sorted? If we are implementing a function to turn an NFA into a DFA, is the input almost an DFA already?

Practically we often think about the common case but in these notes we will mainly be focussing on the *worst case* as this allows us to more easily compare programs without worrying about what the expected use case is.

What's the difference between programs and problems?

Previously we were very particular about the difference between a program and the function it computes i.e. the problem it solves. That same distinction is still important here. Two programs can compute the same function (solve the same problem) and have different complexities. But we might also be interested in the inherent complexity of a problem itself. For example, finding the maximum value in an array is a problem that inherently requires us to look at every element in an array. No solution can be a correct solution without doing this. This suggests a *lower bound* on the complexity of any solution. Throughout this chapter we will abuse terminology to talk about a problem's complexity. When we do this we mean *the best possible program that could solve that problem.*

Therefore, when we talk about programs we will usually think in terms of *upper bounds* (worst case) and when we talk about problems we talk about *lower bounds* (best case). We clarify this more later.

What are we not doing?

This isn't a course in algorithms. We won't be introducing complicated algorithms to solve specific problems. In fact, we won't be looking at many programs. The main learning outcome is to understand

the definition of Big-Oh notation and what it *means*. This is a first year course. We won't be exploring advanced material on computational complexity. The only model of computation we have met is the `while` language, so we won't be looking at things from the context of Turing Machines and we won't be considering parallel, non-deterministic, or random computation, all of which change the story.

We are **not** analysing programs in terms of their readability or understandability. This is not what we mean by complexity here. Such things are important but this chapter is interested in the *complexity of the computation* performed by a program not the program itself.

5.2 Simplifying Assumptions

To answer the question of how long a program will take to execute we need to answer some more simple questions first. For example, how long does it take a computer to compute $1 + 2$? This will depend on the architecture and its instruction set. For some reduced instruction set architectures, computing $1 + 2$ might require multiple instructions to place 1 and 2 into registers first, taking multiple cycles. For a complex instruction set there may be a single instruction that does everything in one cycle, but typically cycles take longer. This distinction might not matter for adding two 32-bit integers. But what about multiplication or division? What if we are doing security for e-commerce involving the RSA cryptographic protocols, where we might need to perform operations on 200-400 digit numbers. These will be implemented as a sequence of machine instructions operating on vectors of 32 bit unsigned integers. Thus the number of operations performed will depend on the size of the integers concerned. In this course we have been considering $\mathbb{Z}$ in general; but arguably arithmetic does not have fixed cost for all integers. As a simplifying assumption we can assume that we only deal with numbers that fit into 32-bit integers, or at least numbers for which the cost of arithmetic operations on them is fixed.

Another example of the above point is that we do not have a native division operator in the `while` language, meaning that division takes many operations to perform. However, most processors will include a bit shift division that has similar cost to addition.

Can we access and write to all memory in constant time? Not necessarily, to perform an operation on a data value the computer may need to move it into a register, and move the result. More generally, computers have memory hierarchies with different access times. As a simplifying assumption we can assume that all accesses occur in constant time. However, in many practical programs memory access time is the limited factor and in such cases it should take part in the complexity analysis.

As part of complexity analysis we measure the usage of some resource but the units that we measure will have a significant impact on the result. Particularly, for time complexity we need to decide which operations matter and what their associated cost should be.

Important
The instructions or operations that we measure, is an informed choice made by the computer scientist. Ideally, this choice should be made so that the results obtained are useful.

5.3 Analyzing Algorithms

Recall the division algorithm introduced in Exercise 1.2.1:

```
1   r := x; d := 0;
2   while y ≤ r do
3     d := d+1; r := r−y;
```

How many primitive operations are performed executing this program? Let's count the operations used on each line. Line 1 contains two assignments. Line 2 performs the loop check. If the loop is executed N times then we perform $N+1$ $\leq$ operations. Similarly, on line 3 we will perform N + operations, N − operations, and $2N$ assignments. The only question we now need to address is how big is N? Notice that variable d is assigned the value of 0; and each time we iterate through the loop we increment d by 1. At the end of the execution of the loop, d will have the value $\lfloor \frac{x}{y} \rfloor$, and this is thus the value of N as well. So, if we assume that all of these operations have the same cost, the cost of this program is

$$3 + 5 \left\lfloor \frac{x}{y} \right\rfloor$$

Now let us consider the slightly more complex program from Example 1.2.5 where the following program checks whether x is prime.

```
1  y:=x−1;
2  z:=1;
3  while (y>1 ∧ z=1) do (
4    r:=x;
5    (while (y ≤ r) do r:=r−y);
6    (if (r=0) then z:=0);
7    y:=y−1;
8  )
```

Let's consider the case where x is prime, this is the worst case as when x is non-prime the outer loop will terminate early. In this case we check if every number less than x divides x. There are $x-2$ such numbers so the outer loop executes $x-2$ times. The inner loop is the algorithm we used above, so we can reuse the results from that analysis. Given the three initial operations (two assignments and a −) before the outer loop and the additional 2 assignments, 1 check and one − inside the outer loop, the cost of this program in this worst case is

$$3 + (x-2)\left(5 + 5\left\lfloor \frac{x}{y} \right\rfloor\right)$$

We could also make a similar calculation for cases where the outer loop terminates early, but we would need to know how early, which is not possible to compute in general as we don't know how primes are distributed in general. But as we motivated previously, the worst case is usually what we want to know about (or at least it is often what people talk about. Later I point out that it is not always that useful).

Exercise 5.1

Perform a similar count of operations for the square-root program shown in Example 1.2.2.

Exercise 5.2

Perform a similar count of operations for the factorial program you wrote in answer to Exercise 1.10.

Exercise 5.3

Perform a similar count of operations for the log-base-two program you wrote in answer to Exercise 1.13.

5.4 Asymptotic Analysis or Order of Growth

There are a number of the problems with performing an exact analysis of the sort we did in the previous section:

- The exact operation counts often end up being extremely complicated;
- It is often necessary to consider many different cases;
- There is no easy way to analyze sub-components of an algorithm and then combine the results for the whole program; and
- Different compilers (or compiler options) might affect the result.

Computer Scientists use *Asymptotic Analysis* or *"Big-Oh" notation* to solve all of these problems; how's that for economy of effort?

So far we have been counting operations, such as addition, subtraction and comparison. It may not have been apparent, but on different machines, we may get very different performances for a particular algorithm. However, unless it has a very strange instruction set – say a one cycle matrix-multiply instruction – there will be a relationship between the size of the input and the length of time taken to run the algorithm. Consider the following program that sums the numbers from 1 to n.

```
sum := 0; while(n>0) do (sum := sum+n; n := n−1)
```

When $n = 50$ the program should (roughly) perform half the number of steps as when $n = 100$. Consider two machines, one where operations take A units of time and another where they take B units of time. If $A = 2$ and $B = 10$ then on the first machine the $n = 100$ case is considerably faster than the $n = 50$ case on the second. But on the same machines the relationship between the running time and the size of n is the same. We don't want to have to consider how fast performing operations is on any particular machine, so how do we abstract away from this point that one machine might implement addition faster than another machine? We should ignore any constants like A and B in our calculations.

Observation 5.4.1

This observation allows us to ignore the constant scaling factor for the time taken.

In short: if the algorithm is acceptably fast, but a speed-up is required — just buy a faster machine! The other observation is that in the end, for big enough problems, we can ignore some of the terms of our exact complexity, because they become insignificant. We explore this idea in the following examples but you have already met this idea in Section 5.1 of the COMP11120 material.

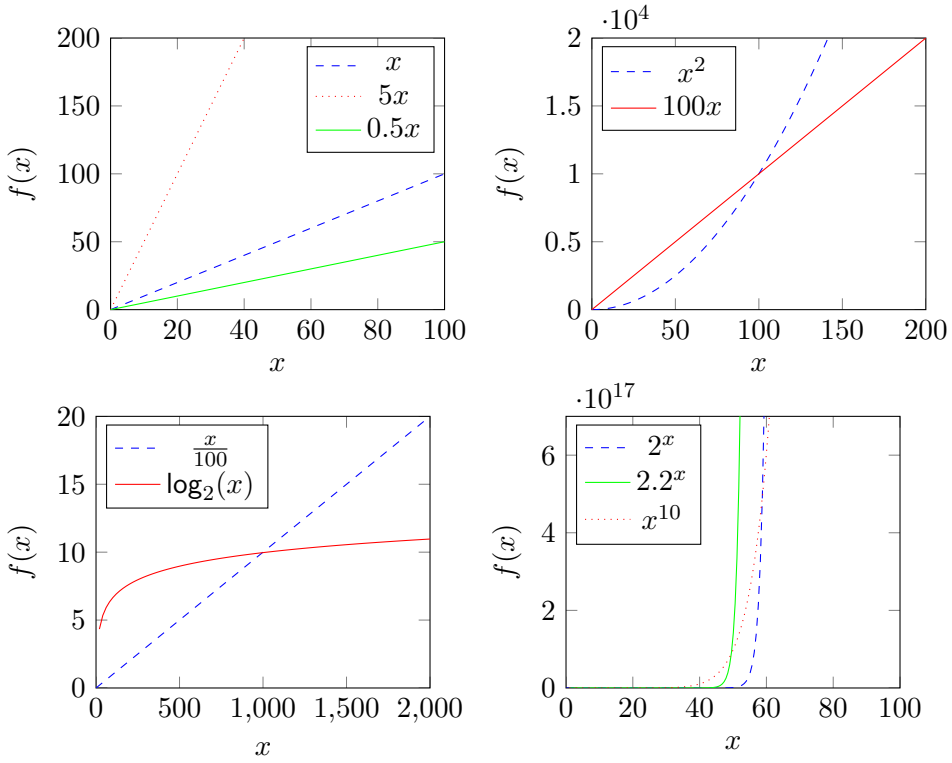

Figure 5.1: Some functions plotted against each other.

Example 5.4.2

Consider the two functions, each of type $\mathbb{N} \to \mathbb{N}$:

$$\begin{aligned} f(n) &= n \\ g(n) &= n^2 \end{aligned}$$

It does not matter which value of n we select, it will always be the case that $n \leq n^2$. We say that n^2 *dominates* n.

Example 5.4.3

Suppose that the two functions (again $\mathbb{N} \to \mathbb{N}$) this time are:

$$\begin{aligned} f(n) &= 100n \\ g(n) &= n^2 \end{aligned}$$

Now, if $n < 100$ then $f(n) > g(n)$. For example at $n = 10$ we get $f(n) = 1000$ and $g(n) = 100$. But *eventually* $g(n)$ will be bigger than $f(n)$. In this case "eventually" means for any value of $n > 100$; for other functions this may differ. See Figure 5.1 for a relevant graph.

There are some less intuitive relationships between functions, as illustrated in Figure 5.1. For example, $\frac{x}{100}$ might feel like it's going to be very small but it is a linear function of x and will grow linearly with x. However, $\mathsf{log}_2(x)$ grows much more slowly than x. Notice that $\frac{1000}{100} = 10$ and $\mathsf{log}_2(1000) < 10$ (as $2^{10} = 1024$).

Definition 5.4.4

We say the function g *eventually dominates* function f, whenever there exists $k : \mathbb{N}$ such that:

$$\forall\ (n : \mathbb{N}).\ n > k \to g(n) > f(n)$$

In other words, whenever n is greater than k, and we have $g(n) > f(n)$, then we can say that g eventually dominates f.

Observation 5.4.5

This observation allows us to ignore terms in a function that are eventually dominated by another – bigger – term in the function.

Puting Observations 5.4.1 and 5.4.5 together we have arrived at the following useful conclusion for polynomial complexities: if

$$T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

then it will eventually be dominated by Cn^k, for some $C > 0$. If, in addition, we do not care about the scaling constant we can simple talk about the function n^k. This is important enough to restate.

Important
Observation 5.4.5 tells us that we can view any polynomial function f of order/degree k as having the same complexity as the function n^k as there will always be a constant C such that Cn^k will eventually dominate f.

We can now introduce the "Big-Oh" notation. We will define a set of functions associated with a particular function g. This set of functions is written $O(g)$, and members of the set have the following property: if $f \in O(g)$ then there is a (positive) constant $C > 0$ such that f is eventually dominated by $C \times g$. Formally, we have the following definition:

Definition 5.4.6

If $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $f \in O(g)$, then there exists $k \in \mathbb{N}$ and $C > 0$ such that for all $n > k$:

$$f(n) \leq Cg(n)$$

5.5 Properties of 'Big-Oh'

In this section we will demonstrate some of the properties of the notation. You should be aware that many people are informal and sloppy in their use of the notation, so be prepared for this. In particular, instead of writing the function properly you will see things such as $O(1)$ and $O(n)$. These are actually shorthand for the polynomial functions $f(n) = 1$ and $f(n) = n$ respectively. In places we may also be sloppy.

The first simple property of the notation is that it is reflexive (if you cannot remember what this means then look back at the material in COMP11120).

Lemma 5.5.1

$f \in O(f)$

Proof

Pick $k = 0$ and $C = 1$. Then for all $n \in \mathbb{N}$:

$$f(n) = Cf(n) \leq Cf(n)$$

□

The trick we used in the above proof is that the definition of the notation says that there exists k and C so we are allowed to provide these values to demonstrate that a function belongs to a certain set.

Exercise 5.4

Show that "Big-Oh" is *not* symmetric; *i.e.* if $f \in O(g)$ then it is not necessarily the case that $g \in O(f)$.

Exercise 5.5

Show that "Big-Oh" is transitive; *i.e.* if $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.

Exercise 5.6

Argue that "Big-Oh" is a pre-order relation. **Hint:** Recall the definition given for a relation to be a pre-order from COMP11120. Or, just look it up!

Exercise 5.7

Is "Big-Oh" anti-symmetric? *i.e.* if $f \in O(g)$ and $g \in O(f)$, then must the two functions f and g be the same?

Let us next show that the constant functions $O(1)$ are a strict subset of the linear functions $O(n)$. Before we do this, let us consider what $O(1)$ means. These are all the functions whose running time is *independent* of the size of the input.

Lemma 5.5.2

$$O(1) \subsetneq O(n)$$

i.e. $O(1)$ *is a strict subset of* $O(n)$.

Proof

Suppose $f \in O(1)$. Then there exists k, C such that for all $n > k$:

$$f(n) \leq C.1 = C$$

We must now show that $f \in O(n)$ as well. Choose $k' = \max(k, 1)$ and $C' = C$. Then for all $n > k'$ we will have:

$$f(n) \leq C \leq Cn$$

Thus $f \in O(n)$.

Now we will show that there is at least one function that is in $O(n)$ that is not in $O(1)$. We choose the function $g(n) = n$. This is clearly in $O(n)$ by Lemma 5.5.1. However, it is not in $O(1)$. Suppose that g *was* a member of $O(1)$. Then there exists k and C such that for all $n > k$

$$g(n) = n \leq C$$

But, by choosing $n > \max(C, k)$, we will violate this condition.

□

Next we establish the observation we made in Observation 5.4.5 that we can treat a polynomial g containing the largest polynomial n^k in the same way as n^k. Here this means that $O(g) = O(n^k)$.

Lemma 5.5.3

Suppose $g(n)$ is a polynomial, i.e.

$$g(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

Provided that the leading coefficient $a_k > 0$, then the set $O(g)$ is the same set as $O(n^k)$

Proof

The proof is in two parts. First, suppose $f \in O(g)$, then by Definition 5.4.6 there exists K and C such that for $n > K$:

$$f(n) \leq Cg(n)$$

Provided $n \geq 1$, and $i > j$ we have that $n^i \geq n^j$. So take $K' = \max(K, 1)$, and $C' = \sum_{i=0}^{k} C|a_i|$, then $f(n) \leq C'n^k$, and thus $f \in O(n^k)$.

Now suppose that $f \in O(n^k)$. Then there are K and C such that for $n > k$ we have:

$$f(n) \leq Cn^k$$

Choose $K' = \max(K, 1)$ and $C' = C/a_k$ (provided $a_k > 0$), then

$$f(n) \leq C'g(n)$$

As required.

□

Next we show that the complexity of polynomial classes is strict. That is, for example,

$$O(n) \subsetneq O(n^2) \subsetneq O(n^3) \subsetneq \dots$$

which should not be surprising.

Lemma 5.5.4

$$O(n^m) \subsetneq O(n^{m+1})$$

Proof

Suppose $f \in O(n^m)$. Then there exists k, C such that for all $n > k$:

$$f(n) \leq Cn^m \leq Cn^{m+1}$$

as $n^m \leq n^{m+1}$ and therefore $f \in O(n^{m+1})$. However, we can show that $g(n) = n^{m+1}$ is not in $O(n^m)$ as follows. Let us suppose that it is, then there exists k, C such that for all $n > k$:

$$g(n) = n^{m+1} \leq Cn^m$$

however if we let $n = C + 1$ (we can control n as this must hold for all n) we get

$$(C+1)^{m+1} = (C+1)^m(C+1) \leq (C+1)^m C$$

and dividing through by $(C+1)^m$ gives us

$$C + 1 \leq C$$

which is clearly contradictory and our assumption that $n^{m+1} \in O(n^m)$ is invalid.

□

Finally, we consider $O(\log(n))$. Notice that Big-Oh notation allows us to ignore the base of the logarithm as we have

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$$

and therefore $\log_a(b)$ (which will be constant) can be taken as a constant scaling factor. To see why we are particularly interested in $O(\log(n))$ see the later section with examples using arrays.

Lemma 5.5.5

$$O(1) \subsetneq O(\log n) \subsetneq O(n)$$

(In Computer Science we usually take $\log(n)$ *to be* $\log_2(n)$*.)*

Exercise 5.8

Prove Lemma 5.5.5. **Hint**: The inverse to $\log_2(x)$ is 2^x.

5.6 Some Exercises

5.6.1 Thinking about Functions

In this course I want you to become familiar with how functions grown and the idea that even though one function may eventually dominate another function, this may not be practically relevant. For example n^2 eventually dominates $1000000n$ but only when n is bigger than 1 million so if the input n is unlikely to be this large then n^2 is a better solution.

To help understand this point we consider the following exercise that requires you to compare functions. You should try to complete this exercise **without a calculator** at first as it is good to get a feel for the comparative size of functions without needing to calculate their exact value (you can check your answers with a calculator). To help with this you should remember that

2	$=$	2^1	$\log_2(2)$	$=$	1
4	$=$	2^2	$\log_2(4)$	$=$	2
$\ldots$		$\ldots$	$\ldots$		$\ldots$
128	$=$	2^7	$\log_2(128)$	$=$	7
256	$=$	2^8	$\log_2(256)$	$=$	8
512	$=$	2^9	$\log_2(512)$	$=$	9
1024	$=$	2^{10}	$\log_2(1024)$	$=$	10

and therefore, for example, $\log_2(500) < 9$ as $\log_2(a) < \log_2(a+b)$ for $b > 0$.

Exercise 5.9

Complete the following table by placing $<, =, >$ in each box to indicate the relationship between either $f(n)$ and $g(n)$ for some n or the relationship between $O(f)$ and $O(g)$ where we abuse notation and write $<$ for strict subset etc.

		$f(n)$? $g(n)$		
$f(n)$	$g(n)$	$n=5$	$n=250$	$O(f)$? $O(g)$
n	$2n$	$<$	$<$	$=$
n	n^2			
1	1000			
n^2+n	$100n$			
$\frac{n^5}{2n^2}$	$5n^3$			
$\log_2(n)$	$\frac{n}{25}$			
$n-2$	$\log_2(\log_2(n))$			
n^{2^n}	2^{n^2}			

I have filled in the first row already. For both $n = 5$ and $n = 250$ the function $g(n) = 2n$ will be bigger than $f(n) = n$. However, they are both in the same Big-Oh class as $O(n) = O(2n)$ due to the constant scaling factor observation.

5.6.2 Looking at Programs

Now that we have the Big-Oh notation we should apply it to some programs. In the `while` language it can be relatively easy to spot when things are getting complex as we can only use loops to multiply

the operations we perform and we require nested loops to obtain large complexities. In the real world you will often be dealing with *recursive* functions, in such cases things can become less clear.

However, note that loops do not necessarily imply certain complexities. Consider the following program:

```
sum := 0; n := 10000;
while(n > 0) do (sum := sum+n; n:= n-1)
```

This program computes the sum of the numbers up to 10000 and contains a loop that iterates 10000 times. However, its complexity is $O(1)$ as its running time is independent of any input. This demonstrates a limitation of asymptotic complexity and a warning about counting loops to guess complexity.

An example of a program with non-constant complexity is our division program (repeated again for the forgetful).

```
r := x; d := 0; while y ≤ r do d := d+1; r := r-y;
```

As we saw in Section 5.3 the number of iterations required is linear in $\frac{x}{y}$, so the complexity of this program is $O(n)$, given $n = \frac{x}{y}$.

As an aside, we need to be careful here and the topic of complexity analysis for multiple variables is... complex[1]. In general, people tend to also be sloppy about this.

Exercise 5.10

Write `while` programs with the complexities $O(1)$, $O(n)$, $O(n^2)$, and $O(\log_2(n))$.

Exercise 5.11

What are the Big-Oh complexities of the following programs:

- The square-root program shown in Example 1.2.2
- The primes program shown in Example 1.2.5
- The factorial program you wrote in answer to Exercise 1.10
- The log-base-two program you wrote in answer to Exercise 1.13

A far less obvious program for complexity analysis is our gcd program from Example 1.2.4:

```
while (x != y) do (
  if (x>y) then
    x := x-y
  else
    y := y-x
)
z:=x
```

We need to identify the worst case inputs. It turns out that this is when x and y are consecutive Fibonacci numbers. A full description of the complexity in this case is beyond the scope of this course. But I encourage the interested reader to search for the answer online.

[1] e.g. see *On Asymptotic Notation with Multiple Variables by Rodney R. Howell*

5.7 Further Examples Using Arrays

In the real world programs often contain arrays so it is useful to briefly consider some examples of complexity analysis with such programs. Additionally, I want to give an example of a practically interesting program with logarithmic complexity and to do this I need to use arrays.

Here I consider programs in the $\texttt{while}_{\texttt{Arr}}$ language introduced in Section 1.4 and reuse some of the programs introduced there. Note that this language is not examinable and the ideas in this section are here to give you a more realistic feel for the use of complexity analysis.

Recall the program introduced in Example 1.4.4 for computing the dot-production program for n-dimensional vectors:

```
dot := 0; for i = 1 to n do dot := dot + a[i] ⋆ b[i]
```

This program is $O(n)$ for vectors of dimension n as for each element in the vector we perform a constant number of operations (which can be hidden in the constant scaling factor).

As a more involved example, consider the problem of writing a telephone directory look-up program. Suppose that we have two arrays, each of dimension n. The first, called e, holds the employee number, and the second, called t, holds the corresponding employee's telephone number. Further suppose that the employee numbers in array e are held in order, *i.e.*

$$i < j \quad \text{implies} \quad e[i] < e[j]$$

One way to write the telephone directory look-up program to find the telephone number of employee with number ex is as follows:

```
tel := −1;
for i = 1 to n do
   if e[i] = ex then tel = t[i] else skip
```

This has the effect of looking for employee ex's telephone number starting at the beginning of the directory, and assigning the variable *tel* the telephone number if an entry is found. What is the complexity of this program? We do a constant amount of work for each element in the array so it is $O(n)$.

Exercise 5.12

> Modify the linear search algorithm above to give a best case complexity of $O(1)$. What is the best case for linear search?

But this is a poor algorithm. Instead we should start in the middle and do a "binary chop" based on the employee number at the middle of the directory.

```
lo := 0;
hi := n;
mid := n/2;
tel := −1;
while ¬(e[mid] = ex) ∧ ¬(lo = hi) do
   (if e[mid] ≤ ex then lo := mid
                   else hi := mid; mid := (hi−lo)/2;)
if e[mid] = ex then tel := t[mid] else tel := −1
```

The key idea in this algorithm is that the area being searched is halved each time around the loop. This is the binary search program shown previously in Example 1.4.2 (see the comment about division there).

What is the complexity here? Every time we go around the loop we effectively *halve* the part of the array we are looking at. Are we familiar with a function that iteratively halves its input? The complexity of this program is $O(\log_2(n))$.

Exercise 5.13

Compute the complexity of the sorting algorithm in Example 1.4.3 and for the program you wrote in Exercise 1.26. Research the best algorithm for sorting in terms of complexity.

5.8 Lower Bounds

There is a counterpart to the 'Big-Oh' notation: the 'Big-Omega' notation. The idea here is that by saying that $f \in \Omega(g)$ we are saying that *eventually* the function f *eventually dominates* g; *i.e.* g is a *lower bound* to f.

Formally, we define Ω as follows:

Definition 5.8.1

If $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $f \in \Omega(g)$, then there exists $k \in \mathbb{N}$ and $C > 0$ such that for all $n > k$:

$$Cg(n) \leq f(n)$$

The important point to remember is: *algorithms* have an upper bound on their execution time ("quicksort is average-case $O(n \log n)$")[2]; and *problems* have a lower bound ("sorting *must* be $\Omega(n)$ since we must compare each element at least once").

Important
This distinction is important. Saying that a problem has a lower bound means that any solution to that problem must be at least that complex. This is a far stronger argument than the arguments we have been making so far, which have been about particular algorithms/programs.

Because these differ (by a factor of $\log n$) there is a chance that we will find a better algorithm for sorting. In fact the best current sorting algorithm is also $O(n \log n)$. The situation where the best algorithm is worse than the problems lower bound is very common, indeed it is unusual for there to be no difference. The existence of a difference is called an "algorithmic gap".

Definition 5.8.2

We say that there is an *algorthmic gap* if the problem lower bound complexity is less than the current best algorithm's upper bound complexity.

[2]If you haven't met quicksort then I strongly suggest looking it up and read about its complexity.

One example of a problem for which we *know* that there is no algorithmic gap is calculating the maximum of a non-empty array of numbers of size n.

```
max := 0;
for i := 1 to n do
    if max ≤ a[i] then max := a[i] else skip
```

Since we have to inspect each value in the array, the lower bound on the calculation is at least $\Omega(n)$. However, we also know that the above algorithm is $O(n)$. And thus there is no algorithmic gap. Whenever this happens, we write $\Theta(n)$.

Formally, we can define Θ as follows:

Definition 5.8.3

We write $f \in \Theta(g)$, if, and only if,

$$f \in \Omega(g) \wedge f \in O(g)$$

Exercise 5.14

Show that

$$f \in \Theta(f)$$

Exercise 5.15

Prove that if $f \in \Theta(g)$, then there exist $A,\ B \in \mathbb{R}_{>0}$ and $k \in \mathbb{N}$ such that whenever $n > k$:

$$A.g(n) \leq f(n) \leq B.g(n),$$

and *vice versa*. This is an alternative definition of Θ.

5.9 Space Complexity

So far we have only discussed *time complexity*. It can also be important that we understand the *space complexity* of a program i.e. how much memory it requires to run. As with time complexity we need to be careful about what we are counting such that it makes sense. For example, it does not make sense to say that the number 2^{30} takes more space to store than the number 1 as in most machines both numbers will be stored in a 32-bit word.

For most of the programs we have considered in the `while` language the space needed is constant as we have a fixed number of variables. In other cases (programs using arrays) the space complexity consists of the space required to store the input and output[3]. But programs that require us to store intermediate results can have non-trivial space complexity. For example, here is a (silly) program that has $O(n)$ space complexity for computing the sum of numbers from 1 to n.

```
for i := 1 to n do (a[0] := i);
sum := 0
for i := 1 to n do (sum := sum + a[i])
```

[3]Sometimes the input is not considered part of the space required for a program.

The program first creates an array containing the numbers from 1 to n and then sums these up.

These days we are often less concerned with space complexity as time complexity for two reasons:

- Space requirements are often less dependent on the size of the input and in many cases are fixed; and
- We have more space than time

However, it is still useful to think about the space complexity of an algorithm. The field of computational complexity theory also spends a reasonable amount of time considering the relationship between time and space complexity, with some interesting results. One of these results is that space complexity cannot be larger than time complexity if we assume that everything that is written to or read from the used space takes some time to do so.

As a final thought, we can always exchange time complexity for space complexity by storing precomputed results. Assuming the time taken to perform lookup in a lookup table is negligible (not the case for large tables) we can turn a time complexity $O(f(n))$ function into time complexity $O(1)$ for $n < k$ where we store the first k values of $f(n)$ in a lookup table. But the space complexity of this lookup will be $O(k)$. In some cases this trade-off may be worthwhile. For a famous example of this look at the idea of Rainbow tables[4] for reversing cryptographic hash functions. And see

`https://en.wikipedia.org/wiki/Space-time_tradeoff`

for a further discussion of this topic.

5.10 Complexity Classes

In this short section I want to explain the meaning of a very important question in theoretical computer science: $P = NP$? We don't know the answer to this question (mostly people think the answer is no).

The first thing to do is introduce the idea of a *complexity class*. This is just a set of functions (problems) with the same asymptotic complexity (in some resource). So, by this definition, $O(n)$ and $O(n^2)$ are both complexity classes. However, it is useful to have a more general notion than the one introduced by Big-Oh notation.

We define the complexity class P as the set of all polynomial-time functions as follows:

$$P = \{f \mid \exists k : f \in O(n^k)\} = \bigcup_k O(n^k)$$

i.e. a function f is in P if there is some polynomial p such that $f \in O(p)$. There are obvious functions that are not in P, for example $2^n \notin P$.

We usually call problems in P *tractable* and problems not in P *intractable*. Intractability means not practically solvable in a reasonable amount of time. Take a program with complexity 2^n as an example, if $n = 100$ and a machine performs 10^{12} operations per second then the program would run for 4×10^{10} years, which is the same order of magnitude as the age of the universe. This is too long to wait.

Another class of problems that is useful to think about are those where we can check that the solution is correct in polynomial time. This formulation doesn't fit into the notation we have so far so we will keep the idea abstract. The idea is that if we can check that the solution to a problem is correct

[4] `https://en.wikipedia.org/wiki/Rainbow_table`

in polynomial time then a *non-deterministic* machine could compute that function in polynomial time, as it could try all possible solutions non-deterministically and check if each solution is correct, all in polynomial time. This class of problems is the *Non-deterministic Polynomial* class, written NP. Clearly all problems in P are also in NP but it is not clear that the other direction holds.

There are many problems we want to solve that are in P, but equally there are many in NP. Here are some examples:

- Interesting problems in P:
  - Calculating the greatest common divisor (our gcd problem)
  - Linear programming
  - Determining if a number is prime
  - Context Free Grammar membership
  - Horn satisfiability
- Interesting problems in NP:
  - Boolean satisfiability
  - Factorisation into prime factors
  - Chinese postman problem
  - Knapsack problem
  - Graph colouring
  - See also Karp's 21 NP-complete problems

Finally, a common method for showing that a problem is in a particular class is via *reductions*. If we have a function $f \in C$ for complexity class C if we can find another function $g \in C$ such that $f \circ g = h$ then we know that $h \in C$. More concretely, *satisfiability* for propositional logic (where clauses have at least 3 literals) is known to be in NP. We can reduce the *graph colouring* problem (for > 2 colours) to the satisfiability problem by encoding the problem of colouring a graph as a propositional satisfiability problem. As solving the satisfiability problem also solves our graph colouring problem, and the reduction can be done in polynomial time, we know that graph colouring is also in NP. If you want to learn more about these topics then look at the notions of NP-hardness and NP-completeness.

5.11 Practical Complexity Analysis

Important
This section contains material introduced by David Lester based on his experience with SpiNNaker. I have left it in as it gives a good example of a low-level practical application of complexity analysis. It is not examinable.

One issue that has not been addressed when "counting operations" is that of counting the number of instructions which are involved in just moving data about. If you look at the code produced by

practically any compiler you will find a great deal of memory/register and register/register transfers going on. The only way to count *these* operations – which often have an important bearing on how long a program will take to run – is to compile the code, and only then count how many copy operations occur.

To give an idea of what can happen, I will show what happens when the division program of Exercise 1.2.1 is compiled for SpiNNaker. If the C translation of the program is compiled with `gcc` using the command line:

```
arm-none-eabi-gcc -march=armv5te -c div.c
```

and then disassembled with `objdump` using the command line:

```
arm-none-eabi-objdump -Dax div.o
```

we obtain the following disassembly:

```
00000000 <division>:
   0: e92d09f0  push {r4, r5, r6, r7, r8, fp}
   4: e28db014  add fp, sp, #20
   8: e24dd008  sub sp, sp, #8
   c: e50b0018  str r0, [fp, #-24]
  10: e50b101c  str r1, [fp, #-28]
  14: e51b3018  ldr r3, [fp, #-24]
  18: e51b201c  ldr r2, [fp, #-28]
  1c: e0030392  mul r3, r2, r3
  20: e3530000  cmp r3, #0
  24: ba000001  blt 30 <division+0x30>
  28: e3a03001  mov r3, #1
  2c: ea000000  b 34 <division+0x34>
  30: e3e03000  mvn r3, #0
  34: e1a08003  mov r8, r3
  38: e51b4018  ldr r4, [fp, #-24]
  3c: e3a05000  mov r5, #0
  40: ea000002  b 50 <division+0x50>
  44: e2855001  add r5, r5, #1
  48: e51b301c  ldr r3, [fp, #-28]
  4c: e0634004  rsb r4, r3, r4
  50: e51b301c  ldr r3, [fp, #-28]
  54: e1530004  cmp r3, r4
  58: daffffff9  ble 44 <division+0x44>
  5c: e0050598  mul r5, r8, r5
  60: e3580001  cmp r8, #1
  64: 0a000001  beq 70 <division+0x70>
  68: e51b301c  ldr r3, [fp, #-28]
  6c: e0644003  rsb r4, r4, r3
  70: e1a06005  mov r6, r5
  74: e1a07004  mov r7, r4
  78: e1a02006  mov r2, r6
```

```
  7c: e1a03007  mov r3, r7
  80: e1a00002  mov r0, r2
  84: e1a01003  mov r1, r3
  88: e24bd014  sub sp, fp, #20
  8c: e8bd09f0  pop {r4, r5, r6, r7, r8, fp}
  90: e12fff1e  bx lr
```

The extra magic required to obtain half-way reasonable code is to use the following extra flag in the command line:

```
arm-none-eabi-gcc -march=armv5te -Ofast -c div.c
```

Now the code becomes:

```
00000000 <division>:
   0: e0130091  muls r3, r1, r0
   4: 43e02000  mvnmi r2, #0
   8: 53a02001  movpl r2, #1
   c: e1500001  cmp r0, r1
  10: e1a03000  mov r3, r0
  14: e3a00000  mov r0, #0
  18: ba000004  blt 30 <division+0x30>
  1c: e0613003  rsb r3, r1, r3
  20: e1510003  cmp r1, r3
  24: e2800001  add r0, r0, #1
  28: daffffb  ble 1c <division+0x1c>
  2c: e0000092  mul r0, r2, r0
  30: e3520001  cmp r2, #1
  34: 10633001  rsbne r3, r3, r1
  38: e1a01003  mov r1, r3
  3c: e12fff1e  bx lr
```

Not only is this code faster, but at just 16 instructions instead of the original 37 it is also much more compact.

Important To ensure that *all* operations impacting on performance are fully taken into account you need to *compile* and then *disassemble* the program. There is no way to just guess how a compiler will compile your code.

Important Do *not* obsess about these issues too early. Ideally, you will have demonstrated that your code is correct, ensured that the algorithms chosen are the best, and profiled you code to find the key inner loops. Only then should you inspect the code generated for these key inner loops to check that they are acceptably efficient.