Logic for Computer Science

Wouter Swierstra

Utrecht University

Contents

About these notes

These course notes are intended for the undergraduate course *Logic for Computer Science* that I teach at the University of Utrecht. In the first part of the course, I cover the first half of the book *Modelling Computing Systems* (Moller and Struth 2013); in these lecture notes I assume students have some familiarity with basic logic, (structural) induction, and a bit of programming experience.

The contents of these notes are cobbled together from several sources, including *Semantics with Applications* (Nielson and Nielson 2007), Frank Pfenning's (2004) lecture notes on natural deduction, and Gabriele Keller and Liam O'Connor-Davis's (2019) lecture notes on inference rules and rule induction.

Wouter Swierstra

October 2020

1 Inductively defined relations

Throughout the lectures so far, we have seen various *inductive definitions*. For example, we can define the set all binary words $\mathbf{W}$ using the following BNF equation:

$$w \quad ::= \quad \varepsilon \mid 0w \mid 1w$$

That is, every binary word is either empty (ε), or it starts with either a 1 or a 0, followed by some shorter word. We can then define *inductive functions* over such sets, by introducing cases for each alternative. For example, the function `length` computes the length of a given binary word:

```
length : W → N
length(ε)  = 0
length(0w) = 1 + length(w)
length(1w) = 1 + length(w)
```

In this style, we have seen numerous examples of inductively defined sets, including the natural numbers, powersets of a given finite set, binary trees, and propositional logic formulas. We can define each of these sets using BNF; subsequently we can define functions over such sets using induction.

Yet we have not yet encountered many inductively defined *relations*. In this section, we will try to define a relation $w \leqslant w'$ that states that w is a prefix of w'. Before trying to define this relation, consider the following examples:

- 0 is a prefix of 001, or written differently $0 \leqslant 001$;
- $00 \leqslant 001$ and $001 \leqslant 001$ also hold;
- but 01 is *not* a prefix of 001;
- finally, ε is trivially a prefix of 001.

How can we give an *inductive* definition of this prefix relation? One way to characterise the relation is with the following three clauses:

- for all $w \in \mathbf{W}$, $\varepsilon \leqslant w$;
- if $w \leqslant w'$, then $0w \leqslant 0w'$;
- if $w \leqslant w'$, then $1w \leqslant 1w'$;

This is a bit clunky – how can we check whether or not a given 'proof' that $w \leqslant w'$ is constructed using these rules or not? If we need to define more complex inductive relation, we might need many such rules. There is a clear need for more precise notation: a good analogy is the early definitions of inductive sets that we saw, before introducing BNF. Is there not a better notation for inductively defined relations?

Inference rules

In this section, we will introduce the *inference rule* notation for inductively defined relations. This notation plays a central role in our definitions of proofs and programming logic in the remaining chapters.

Rather than the bullet points we saw on the previous page, we can also define inductive relations by means of *inference rules*:

$$\frac{}{\varepsilon \leqslant w}\ \text{Base}$$

$$\frac{w \leqslant w'}{0w \leqslant 0w'}\ \text{Step0}$$

$$\frac{w \leqslant w'}{1w \leqslant 1w'}\ \text{Step1}$$

Here we have three *inference rules*. Each rule consists of three parts: the name, premises and conclusion. These three rules are named Base, Step0 and Step1; each rule corresponds to one of the bullet points we saw previously. Together these rules define a binary relation on binary words $(\leqslant) \subseteq \mathbf{W} \times \mathbf{W}$.

The part above the horizontal line in each rule are the *premises* - these are the statements that you must establish in order to use this rule. The statement under the horizontal line is the *conclusion* that you can draw once you have established the premises hold. A rule without (recursive) premises is sometimes called an *axiom*.

These inference rules state that there are three ways to prove that $w \leqslant w'$ for a given pair of words w and w':

- for any binary word w, we can use the Base rule to establish $\varepsilon \leqslant w$;
- for any binary words w and w', if we have already established $w \leqslant w'$ then we can use the Step0 rule to show $0w \leqslant 0w'$;

- similarly, for any binary words w and w', if we have already established $w \leqslant w'$ then we can use the Step1 rule to show $1w \leqslant 1w'$;

By repeatedly applying these rules, we can write larger proofs. For example, to give a formal proof that $01 \leqslant 010$ we can use all three rules in the following fashion:

$$\dfrac{\dfrac{\dfrac{}{\varepsilon \leqslant 0}\,\text{Base}}{1 \leqslant 10}\,\text{Step1}}{01 \leqslant 010}\,\text{Step0}$$

Such a proof is sometimes referred to a as *derivation*. You may want to think of each inference rules describing a different building block that can use to assemble more complex derivations.

Although the derivation above happens to use all three rules, other derivations may use some rules more than once or not at all.

Exercise 1.1

Give a derivation of $00 \leqslant 001$. How often does your derivation use each inference rule?

We can read each rule and derivation from both top-down and bottom-up. Read bottom-up, a derivation establishes that a given relation holds, repeatedly breaking the statement into smaller pieces. Read top-down, a derivation starts with the axioms of a relation; by applying other inference rules, we then establish other statements also hold, until we reach the end of the derivation.

Example: Palindromes

A word over an alphabet Σ is called a **palindrome** if it reads the same backward as forward. For example, 'racecar', 'radar', and 'madam' are all palindromes.

We can define the set of all palindromes $\mathrm{P} \subseteq \Sigma^\star$ as unary relation using three inference rules. Whenever we can establish isPalindrome(w) using these rules, we claim that w must be a palindrome:

$$\frac{}{\text{isPalindrome}(\varepsilon)}\,\text{Empty}$$

$$\frac{a \in \Sigma}{\text{isPalindrome}(a)}\,\text{Single}$$

$$\frac{a \in \Sigma \qquad \text{isPalindrome}(w)}{\text{isPalindrome}(a\,w\,a)}\,\text{Step}$$

Let's go over these rules one by one. The Empty and Single rules say that ε is a palindrome and that for each letter in a in our alphabet Σ, the word a is a palidrome. The more interesting rule, Step, states that if we can establish w is a palindrome, then so is the larger word $a\ w\ a$, that adds the letter $a \in \Sigma$ to the front and back of w.

The Step rule is a bit more interesting than the other rules we have seen so far: it has *more than one premise*. That is, to use the Step rule, we need to establish that **both** $a \in \Sigma$ and isPalindrome(w).

Exercise 1.2

Give a derivation showing isPalindrome(00) and isPalindrome(101).

Exercise 1.3

There are two axioms that can be used to prove isPalindrome(w). Given a derivation of isPalindrome(w), can you predict with which axiom was used to start the derivation?

Dyck words

In the examples we have seen so far, there is typically only ever one inference rule that is applicable. This is not always the case however. To give a convincing example of a more complicated set of inference rules, however, requires a bit of work.

Consider the alphabet $\Sigma = \{\,[\,,\,]\,\}$, that the set containing the open bracket character '[' and closing bracket character ']'. Now consider the words over this alphabet, Σ^* ,such as:

- $[] \in \Sigma^\star$
- $[][] \in \Sigma^\star$
- $[[][]] \in \Sigma^\star$
- $]] \in \Sigma^\star$

Some of these words correspond to a *balanced* set of brackets, where each closing bracket is preceded by a matching open bracket and each open bracket is closed before the end of the word. This subset of all words over Σ is sometimes referred to as the *Dyck language*. When studying programming languages, we typically want to work with sequences of parentheses or curly braces that are well balanced—how can we characterise the set of all balanced words?

Before trying to define the set of all balanced words, we can consider a few examples. For instance, $[[][]]$ is balanced; whereas $[][$ and $]][]$ are both not balanced. We would like to define a unary relation on $\Sigma^\star$ that exactly characterises the balanced words. One way to do so is by defining the following three inference rules:

$$\frac{}{\text{isBalanced}(\varepsilon)}\ \text{Empty}$$

$$\frac{\text{isBalanced}(w)}{\text{isBalanced}([\,w\,])}\ \text{Bracket}$$

$$\frac{\text{isBalanced}(w) \qquad \text{isBalanced}(w')}{\text{isBalanced}(ww')}\ \text{Append}$$

Once again, let's go over the rules one by one. There is a single axiom, Empty, that states that the empty word ε is balanced. The two other rules are a bit more complex.

The Bracket rule states that any balanced word can be enclosed in brackets and remain balanced. The final rule, Append, states that if two words w and w' are balanced, then so is $w\,w'$, that is, the word formed by concatenating w and w'. Using these rules, we can prove that $[[]\,[]]$ is balanced:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{isBalanced}(\varepsilon)}\ \text{Bracket}}{\text{isBalanced}([])}\ \text{Bracket} \qquad \dfrac{\dfrac{}{\text{isBalanced}(\varepsilon)}\ \text{Bracket}}{\text{isBalanced}([])}\ \text{Bracket}}{\text{isBalanced}([]\,[])}\ \text{Append}}{\text{isBalanced}([[]\,[]])}\ \text{Bracket}$$

Exercise 1.4

Which of the example words below are balanced?

1. $[] \in \Sigma^\star$
2. $][\in \Sigma^\star$
3. $[][] \in \Sigma^\star$
4. $]] \in \Sigma^\star$

If they are balanced, give a derivation. If they are not, explain why no derivation can exist.

Exercise 1.5

How many *different* derivations of isBalanced(w) are there?

Hint: recall that $\varepsilon\, w = w = w\, \varepsilon$ for all words w.

What about the isPalindrome relation? Can there be more than one different derivation that a binary word is a palindrome? Why or why not?

Amibuguity

Although we – as humans – can easily enough construct a derivation for a given Dyck word using the rules above, this is not as obvious as you may think. Consider the above derivation of isBalanced([[]]); we can see easily enough that a derivation should start by using the Bracket rule – but this isn't the only possible way to start. We could just as well have started by applying the Append rule as follows:

$$\frac{\text{isBalanced}([) \qquad \text{isBalanced}([]])}{\text{isBalanced}([[]])}\ \text{Append}$$

This derivation is unfinished–and indeed there is no way to complete it since we still need to establish isBalanced([), for which no derivation exists.

This example illustrates that *more than one* rule may be applicable to establish a certain property. Indeed, there may be more than one derivation for the same property. This is a crucial difference between inductively defined functions and inductively defined relations: where a function should produce a single result on a given input, there may be many different derivations of the same fact.

Rule induction

Why go through all this effort to define an inductive relation using inference rules? One advantage is that we can now *prove* properties of balanced words by induction over their derivation.

Theorem 1. *For every word $w \in \Sigma^\star$, if w is balanced then w has an equal number of opening and closing brackets.*

Proof If w is an arbitrary balanced word, there must be some *derivation* establishing isBalanced(w). This derivation, however, is a finite structure built from the inference rules we have given above. As a result, we can perform *induction on the derivation* and distinguish the following three cases:

- if the derivation consists of the Empty axiom, we can conclude that w must be equal to the empty word ε. As ε has an equal number of opening and closing brackets (namely zero), we are done.
- if the derivation ends with the Bracket rule, we learn that w is actually of the form $[w']$ for some other balanced word w'. Our induction hypothesis tells us that w' has an equal number of opening and closing brackets; as the Bracket rule adds one opening bracket and one closing bracket, our proof holds for our original word w.
- finally, if the derivation ends in the Append rule, we know that w can be written as $w_1 w_2$ for some pair of balanced words w_1 and w_2. By induction, we know that both w_1 and w_2 have an equal number of opening and closing brackets, hence w must also have an equal number of opening and closing brackets.

It is important to emphasise that this proof does **not** do induction on the word w itself, but rather on the *derivation* showing that w is balanced.

By making the inductive structure of the isBalanced relation explicit by means of inference rules, we can suddenly reason about all possible proofs of 'balancedness'. By contrast, we could also define an inductive function that checks if a given word is balanced or not —– but any proofs about balanced words would need to follow the inductive structure of the words themselves.

The proof technique illustrated above, using induction over a derivation, is sometimes referred to as *rule induction*. We won't perform many such proofs, but they form a crucial proof technique when studying logic and programming languages. In the MSc course on *Concepts of programming languages* you will encounter many more examples of systems of inference rules and proofs about them using rule induction.

For now, however, we will limit ourselves to studying systems of inference rules and the derivations we can write using them.

Exercises

Exercise 1.6

Define an inductive relation on the natural numbers, isEven $\subseteq \mathbb{N}$. The relation isEven(n) should hold precisely when n is an even number.

Hint: To define an inductive relation, you need to find an appropriate base case and inductive case.

Show that isEven(4) holds.

Exercise 1.7

Define a binary inductive relation on the natural numbers, $\leqslant \subseteq \mathbb{N} \times \mathbb{N}$. The relation $\mathrm{n} \leqslant \mathrm{m}$ should hold precisely when n is less than or equal to m.

Prove that $1 \leqslant 3$.

Explain how you can use rule induction to prove that this relation is reflexive and transitive.

Solutions to exercises

Exercise 1.1

$$\dfrac{\dfrac{\dfrac{}{\varepsilon \leqslant 1}\,\text{Base}}{0 \leqslant 01}\,\text{Step0}}{00 \leqslant 001}\,\text{Step0}$$

This proof uses the Step0 rule twice and the Base rule once. It doesn't the Step1 rule at all.

Exercise 1.2

$$\dfrac{\dfrac{}{\text{isPalindrome}(\varepsilon)}\,\text{Empty}}{\text{isPalindrome}(00)}\,\text{Step}$$

$$\dfrac{\dfrac{}{\text{isPalindrome}(0)}\,\text{Single}}{\text{isPalindrome}(101)}\,\text{Step}$$

Exercise 1.3

If the length of the word w is even, we can repeatedly remove two chararters until we have none left over. In the final step, we then apply the Empty rule. If the length of w is odd, however, the derivation must end using the Single rule.

Exercise 1.4

1. The word [] is balanced, as shown by the following derivation:

$$\dfrac{\text{isBalanced}(\varepsilon)}{\text{isBalanced}([])}\,\text{Bracket}$$

2. There is no derivation showing that isBalanced(][). The only rule that introduces brackets is the Bracket rule; this rule must introduces a opening bracket that is closed later. As this word starts with a closing bracket, no derivation can exist.
3. The word [][] is balanced, as shown by the following derivation:

$$\dfrac{\dfrac{\text{isBalanced}(\varepsilon)}{\text{isBalanced}([])}\,\text{Bracket} \qquad \dfrac{\text{isBalanced}(\varepsilon)}{\text{isBalanced}([])}\,\text{Bracket}}{\text{isBalanced}([][])}$$

4. The word $]]$ is not balanced. Just as we argued above, all balanced words start with an opening bracket - hence this word cannot be balanced.

Exercise 1.5

Given any derivation isBalanced(w) for some word w, we can always construct a new derivation as follows:

$$\dfrac{\text{isBalanced}(\varepsilon) \qquad \text{isBalanced}(w)}{\text{isBalanced}(w)}\,\text{Append}$$

Hence there are *infinitely* many possible derivations showing establishing isBalanced(w) for balanced words w.

The isPalindrome relation, however, is very different: there is precisely one possible rule applicable for each word w; each derivation is unique.

Exercise 1.6

The easiest way to achiev this is using the following two rules:

$$\dfrac{}{\text{isEven(Zero)}}\,\text{ZeroEven}$$

$$\dfrac{\text{isEven(n)}}{\text{isEven(Succ(Succ(n)))}}\,\text{StepEven}$$

To establish that four is even, we provide the following derivation (using arabic numerals rather than Peano natural numbers):

$$\dfrac{\dfrac{\dfrac{}{\text{isEven(0)}}\,\text{ZeroEven}}{\text{isEven(2)}}\,\text{StepEven}}{\text{isEven(4)}}\,\text{StepEven}$$

Exercise 1.7

There are a few different ways to define this relation. One way to do so is using the following two rules:

$$\frac{}{0 \leqslant n}\ \text{Base}$$

$$\frac{n \leqslant m}{\text{Succ}(n) \leqslant \text{Succ}(m)}\ \text{Step}$$

We can then provide the following derivation showing that $1 \leqslant 3$:

$$\frac{\dfrac{}{0 \leqslant 2}\ \text{Base}}{1 \leqslant 3}\ \text{Step}$$

To prove that this relation is reflexive, we need to establish that: $\forall n, n \leqslant n$. We do so by induction on n:

- if $n = 0$ we can use the Base rule to prove that $0 \leqslant 0$;
- if n = Succ k, we know from our induction hypothesis that $k \leqslant k$. By using the Step rule, we can then show that Succ $k \leqslant$ Succ k as required.

It is harder to prove that this relation is transitive. We assume that $n \leqslant m$ and $m \leqslant p$ and need to prove that $n \leqslant p$. Now we use *rule induction* to inspect our first proof:

- if $n \leqslant m$ holds because of the Base rule, we know that $n = 0$, hence we can use the Base rule to also establish that $n \leqslant p$.
- if $n \leqslant m$ holds because of the Step rule, we know that both n and m are non-zero. We therefore know that n = Succ n' and m = Succ m', for some natural numbers n' and m'. But if m is non-zero, our second assumption ($m \leqslant p$) cannot be built using the Base rule. Hence, we can conclude that p is also non-zero, i.e., p = Succ p', for some p'. By our induction hypthesis we can establish that $n' \leqslant p'$; using the Step rule, we can then show Succ $n' \leqslant$ Succ p' as required.

2 Natural deduction

So far, we have encountered propositional logic several times. In the first lecture, we defined the syntax of propositional logic informally; later, we saw how to define this syntax more formally as an inductively defined set using a BNF equation. We have defined the *semantics* of propositional logic in terms of truth tables; later, we saw how we could give an alternative semantics using proof strategies. In contrast to the *syntax* of propositional logic, however, both these approaches to semantics fail to nail down the semantics of propositional logic precisely. This chapter aims to achieve just that.

What is a proof?

We can define the set of propositional logical formulas over a set of atomic propositional variables **PV** using the following BNF equation:

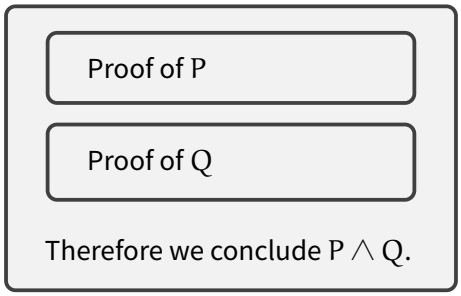
$$p, q \quad ::= \quad \text{true} \mid \text{false} \mid PV \mid \neg p \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid p \Leftrightarrow q$$

We will sometimes refer to the set of all propositional logic formulas as **PLF**.

This definition enables us to distinguish between those strings of symbols that correspond to well-formed formulas, such as $p \vee \neg q$, and those that do not, such as $\neg \vee \vee p$. But this does not yet tell us why a formula such as $p \Rightarrow p$ is always true, but $p \vee p$ is not. How can we distinguish the propositional logic formulas that are true from those that are false? Or put differently, given some formula p, *what is a proof of* p? If we define the *syntax* of propositional logic as an inductively defined set, why should be not be able to give an inductive definition of a formula's semantics?

This chapter tries to answer this question in three parts by giving a formal account of proof strategies, a formal account of truth tables, a theorem relating the two.

Proof strategies are typically given by using the following notation:



This strategy for conjunction shows how to prove $P \wedge Q$, given a proof of P and a proof of Q. We can translate this strategy to our inference rule notation directly:

$$\frac{\text{isTrue}(P) \qquad \text{isTrue}(Q)}{\text{isTrue}(P \wedge Q)} \wedge\text{-I}$$

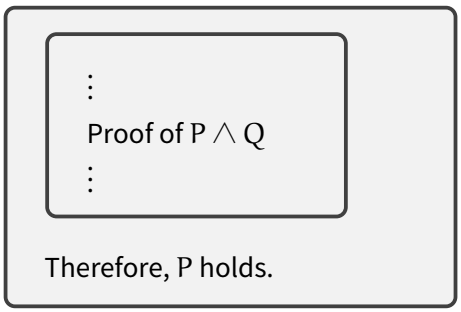
In accordance with the notation used in the proof strategies in the book, we use capital letters, P and Q for our *metavariables*that is P and Q stand for some arbitrary proposition, rather than an atomic proposition variable. It would perhaps be more precise to use p and q to avoid this confusion.

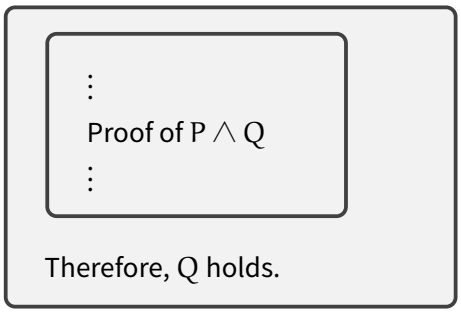
In this style, we can try to define a unary relation isTrue on the formulas of propositional logic, i.e., isTrue $\subseteq$ **PLF**; the inference rules then define the set of all possible valid proofs. Most logical textbooks do not introduce an explicit name for the relation capturing 'truthfulness' – like our isTrue relation - but rather identify a formula with its semantics, writing:

$$\frac{P \qquad Q}{P \wedge Q} \wedge\text{-I}$$

The aim of this chapter is to find a suitable collection of inference rules describing all possible proofs of a propositional logic formula. Clearly, we will need more rules than the conjunction introduction rule above. What about conjunction elimination?

There were two strategies for conjunction elimination:





What should the corresponding inference rule for conjunction elimination be? There is very little creativity necessary to come up with the following two rules:

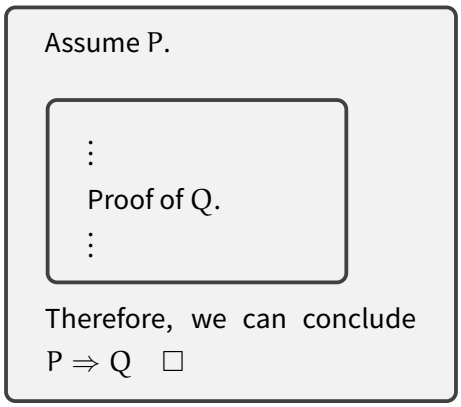
$$\frac{P \wedge Q}{P} \wedge\text{-E}_1 \qquad\qquad \frac{P \wedge Q}{Q} \wedge\text{-E}_2$$

Using these rules, we can start to write the following (incomplete) proof:

$$\dfrac{\dfrac{\dfrac{\dots}{P \wedge Q}}{Q} \wedge\text{-E}_2 \quad \dfrac{\dfrac{\dots}{P \wedge Q}}{P} \wedge\text{-E}_1}{Q \wedge P} \wedge\text{-I}$$

This derivation fragment shows how to establish $Q \wedge P$ if we already have a proof of $P \wedge Q$. The derivation is, however, incomplete—we have not yet shown that $P \wedge Q$ holds. Nonetheless, this example illustrates how different inference rules can be used to combine larger proofs.

To write a complete derivation, we might want to try proving $P \wedge Q \Rightarrow Q \wedge P$. To do so, we will also need an inference rule corresponding to the introduction rule for implication:



In the implication introduction rule, we are allowed to *assume* that P holds to give a proof of Q, and then conclude $P \Rightarrow Q$ holds. There is one important catch: we can *only* use our assumption that P holds to prove that Q holds. In particular, we cannot decide to use our proof of P elsewhere (unless we can provide a separate proof that P holds). This example makes it clear that we need to account for the assumptions that we make when writing our proofs.

Assumptions and contexts

Instead of trying to define a *unary* relation on propositions corresponding to the set of valid propositions, we instead solve a more general problem: we will define a *binary* relation that states that we can find a

proof of some propositional logic formula p using a set assumptions Γ. Before we can do so, however, we need to be precise about the structure of our assumptions Γ. One way to model these assumptions is as a list of all the predicate logic formulas that we assume to hold:

$$\Gamma \quad ::= \quad \varepsilon \mid \Gamma, p$$

This list of assumptions is sometimes referred to as a *context*; we will sometimes refer to the set of these contexts as **PLF***. In the rest of this section, we will complete the inductive definition of a relation on **PLF**$^* \times$ **PLF**. This relation, written as $\Gamma \vdash P$, states that there is a proof of the formula P from the list of assumptions Γ. When we do not need any assumptions to prove P holds, we will write $\vdash P$ rather than $\varepsilon \vdash P$.

We can rephrase our previous rules for conjunction as follows:

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \wedge I$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \wedge E_1$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \wedge E_2$$

These rules did not use or change the context Γ, so the rules remain largely unchanged.

The implication introduction rule, however, does add new assumptions:

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} I\Rightarrow$$

Here we can see how Γ may change during a derivation. To show $P \Rightarrow Q$, we add P to our list of assumptions and establish that Q holds.

Before we can complete the derivation of $\vdash P \wedge Q \Rightarrow Q \wedge P$, we need one last rule:

$$\frac{P \in \Gamma}{\Gamma \vdash P} \text{ Assumption}$$

This rule allows us to *use* an assumption P; this will be one of the few axioms of our system. If we have previously assumed P holds, for example by using the implication introduction rule, we can always conclude that P holds. This seems like a very trivial rule – but it is the cornerstone of any formal proof. A derivation of $\Gamma \vdash P$ shows how to establish P holds from the assumptions Γ; read from top-to-bottom, each derivation starts from the assumptions, using them to establish other statements, until we can conclude that P itself holds.

Using the rules we have seen so far, we can give a derivation of $\vdash P \wedge Q \Rightarrow Q \wedge P$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P \wedge Q \in P \wedge Q}\text{Assumption}}{P \wedge Q \vdash P \wedge Q}}{P \wedge Q \vdash Q}\wedge\text{-E}_2 \qquad \dfrac{\dfrac{\dfrac{}{P \wedge Q \in P \wedge Q}\text{Assumption}}{P \wedge Q \vdash P \wedge Q}}{P \wedge Q \vdash P}\wedge\text{-E}_1}{\dfrac{P \wedge Q \vdash Q \wedge P}{\vdash P \wedge Q \Rightarrow Q \wedge P}\Rightarrow\text{-I}}\wedge\text{-I}$$

Often, we will leave out the explicit premise of the assumption rule. For example, we might write the following in the proof above:

$$\dfrac{\dfrac{}{P \wedge Q \vdash P \wedge Q}}{P \wedge Q \vdash Q}\wedge\text{-E}_2$$

Here we have left out the (obvious) observation $P \wedge Q \in P \wedge Q$.

Example Using the rules we have seen so far, we can also prove that $\vdash P \Rightarrow (P \wedge P)$ as follows:

$$\dfrac{\dfrac{\dfrac{}{P \vdash P} \quad \dfrac{}{P \vdash P}}{P \vdash P \wedge P}\wedge\text{-I}}{\vdash P \Rightarrow P \wedge P}\Rightarrow\text{-I}$$

Note that we use assumption P in two separate parts of the proof; we can freely use an assumption more than once as long as it is occurs in the list of assumptions we currently have available.

Non-example Whenever we use our assumption rule, we need to ensure that the assumption we are using is available *in the current list of assumptions*. Here is an example of an incorrect derivation that does not use the inference rules we have seen so far correctly:

$$\dfrac{\dfrac{\dfrac{}{P \vdash P}}{\vdash P \Rightarrow P}\Rightarrow\text{-I} \qquad \dfrac{}{P \vdash P}}{\vdash (P \Rightarrow P) \wedge P}\wedge\text{-I}$$

Exercise 2.1

Can you explain what is wrong with the above derivation?

Non-example The statement $(P \Rightarrow P) \Rightarrow P$ is not true in general. Another way an incorrect proof may appear correct is by abusing assumptions. For example, here is an incorrect derivation showing $\vdash (P \Rightarrow P) \Rightarrow P$

$$\dfrac{\dfrac{}{P \Rightarrow P \vdash P}}{\vdash (P \Rightarrow P) \Rightarrow P} \Rightarrow -I$$

Although we *are* allowed to assume that $P \Rightarrow P$ holds, we *cannot* assume that P holds. In the lectures on proof strategies we saw a similar example that may appear correct, but subtly abused the strategies for implication. The only assumption we are allowed to make is that $P \Rightarrow P$ holds, but this is insufficient to show that P also holds.

Non-example As a final example of how derivations may be wrong, consider the following derivation:

$$\dfrac{\dfrac{}{P \vdash P}}{\vdash (P \Rightarrow P) \Rightarrow P} \Rightarrow -I$$

Here we have incorrectly used the implication introduction rule. Instead of introducing the assumption $P \Rightarrow P$, we have added the assumption P to the context. Once again, the statement $(P \Rightarrow P) \Rightarrow P$ does not hold in general and no derivation of $\vdash (P \Rightarrow P) \Rightarrow P$ exists.

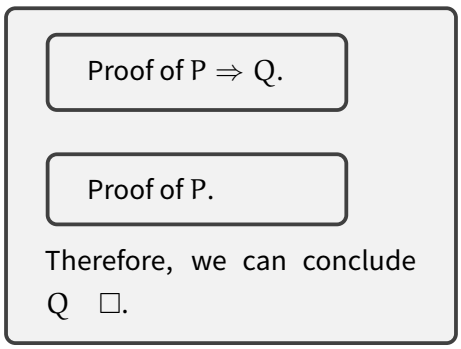
Exercise 2.2

Give a natural deduction proof of $\vdash P \Rightarrow (Q \Rightarrow (Q \wedge P))$

Natural deduction

In this section, we will give the inference rules for the remaining propositional logic operators. The resulting proof system, sometimes referred to as *natural deduction*, was originally developed by Gentzen (1935). It tries to capture the way in which mathematicians naturally do proofs in a more formal fashion. One particularly pleasant property is that we can give the rules for each propositional logic operator independently of the others. As a result, we are free to explore other logics, where we may choose different operators or different rules. Many of these rules closely mirror the proof strategies that we have seen previously – which is no coincidence: the proof strategies were an informal presentation of natural deduction that we can finally formalize here.

Implication elimination Although we have seen the rule for implication introduction, we still need to give the corresponding elimination rule. The proof strategy for implication elimination had the following form:



Once again, we can translate this to a inference rule by turning each sub-proofs into a premise:

$$\frac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q} \; E\Rightarrow$$

Exercise 2.3

Give a natural deduction proof of $\vdash (P \Rightarrow Q) \Rightarrow ((Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$

Rules for truth and falsity

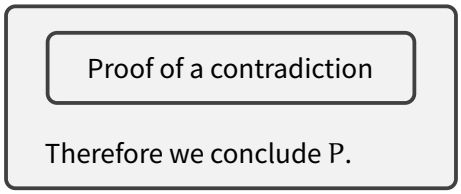
Most logic textbooks use $\top$ and $\bot$ rather than true and false respectively. We use the same notation in our inference rules. To give semantics to $\top$ and $\bot$, we need to define their elimination and introduction rules. It turns out we only need *two* rules to define both connectives.

First of all, the introduction rule for truth is trivial:

$$\frac{}{\Gamma \vdash \top} \; \top\text{-I}$$

This rule states that we can always find a proof that $\top$ holds. Unfortunately, proving true is not particularly useful. There is no introduction rule for $\bot$, as there should be no way to prove falsity.

The elimination rule for falsity follows the following proof strategy:



This strategy says that if we have managed to prove a contradiction from our assumptions somehow, we can conclude whatever we like. We can formulate this as an inference rule in the following fashion:

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P} \bot\text{-E}$$

To see how this rule is used, consider the proof showing $P \Rightarrow \bot, P \vdash Q$:

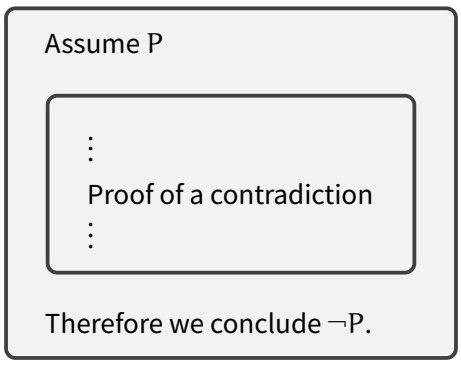
$$\dfrac{\dfrac{\overline{P \Rightarrow \bot, P \vdash P} \quad \overline{P \Rightarrow \bot, P \vdash P \Rightarrow \bot}}{P \Rightarrow \bot, P \vdash \bot} \Rightarrow\text{-E}}{P \Rightarrow \bot, P \vdash Q} \bot\text{-E}$$

This proof is quite strange: none of our assumptions mention Q, yet somehow we still manage to prove that Q holds. The reasoning is that if both P and $P \Rightarrow \bot$ hold, then we can derive a contradiction. As no such contradiction should exist, we can draw any conclusion that we want.

A similar situation also arose in the truth table for implication. Recall that the implication $P \Rightarrow Q$ always holds when P is false, regardless of Q. Similarly, if we manage to prove $\bot$ from our assumptions, we can conclude that any arbitrary Q holds.

Rules for negation

What about negation? We still need to define the introduction rule to prove $\neg P$ and elimination rule to use an assumption of the form $\neg P$. The *introduction strategy* for negation had the following form:



In words, this strategy says that if assuming P does holds leads to a contradiction, we can conclude that $\neg P$ holds. This matches our intuition that $\neg P$ behaves just like $P \Rightarrow \bot$. We can turn this proof strategy into an inference rule readily enough:

$$\frac{\Gamma, P \vdash \bot}{\Gamma \vdash \neg P}\ \neg\text{-I}$$

The intuition that $\neg P$ behaves like $P \Rightarrow \bot$ is also apparent in the *elimination* rule for negation:

$$\frac{\Gamma \vdash \neg P \qquad \Gamma \vdash P}{\Gamma \vdash \bot}\ \neg\text{-E}$$

If we can prove both P and $\neg P$ from our assumptions Γ, we have established a contradiction $\bot$; combined with the elimination rule for falsity, $\bot$-E, we saw previously we can draw whatever conclusion we like.

Exercise 2.4

Given that $P \Leftrightarrow Q$ is equivalent to $P \Rightarrow Q \wedge Q \Rightarrow P$, devise suitable introduction and elimination rules for logical equivalence, $P \Leftrightarrow Q$.

Exercise 2.5

Use the rule you proposed in the previous exercise to prove $\vdash P \wedge \top \Leftrightarrow P$.

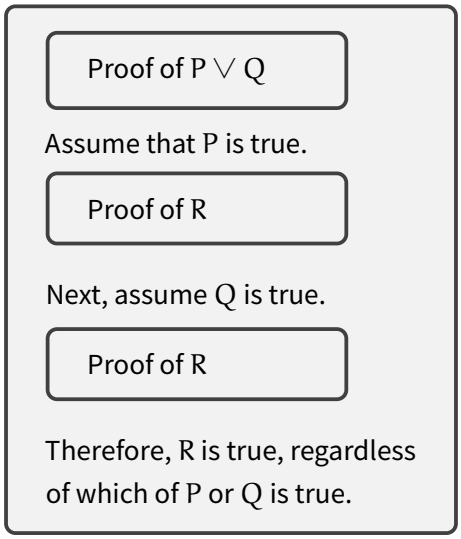
Rules for disjunction

Although we have covered the inference rules for most logical operators, we still have not yet seen the rules for disjuction. The *disjunction introduction* rules are reassuringly simple:

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ \vee\text{-I}_1$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}\ \vee\text{-I}_2$$

The rule for *disjunction elimination*, however, is a bit more tricky. The associated proof strategy was formulated as follows:



The problem with disjunction elimination is that it isn't clear how to use a proof of the form $P \vee Q$ directly. If we know that either P holds or Q holds, we cannot conclude that P must hold; nor can we conclude that Q must hold. Instead, we use the assumption $P \vee Q$ to establish some third proposition R. To show that R does hold, we need to provide two proofs: one that states that if P holds then so does R; the second states that if Q holds then so does R. Together these three ingredients guarantee that R must hold, regardless of whether P holds, Q holds, or both hold.

We can make all of this precise in the following inference rule:

$$\frac{\Gamma \vdash P \vee Q \qquad \Gamma, P \vdash R \qquad \Gamma, Q \vdash R}{\Gamma \vdash R} \vee\text{-E}$$

Example We can use the introduction and elimination rules for disjunction to prove that the disjunction operator is commutative, or more formally, that $P \vee Q \vdash Q \vee P$:

$$\dfrac{\dfrac{}{P \vee Q \vdash P \vee Q} \qquad \dfrac{\dfrac{}{P \vee Q, P \vdash P}}{P \vee Q, P \vdash Q \vee P} \vee\text{-I}_2 \qquad \dfrac{\dfrac{}{P \vee Q, Q \vdash Q}}{P \vee Q, Q \vdash Q \vee P} \vee\text{-I}_1}{P \vee Q \vdash Q \vee P} \vee\text{-E}$$

Exercise 2.6

Give a derivation showing $\vdash (P \vee \bot) \Rightarrow P$.

Reductio ad absurdum

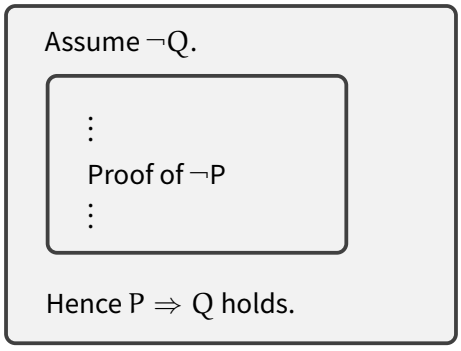
To complet our natural deduction rules for classical propositional logic, we need one final rule:

$$\frac{\Gamma, \neg P \vdash \bot}{\Gamma \vdash P}\ \text{RAA}$$

This rule, sometimes called *reductio ad absurdum*, states that if $\neg P$ leads to a contradiction, P must hold. Note that this rule is subtly different to the introduction rule for negation, that establishes $\neg P$ when assuming P leads to a contradiction. Besides our rule for using assumptions, this is the only rule that is not the introduction or elimination rule of one of the logical operators. There are valid philosophical reasons to avoid using this rule, or even reject it as a valid inference rule of our logic.

Derivable rules

This completes our presentation of natural deduction. The complete overview of all the rules is given at the end of this chapter. An eagle-eyed reader may have spotted that there are certain proof strategies that do not have a corresponding inference rule. For example, the following strategy, sometimes referred to as a *proof by contraposition*, does not have a corresponding inference rule:



Don't we need an inference rule to account for proof using this rule? We can formulate the corresponding inference rule readily enough:

$$\frac{\Gamma \vdash \neg Q \Rightarrow \neg P}{\Gamma \vdash P \Rightarrow Q}\ \text{Contraposition}$$

But adding haphazardly adding inference rules is not a good idea—we may accidentally add rules that break our logic in an unexpected way. Furthermore, the 'smaller' our collection of inference rules, the fewer cases we have to reason about when studying all possible proofs built from these rules. Instead of adding a new rule, we can instead try to *prove* this proof principle from the rules we have seen so far. To do so, we show that $\neg Q \Rightarrow \neg P \vdash P \Rightarrow Q$ holds for arbitrary propositions P and Q. In any proof using the rule for contraposition above, we can replace the rule with the derivation below:

$$\dfrac{\dfrac{\dfrac{}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash P} \quad \dfrac{\dfrac{}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash \neg Q \Rightarrow \neg P} \quad \dfrac{}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash \neg Q}}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash \neg P}}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash \bot}}{\dfrac{\neg Q \Rightarrow \neg P, P \vdash Q}{\neg Q \Rightarrow \neg P \vdash P \Rightarrow Q}}$$

Such general proof principles that can be proven from our inference rules are sometimes referred to as *derivable rules*.

Exercise 2.7

Identify each inference rule that has been used to construct the proof above.

Exercise 2.8

Use the *reductio ad absurdum* twice rule to prove that $\vdash P \vee \neg P$.

Exercise 2.9

Given a propositional logic formula P and context Γ. Is there always a derivation possible showing $\Gamma \vdash P$? If so, explain why. If not, give an example proposition that has you believe should not have a proof.

Exercise 2.10

When $\Gamma \vdash P$ holds, is this proof unique? If so, choose a propositional logic formula P and context Γ such that there are different derivations showing $\Gamma \vdash P$. If not, explain why all derivations are equal.

Semantics of propositional logic

In the previous section we introduced the system of *natural deduction*. We can use the inference rules presented therein to show how some propositional logic formula P follows from a list of assumptions Γ. Yet this is not the first semantics for propositional logic that we have encountered—in the very first lecture we showed how to prove a propositional logic formula was a tautology using *truth tables*. How are truth tables and natural deduction related?

Before we can answer this question, we need to give a more precise account of truth tables.

Booleans

Each truth table shows how a formula from propositional logic can be *evaluated* to produce a *boolean value*, namely true (T) or false (F). To make this statement precise, let's start by defining the set of boolean values using the following BNF equation:

```
Bool ::= T | F
```

We can define functions over the booleans by simply enumerating their behaviour on all possible inputs. For example, we can define the familiar functions `not` and `and` as follows:

```
not : Bool → Bool
not(T) = F
not(F) = T

and : Bool × Bool → Bool
and(F,F) = F
and(F,T) = F
and(T,F) = F
and(T,T) = T
```

These functions should be familiar to anyone who has programmed with boolean values.

Exercise 2.11

Define `or : Bool × Bool → Bool` and `implies : Bool × Bool → Bool` in the same fashion as the definitions of `not` and `and` above.

When we fill out a truth table for some propositional formula p, we show how each choice of atomic propositional variables of p results in a boolean value for the entire formula. For instance, consider the following truth table, corresponding to the formula $\neg(P \vee Q) \Rightarrow \neg P \wedge \neg Q$.

P	Q	¬	(P	∨	Q)	⇒	(¬P	∧	¬Q)
F	F	T	F	F	F	**T**	T	T	T
F	T	F	F	T	T	**T**	T	F	F
T	F	F	T	T	F	**T**	F	F	T
T	T	F	T	T	T	**T**	F	F	F

For each possible boolean value that we might associate with the atomic propositional variables P and Q, we can consult the corresponding row of the truth table to determine the value of the entire propositional formula $\neg(P \vee Q) \Rightarrow \neg P \wedge \neg Q$.

How can we describe these truth tables in terms of the familiar mathematical constructions on sets and functions that we have seen so far in this course?

Before we do so, we need to introduce the auxiliary notion of *truth assignment*. A *truth assignment* consists of a function ν : **PV** $\rightarrow$ **Bool**, mapping *atomic propositional variables* to a Boolean value. For example, the third row from the truth table above, where P is T and Q is F, corresponds to the following truth assignment:

$$\begin{aligned} \nu &: \textbf{PV} \rightarrow \textbf{Bool} \\ \nu(P) &= T \\ \nu(Q) &= F \end{aligned}$$

Using the notion of truth assignment, we will show how to assign semantics to an entire propositional logic *formula*.

Given any truth assignment ν and propositional logic formula p, we can compute the boolean truth value of a p. To do so, recall that we defined the inductive set of all propositional logic formulas using the following BNF equation:

$$p, q \quad ::= \quad \text{true} \mid \text{false} \mid PV \mid \neg p \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid p \Leftrightarrow q$$

We would like to show how to assign semantics to these formulas, that is, to define a function that maps a truth assignment to the boolean value associated with the entire formula:

$$\text{semantics} : \textbf{PLF} \times (\textbf{PV} \rightarrow \textbf{Bool}) \rightarrow \textbf{Bool}$$

Given a formula p and truth assignment ν, we compute the boolean value that states whether p holds under this truth assignment or not. Traditionally, this function is written as 'operator' using the notation $[\![p]\!](\nu)$ rather than $\text{semantics}(p, \nu)$; despite this potentially confusing notation, the intention should be clear: define a function by induction on the propositional logic formula that determines whether the formula is true or false, given a certain truth assignment.

This function is entirely straightforward to define:

$$
\begin{aligned}
[\![\mathsf{true}]\!](\nu) &= \mathsf{T} \\
[\![\mathsf{false}]\!](\nu) &= \mathsf{F} \\
[\![\mathsf{P}]\!](\nu) &= \nu(\mathsf{P}) \\
[\![\neg \mathrm{p}]\!](\nu) &= \mathsf{not}([\![\mathrm{p}]\!](\nu)) \\
[\![\mathrm{p} \vee \mathrm{q}]\!](\nu) &= \mathsf{or}([\![\mathrm{p}]\!](\nu), [\![\mathrm{q}]\!](\nu)) \\
[\![\mathrm{p} \wedge \mathrm{q}]\!](\nu) &= \mathsf{and}([\![\mathrm{p}]\!](\nu), [\![\mathrm{q}]\!](\nu)) \\
[\![\mathrm{p} \Rightarrow \mathrm{q}]\!](\nu) &= \mathsf{implies}([\![\mathrm{p}]\!](\nu), [\![\mathrm{q}]\!](\nu))
\end{aligned}
$$

This function is defined by induction over the structure of propositional logic formulas. In each case, we map the formula to a boolean value. In the base cases, true and false, we can immediately return the corresponding boolean value, T and F, respectively. In the cases for the operators from propositional logic, such as negation, disjunction, conjunction, and implication, we can compute the boolean value associated with the sub-formulas and combine the resulting boolean values using a suitable boolean operator, such as `not`, `or`, `and` and `implies` respectively. For example, the case for disjunction states:

$$[\![\mathrm{p} \vee \mathrm{q}]\!](\nu) = \mathsf{or}([\![\mathrm{p}]\!](\nu), [\![\mathrm{q}]\!](\nu))$$

To evaluate the disjunction $\mathrm{p} \vee \mathrm{q}$ using the truth assignment ν, we evaluate p and q to two boolean values. We combine these boolean values using the boolean `or` operator to assign a single boolean value to the entire formula.

The only remaining case is that for atomic propositional variables. When we encounter an atomic propositional variable, we also need to compute a boolean result. To do so, we can consult our truth assignment ν that maps each atomic propositional variable to its corresponding boolean value.

This completely defines the semantics of all propositional logic formulas. That is, we have defined a function that maps each propositional logic formula p into a function that, given a truth assignment for all atomic propositional variables, computes the truth value of the entire propositional logic formula p. This function is completely unsurprising — but what does this have to do with truth tables?

Truth tables and semantics

If you think back to the lectures on functions and induction, we saw how to *define* a function on a *finite* domain by listing all it output value for every possible input value.

For example, suppose I have three students in my class:

S = {Alice, Bob, Carroll }.

I can define a function mapping assigning each student a mark between one and ten, that is, marks : $\mathrm{S} \to \{1..10\}$, by simply listing the mark that each student has obtained:

$$\begin{aligned} \text{marks(Alice)} &= 8 \\ \text{marks(Bob)} &= 6 \\ \text{marks(Carroll)} &= 7 \end{aligned}$$

We sometimes refer to such a definition of a function with a finite domain as the *tabulation* of the marks function. Such a tabulation is in one-to-one correspondence with all possible functions marks : $\mathrm{S} \to \{1..10\}$.

Now consider the semantics of a given propositional logic formula $[\![\mathrm{p}]\!]$. This semantics consists of a function $(\textbf{PV} \to \textbf{Bool}) \to \textbf{Bool}$. Now note that because p only contains a finite number of atomic propositional variables, we can define this semantics by giving a *tabulation* of this semantics, i.e., giving the boolean value associated with *each possible truth assignment* – much as we defined a marks function by giving listing the mark value associated with each student. But such a listing the boolean value associated with each possible truth assignment is exactly what a truth table is!

When filling out a truth table for some propositional logic formula p, you are essentially computing the truth value of p *for all possible choice of value for the atomic variables in p*. That is, we are tabulating the function $[\![\mathrm{p}]\!]$.

This shows how truth tables are in one-to-one correspondence with the semantics of propositional logic formulas, $[\![\mathrm{p}]\!] : (\textbf{PV} \to \textbf{Bool}) \to \textbf{Bool}$.

Relating natural deduction and semantics

In the previous section, we outlined one possible semantics for propositional logic formulas as truth tables, or equivalently, as functions $(\textbf{PV} \to \textbf{Bool}) \to \textbf{Bool}$. Yet at the beginning of this chapter, we gave a very different semantics for propositional logic, namely the system of *natural deduction*, defined by a collection of inference rules. How can we relate these different semantics for propositional logic?

This is important—after all, how do we know we chose the right rules in our system of natural deduction? If there is a mistake in our rules, we might be able to prove false statements; or perhaps, we have forgotten to include certain rules, making it impossible to prove certain tautologies.

To make this precise, we need to introduce some new notation. Given a truth assignment ν we write $\nu \models \mathrm{p}$ if $[\![\mathrm{p}]\!]\nu = \mathbf{T}$. When $\nu \models \mathrm{p}$ for *each* possible truth assignment ν, we say write $\models \mathrm{p}$. In this case, we have established that p is a tautotology.

It turns out that natural deduction inference rules we have seen here satisfy two important properties:

Soundness If $\vdash \mathrm{p}$ then $\models \mathrm{p}$. In other words, if we can find a proof of p using the inference rules of natural deduction, then the truth table for p has a **T** in each row.

Completeness If $\models \mathrm{p}$ then $\vdash \mathrm{p}$. In other words, if the truth table of p has a **T** in each row, there is *some* derivation of p using the inference rules of natural deduction.

To prove soundness and completeness requires is not easy – but this can be done using the basic logical techniques that we have covered in this course. The proofs can be found in many introductory textbooks on logic (Van Dalen 1994).

The system of natural deduction that we have seen can be used to prove statements in propositional logic. This system can be extended to also handle *predicate* logic, but we will refrain from doing so in these notes.

Exercise 2.12

Which statement would you expect is easier to prove: soundness or completeness? How would you go about proving these statements?

Overview of natural deduction

Assumptions

$$\frac{P \in \Gamma}{\Gamma \vdash P}\ \text{Assumption}$$

Truth and falsity

$$\frac{}{\Gamma \vdash \top}\ \top\text{-I} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ \bot\text{-E}$$

Implication

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\ \text{I}\Rightarrow \qquad \frac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}\ \text{E}\Rightarrow$$

Conjunction

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ \wedge\text{I} \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P}\ \wedge\text{E}_1 \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q}\ \wedge\text{E}_2$$

Negation

$$\frac{\Gamma, P \vdash \bot}{\Gamma \vdash \neg P}\ \neg\text{-I} \qquad \frac{\Gamma \vdash \neg P \qquad \Gamma \vdash P}{\Gamma \vdash \bot}\ \neg\text{-E}$$

Disjunction

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ \vee\text{-I}_1 \qquad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}\ \vee\text{-I}_2 \qquad \frac{\Gamma \vdash P \vee Q \qquad \Gamma, P \vdash R \qquad \Gamma, Q \vdash R}{\Gamma \vdash R}\ \vee\text{-E}$$

Logical equivalence

$$\frac{\Gamma, P \vdash Q \qquad \Gamma, Q \vdash P}{\Gamma \vdash P \Leftrightarrow Q} \Leftrightarrow\text{-I}$$

$$\frac{\Gamma \vdash P \Leftrightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q} \Leftrightarrow\text{-E}_1$$

$$\frac{\Gamma \vdash P \Leftrightarrow Q \qquad \Gamma \vdash Q}{\Gamma \vdash P} \Leftrightarrow\text{-E}_2$$

Double negation

$$\frac{\Gamma, \neg P \vdash \bot}{\Gamma \vdash P} \text{RAA}$$

Further exercises

Exercise 2.13

Give a natural deduction proof of $Q \vdash (Q \Rightarrow R) \Rightarrow R$.

Exercise 2.14

Give a natural deduction proof of $\vdash \neg(A \wedge B) \Rightarrow (A \Rightarrow \neg B)$

Exercise 2.15

Give a natural deduction proof of $(P \wedge Q) \wedge R, S \wedge T \vdash Q \wedge S$

Exercise 2.16

Give a natural deduction proof of $\vdash (A \Rightarrow C) \wedge (B \Rightarrow \neg C) \Rightarrow \neg(A \wedge B)$

Exercise 2.17

Give a natural deduction proof of $\vdash (A \wedge B) \Rightarrow ((A \Rightarrow C) \Rightarrow \neg(B \Rightarrow \neg C))$

Exercise 2.18

Give a natural deduction proof of $\vdash A \vee B \Rightarrow B \vee A$

Exercise 2.19

Give a natural deduction proof of $\vdash \neg A \wedge \neg B \Rightarrow \neg(A \vee B)$

Exercise 2.20

Give a natural deduction proof of $\neg A \vee \neg B \vdash \neg(A \wedge B)$

Exercise 2.21

Give a natural deduction proof of $A \Leftrightarrow B \vdash (\neg A \Leftrightarrow \neg B)$

Exercise 2.22

Give a natural deduction proof of $\neg(A \Leftrightarrow \neg A)$

Exercise 2.23

Give a natural deduction proof of $A \vee B \vdash C \Rightarrow (A \vee B) \wedge C$

Exercise 2.24

Give a natural deduction proof of $(A \vee (B \wedge A)) \Rightarrow A$

Solutions to selected exercises

Exercise 2.1

The implication introduction rule only introduces the assumption P in the *current* subtree of the derivation. In particular, we cannot use the assumption P in the right-hand side of the conjunction, as we do here.

Exercise 2.2

$$\dfrac{\dfrac{\dfrac{\overline{P, Q \vdash Q} \quad \overline{P, Q \vdash P}}{P, Q \vdash Q \wedge P}\,\wedge\text{-I}}{P \vdash Q \Rightarrow (Q \wedge P)}\,\Rightarrow\text{-I}}{\vdash P \Rightarrow (Q \Rightarrow (Q \wedge P))}\,\Rightarrow\text{-I}$$

Exercise 2.3

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{P \Rightarrow Q, Q \Rightarrow R, P \vdash Q \Rightarrow R} \quad \dfrac{\overline{P \Rightarrow Q, Q \Rightarrow R, P \vdash P} \quad \overline{P \Rightarrow Q, Q \Rightarrow R, P \vdash P \Rightarrow Q}}{P \Rightarrow Q, Q \Rightarrow R, P \vdash Q}\,\Rightarrow\text{-E}}{P \Rightarrow Q, Q \Rightarrow R, P \vdash R}\,\Rightarrow\text{-E}}{P \Rightarrow Q, Q \Rightarrow R \vdash (P \Rightarrow R)}\,\Rightarrow\text{-I}}{P \Rightarrow Q \vdash (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)}\,\Rightarrow\text{-I}}{\vdash (P \Rightarrow Q) \Rightarrow ((Q \Rightarrow R) \Rightarrow (P \Rightarrow R))}\,\Rightarrow\text{-I}$$

Exercise 2.4

There are various variations of the following three rules that are all valid choices:

$$\dfrac{\Gamma, P \vdash Q \qquad \Gamma, Q \vdash P}{\Gamma \vdash P \Leftrightarrow Q}\,\Leftrightarrow\text{-I}$$

$$\dfrac{\Gamma \vdash P \Leftrightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}\,\Leftrightarrow\text{-E}_1$$

$$\dfrac{\Gamma \vdash P \Leftrightarrow Q \qquad \Gamma \vdash Q}{\Gamma \vdash P}\,\Leftrightarrow\text{-E}_2$$

Exercise 2.5

$$\dfrac{\dfrac{\dfrac{}{P \vdash P} \quad \dfrac{}{P \vdash \top}\,\top\text{-I}}{P \vdash P \wedge \top}\,\wedge\text{-I} \quad \dfrac{\dfrac{}{P \wedge \top \vdash P \wedge \top}}{P \wedge \top \vdash P}\,\wedge\text{-E}_1}{\vdash P \wedge \top \Leftrightarrow P}\,\Leftrightarrow\text{-I}$$

Exercise 2.6

$$\dfrac{\dfrac{}{P \vee \bot \vdash P \vee \bot} \quad \dfrac{}{P \vee \bot, P \vdash P} \quad \dfrac{\dfrac{}{P \vee \bot, \bot \vdash \bot}}{P \vee \bot, \bot \vdash P}\,\bot\text{-E}}{\dfrac{P \vee \bot \vdash P}{\vdash P \vee \bot \Rightarrow P}\,\Rightarrow\text{-I}}\,\vee\text{-E}$$

Exercise 2.7

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash \neg Q \Rightarrow \neg P} \quad \dfrac{}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash \neg Q}}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash \neg P}\,\Rightarrow\text{-E} \quad \dfrac{}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash P}}{\neg Q \Rightarrow \neg P, P, \neg Q \vdash \bot}\,\neg\text{-E}}{\neg Q \Rightarrow \neg P, P \vdash Q}\,\neg\text{-I}}{\neg Q \Rightarrow \neg P \vdash P \Rightarrow Q}\,\Rightarrow\text{-I}}{\vdash (\neg Q \Rightarrow \neg P) \Rightarrow P \Rightarrow Q}\,\Rightarrow\text{-I}$$

Exercise 2.8

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(P \vee \neg P), \neg P \vdash \neg(P \vee \neg P)} \quad \dfrac{\dfrac{}{\neg(P \vee \neg P), \neg P \vdash \neg P}}{\neg(P \vee \neg P), \neg P \vdash P \vee \neg P}\,\vee\text{-I}_2}{\neg(P \vee \neg P), \neg P \vdash \bot}\,\neg\text{-E} \quad \dfrac{\neg(P \vee \neg P), P \vdash \neg(P \vee \neg P) \quad \dfrac{\dfrac{}{\neg(P \vee \neg P), P \vdash P}}{\neg(P \vee \neg P), P \vdash (P \vee \neg P)}\,\vee\text{-I}_1}{\dfrac{\neg(P \vee \neg P), P \vdash \bot}{\neg(P \vee \neg P) \vdash \neg P}\,\neg\text{-I}}\,\text{RAA}}{\neg(P \vee \neg P) \vdash \bot}\,\neg\text{-E}}{\vdash P \vee \neg P}\,\text{RAA}$$

Exercise 2.9

This is not always possible. For example, in the empty context there is no proof showing that P holds (for some atomic propositional variable P). For example, in the valuation mapping $\nu(\mathrm{P}) = \mathbf{F}$, the formula P is false; if there was a derivation showing that $\vdash \mathrm{P}$ holds, this would contradict soundness.

Exercise 2.10

The proofs are not necessarily unique. For example, there are two possible derivations of $\mathrm{P} \wedge \mathrm{P} \vdash \mathrm{P}$ using the two different rules for conjunction elimination.

Exercise 2.11

```
or : Bool × Bool → Bool
or(F,F) = F
or(F,T) = T
or(T,F) = T
or(T,T) = T

implies : Bool × Bool → Bool
implies(F,F) = T
implies(F,T) = T
implies(T,F) = F
implies(T,T) = T
```

Exercise 2.12

Soundness is a bit easier to show than completeness. To establish that our system of natural deduction rules is *sound*, it suffices to show that each of our rules holds. This is fairly easy to check using *rule induction* on the possible derivations of $\vdash \mathrm{p}$.

3 Reasoning about programs

In this final chapter we show how to develop a *logic* to reason about *programs*. Where the previous chapter defined natural deduction, a set of proof rules to reason about propositional logic formulas, this chapter aims to achieve something similar for reasoning about (simple) imperative programming languages.

Syntax of programs

Before we can talk about semantics, we need to fix the (syntax of) the programming language that we will be studying. Here we fix the syntax for our programs p, (integer) expressions e and boolean expressions b using the following BNF equations:

$$
\begin{array}{lll}
\mathrm{p} & ::= & \mathrm{x} := e \\
& \mid & \mathrm{p}_1 ; \mathrm{p}_2 \\
& \mid & \text{if } \mathrm{b} \text{ then } \mathrm{p}_1 \text{ else } \mathrm{p}_2 \\
& \mid & \text{while } \mathrm{b} \text{ do } \mathrm{p} \\
e & ::= & \mathrm{n} \mid \mathrm{x} \mid e + e \mid e \times e \mid e - e \\
\mathrm{b} & ::= & \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 \leqslant e_2 \mid \neg\, \mathrm{b} \mid \mathrm{b}_1 \;\&\&\; \mathrm{b}_2
\end{array}
$$

Here we assume that n is drawn from the set of integers; variables such as x are drawn from some fixed set of variable names, **Var**. We will refer to this set of inductively defined programs as **While**.

This is, of course, not a real programming language. There are many operations missing—such as the disjunction of booleans, division and exponents of numbers—but our aim is *not* to try and define the complete syntax of a realistic language, but rather to define a minimal language that is small enough to study. The more language features we add, the more complex our semantics will become. Similarly, our semantics will avoid many low-level details, such as integer overflows and memory management. Nonetheless, I hope that you can recognize the core of many programming languages in these definitions.

Semantics for expressions

Before we study the semantics of *programs*, we will define the semantics of expressions. To do so, we will define a pair of functions that map expressions and boolean expression to their corresponding integer and boolean respectively.

These functions follow the same pattern as the semantics for propositional logic formulas we saw in the previous chapter. In the previous chapter, we saw how passing in a *truth assignment* can be used to assign semantics to propositional logic variables. In the semantics of expressions, we have a similar problem: as our expressions may contain variables, such as x + 3, our semantics we cannot define a semantics mapping expressions to integers directly.

To evaluate expressions containing variables, we need information about the current state of the computer's memory. To model this, our semantics will take an argument, σ : **Var** $\to$ **Int**, that maps each variable to its current value in memory. We can then define our semantics for integer expressions as follows:

$$
\begin{aligned}
[\![e]\!] &: (\textbf{Var} \to \textbf{Int}) \to \textbf{Int} \\
[\![n]\!](\sigma) &= n \\
[\![x]\!](\sigma) &= \sigma(x) \\
[\![e_1 + e_2]\!](\sigma) &= [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma) \\
[\![e_1 \times e_2]\!](\sigma) &= [\![e_1]\!](\sigma) \times [\![e_2]\!](\sigma) \\
[\![e_1 - e_2]\!](\sigma) &= [\![e_1]\!](\sigma) - [\![e_2]\!](\sigma)
\end{aligned}
$$

Running this semantics on expressions is sometimes referred to as *evaluation*.

Example

Suppose we want to evaluate x + 3, given the current memory $\sigma : V \to$ **Int** for which we know that $\sigma(x) = 7$. We can then proceed as follows:

$$[\![x + 3]\!](\sigma) = [\![x]\!](\sigma) + [\![3]\!](\sigma) = \sigma(x) + 3 = 7 + 3 = 10$$

It may seem like this semantics 'does nothing'. How should we read the following line from the definition above?

⟦ e_1 + e_2 ⟧ (σ) ; = ; ⟦ e_1 ⟧(σ) + ⟦ e_2 ⟧(σ)

It is important to distinguish the two occurrences of the + symbol in this formula. On the left hand side, we indicate that we are defining our semantics for the case that our expression is of the form $e_1 + e_2$. Here the + symbol is part of the *syntax* of our expression language. We could equally well have chosen to use a different operator to represent addition, such as $\oplus$. The second plus symbol, used on the right-hand side of the equation, is the regular addition between integers. The integers being added here are $ $[\![e_1]\!](\sigma)$ and $[\![e_2]\!](\sigma)$ corresponding to the two sub-expressions of the + operator of our expression language. Something similar appeared in our semantics for propositional logic in the previous chapter, where we mapped the disjunction of propositional logic formulas ($\vee$) to the `or` operation on booleans.

Exercise 3.1

Give a similar semantics for boolean expressions.

Semantics for programs

Now that we have a semantics for expressions, we can focus on how to define a semantics of our *programs*. Where the semantics for (boolean) expressions closely resembles the semantics for propositional logic formulas we saw in the previous chapter, it is not quite so clear how to define the semantics of our programs. Consider a program such as:

```
x := 17
```

What is the result of this program? The semantics we saw previously mapped expressions to integers and boolean expressions to booleans. But what value does this program return? We might adopt some convention, such as that the result of the above program is 17 – but this is not enough. Consider the following program that never terminates:

```
while(true) do
  {x := x + 1}
```

What value does this program return?

These examples highlight two issues that we need to address. Firstly, programs *modify* the state of our computer's memory, rather than return a value. A consequence of this observation is that our semantics should account for *how* a program modifies the initial state of our computer's memory. Secondly, our programs may *diverge*, not returning any result at all—-any semantics we define to assign meaning to our programs must also account for programs that never terminate.

Example: program execution

Suppose we start have the following program:

```
1  x := 3;
2  p := 0;
3  i := 1;
4  while (i ⩽ x) do
5  {
6    p := p+i;
7    i := i+1
8  }
```

We start execution from some begin state – let's assume that the variables x, p and i all start as 0,1,2 respectively. That is initially we're in a state σ such that:

$$\sigma(\mathtt{x}) = 0 \quad \sigma(\mathtt{p}) = 1 \quad \sigma(\mathtt{i}) = 2$$

To execute this program, we read through it line-by-line, performing the corresponding updates to our state. We can run our program by hand, storing the state of all three variables in the following table:

line	change	$\sigma(\mathtt{x})$	$\sigma(\mathtt{p})$	$\sigma(\mathtt{i})$
1	initial state	0	1	2
2	execute x:=3	3	1	2
3	execute p:=0	3	0	2
4	execute i:=1	3	0	1
6	enter while and execute p:=p+i	3	1	1
7	execute i:=i+1	3	1	2
6	enter while and execute p:=p+i	3	3	2
7	execute i:=i+1	3	3	3
6	enter while and execute p:=p+i	3	6	3
7	execute i:=i+1	3	6	4

The 'line' column refers to the line number of the statement we are executing; the 'change' gives a (human-readable) description of what has happened; the other three columns contain the current value of the three variables in which we are interested.

Here we can see that our program terminates in the final state where

$\sigma(\mathtt{x}) = 3 \quad \sigma(\mathtt{p}) = 6 \quad \sigma(\mathtt{i}) = 4$

This example intends to give some intuition of how a program is executed, modifying the state of the computer's memory along the way. In the next sections, we will try to nail down this intuition further, giving a precise mathematical account of program execution.

Modelling state

Just as we did for our semantics for expressions, we will need to model the current state of our computer's memory, storing the value of all the variables. We can do so using a function, mapping variable names to the values they store:

σ : **Var** $\to$ **Int**

In what follows, we will sometimes refer to the set of functions **Var** $\to$ **Int** as the set of all possible states, **State**.

Given a state σ, if we want to know the value of a given variable $\mathtt{x}$, we can simply look up the corresponding value by passing it as argument to this function, $\sigma(\mathtt{x})$.

We will sometimes also need to *update* the current state. To do so, we write $\sigma[\mathtt{y} \mapsto \mathtt{n}]$ for the memory that is the same as σ for all variables in **Var** *except* $\mathtt{y}$, where it stores the value $\mathtt{n}$. In other words, this updates the current memory at one location, setting the value for $\mathtt{y}$ to $\mathtt{n}$.

More formally, the new state σ' of our memory after the update $\sigma[\mathtt{y} \mapsto \mathtt{n}]$ is defined as follows:

$$\sigma'(\mathtt{x}) = \begin{cases} \mathtt{n} & \text{if } \mathtt{x} = \mathtt{y} \\ \sigma(\mathtt{x}) & \text{otherwise} \end{cases}$$

Operational semantics

We can now define an inductive relation capturing the semantics of our programming language. The key idea is that we define a relation on **While**$\times$ **State**$\times$ **State**– that is given the current state of the computer's memory and the program that we're executing, executes the program to produce some final state.

This formalizes the example we had a few slides ago, where we 'stepped through' the execution of a program studying how the state changed at every step.

We will write use the following notation:

$\langle$ p , σ $\rangle \to \sigma$'

To mean that executing the program p with the initial state σ terminates in some final state σ'. This relation defines what is called an **operational semantics** for our programs, describing how to execute a program step by step. Our programming language has four language constructs:

- Assignments – $\mathrm{x} := e$
- Conditionals – if b then p_1 else p_2 fi
- Sequential composition – $\mathrm{p}_1; \mathrm{p}_2$
- Loops – while b do p end

We will now give rules describing the meaning of each of these constructs one by one.

Assignment

There is a single rule describing the behaviour of assigments:

$$\frac{[\![e]\!](\sigma) = \mathrm{n}}{\langle \mathrm{x} := e, \sigma \rangle \ \rightarrow \ \sigma[\mathrm{x} \mapsto \mathrm{n}]} \text{ Assignment}$$

This rule shows that each assignment statement terminates in a single step. Here we use the notation introduced previously, $\sigma[\mathrm{x} \mapsto \mathrm{n}]$, to describe the final state after executing the assignment. This final state is identitical to the initial state, σ, but the variable x now has the value n.

Example: executing assignments Given a state σ satisfying $\sigma(\mathrm{y}) = 3$, we can describe the behaviour of the command $\mathrm{x} := \mathrm{y} + 2$ as follows:

$$\frac{[\![\mathrm{y} + 2]\!](\sigma) = 5}{\langle \mathrm{x} := \mathrm{y} + 2, \sigma \rangle \ \rightarrow \ \sigma[\mathrm{x} \mapsto 5]}$$

Conditional statements

In contrast to assignments, we need to rules to describe how to evaluate an if-then-else statement, one for each possible execution path:

$$\frac{[\![\mathrm{b}]\!](\sigma) = \text{true} \qquad \langle \mathrm{p}_1, \sigma \rangle \rightarrow \sigma'}{\langle \text{ if } \mathrm{b} \text{ then } \mathrm{p}_1 \text{ else } \mathrm{p}_2 \text{ fi}, \sigma \rangle \rightarrow \sigma\text{'}} \text{ If-true}$$

$$\frac{[\![\mathrm{b}]\!](\sigma) = \text{false} \qquad \langle \mathrm{p}_2, \sigma \rangle \rightarrow \sigma'}{\langle \text{ if } \mathrm{b} \text{ then } \mathrm{p}_1 \text{ else } \mathrm{p}_2 \text{ fi}, \sigma \rangle \rightarrow \sigma\text{'}} \text{ If-false}$$

Both these rules begin by evaluating the 'guard' b. If the guard b evaluates to true, we continue executing the `then` branch, leaving the state unchanged. If the guard b evaluates to false, we continue executing the `else` branch, leaving the state unchanged.

Example: executing a conditional Suppose we start from a state σ satisfying $\sigma(x) = 3$ and $\sigma(y) = 10$

We can execute the following program:

```
if x < y then r := x else r := y
```

Using the previous derivation rules, we can show:

$$\frac{[\![x < y]\!](\sigma) = \text{true} \qquad \dfrac{[\![x]\!](\sigma) = 3}{\langle\, r := x \,,\, \sigma \,\rangle \to \sigma[r \mapsto 3]}\ \text{Assignment}}{\langle\, \text{if } x < y \text{ then } r := x \text{ else } r := y \text{ fi} \,,\, \sigma \,\rangle \to \sigma[r \mapsto 3]}$$

Sequential composition

We now turn our attention to the penultimate language construct: sequential composition, $p_1; p_2$:

$$\frac{\langle\, p_1 \,,\, \sigma \,\rangle \to \sigma' \qquad \langle\, p_2 \,,\, \sigma' \,\rangle \to \sigma''}{\langle\, (p_1; p_2) \,,\, \sigma \,\rangle \to \sigma''}\ \text{Seq}$$

Here we see how nicely our inference rule notation can be used to capture the intended behaviour of sequential composition: we execute p_1 until it terminates in some state σ', which we then use to start execution of p_2. When p_2 terminates in some state σ'', the composite program $p_1; p_2$ terminates in this state.

Exercise 3.2

Let p refer to the program:

```
x := x + y; y := x + 3
```

Now let σ be a state such that: $\sigma(x) = 1$ $\sigma(y) = 1$

Give a state τ and derivation showing that $\langle\, p \,,\, \sigma \,\rangle \to \tau$.

Loops

Finally, we turn our attention to the last language construct: while-loops. Just as we saw for conditionals, we need two separate rules, depending whether or no the guard b holds. The first rule, when the guard evaluates to false, is the easiest of the two:

$$\frac{[\![\mathrm{b}]\!](\sigma) = \mathtt{false}}{\langle\ \text{while } \mathrm{b} \text{ do } \mathrm{p}\ ,\ \sigma\ \rangle \to \sigma}\ \text{While-false}$$

If the guard b evaluates to false, we do not have to do any further work to execute the while loop and we terminate in the final state σ. When the guard is true, however, the semantics is a bit more complex:

$$\frac{[\![\mathrm{b}]\!](\sigma) = \mathtt{true} \qquad \langle\ \mathrm{p}\ ,\ \sigma\ \rangle \to \sigma\text{'} \qquad \langle\ \text{while } \mathrm{b} \text{ do } \mathrm{p}\ ,\ \sigma\text{'}\ \rangle \to \sigma\text{''}}{\langle\ \text{while } \mathrm{b} \text{ do } \mathrm{p}\ ,\ \sigma\ \rangle \to \sigma\text{''}}\ \text{While-true}$$

This second rule combines many of the elements of sequential composition and conditional evaluation that we saw previously. If the guard b evaluates to true, we execute the loop body p. If this terminates and results in some final state σ', we execute the while loop again, now starting with the state σ'. If this (eventually) terminates in the final state σ'', the entire while loop terminates in σ''.

Exercise 3.3

Given the following program p:

```
while(i ⩽ n) do { x := x * i; i := i + 1}
```

Given an initial state σ that satisfies: $\sigma'(\mathtt{x}) = 1 \quad \sigma'(\mathtt{i}) = 1$

Describe the size and shape of possible derivations $\langle\ \mathrm{p}\ ,\ \sigma\ \rangle \to \sigma$' in terms of the initial value of n in σ. What does this program compute?

Exercise 3.4

Given the following program p:

```
while(true) do x := x + 1
```

Are there states σ and σ' such that there is a derivation showing $\langle\ \mathrm{p}\ ,\ \sigma\ \rangle \to \sigma$'?

Exercise 3.5

The semi-colon operator is used to compose two programs, running one after the other. Given three programs, we can compose them in two possible ways: $(\mathrm{p}_1; \mathrm{p}_2); \mathrm{p}_3$ or $\mathrm{p}_1; (\mathrm{p}_2; \mathrm{p}_2)$. Does the order of composition matter? If so, give a program that illustrates this; if not, argue why using the operational semantics presented in this section. What about $\mathrm{p}_1; \mathrm{p}_2$ and $\mathrm{p}_2; \mathrm{p}_1$? Are these the same?

From operational semantics to program logic

These operational semantics determine how a program is executed. For each program p and initial state σ, we can repeatedly apply the rules from our semantics to try and find a state σ' such that:

$\langle p , \sigma \rangle \to \sigma'$.

But now consider the following program:

```
if x < y then r := x else r := y
```

Can we prove that after execution `r` will *always* store the minimal value of `x` and `y`? Using our operational semantics, we can find a derivation, witnessing the execution this program for a specific intitial state—but what if we want to reason about *all* possible initial states?

This motivates the shift from *operational semantics* to *program logic*. In the remainder of this chapter, we will describe *specifications*, that describe how a program should behave, together with a logic for *proving* that a particular program satisfies its specification. Finally, we will relate this logic to the operational semantics we saw in the previous chapter.

Specifications

A **formal specification** is a mathematical description of what a program should do.

Such a specification ignores many important details, such as the *non-functional* requirements about how fast the program is, the language used for its implementation, the open-source license under which it is available, or the hardware on which it must run.

Instead, we will use a formal specification to answer one question:

Is this program doing what it should?

There are different ways to specify a program's behaviour. In this chapter, we consider a *specification* to consist of a *preconditian* and a *postcondition*. Intuitively, the precondition captures the assumptions the program makes about the initial state; the postcondition expresses the properties that are guaranteed to hold after the program has finished executing.

Examples Previously, we saw the following program that assigns to `r` the miminum of `x` and `y`.

```
if x < y then r := x else r := y
```

As this program does not make any assumption about the initial state, the precondition is simply `true`—which holds of any state. The postcondition states that `r = min(x,y)`, i.e., after execution the value assigned to `r` is the minimum of `x` and `y`.

Another simple programming task might be to write a few lines of code that swaps the values of two variables, x and y. We can write the specification for this program as follows:

- The precondition assumes that x and y have some value N and M respectively. Written more formally: $x = N \wedge y = M$;
- The postcondition states that the values of x and y have been swapped, that is, after execution the proposition $x = M \wedge y = N$ should hold.

Notation

To define our logic for reasoning about programs, we introduce some new notation. In what follows, we will write

$$\{P\}\ p\ \{Q\}$$

to state that for any initial state σ satisfying the precondition P, whenever executing the program p terminates in some final state σ', then σ' satisfies the postcondition Q. Such a triple of precondition, program and postcondition is sometimes referred to as a *Hoare triple*, in honour of the British computer scientist Tony Hoare, who was one of the pioneers of this approach to reasoning about programs.

In a sense, we will use this notation to denote a relation on **State** $\times$ **While** $\times$ **State**.

Examples

Let's consider some examples of Hoare triples that we expect should hold.

- $\{x = 3\}\quad x := x + 1\quad \{x = 4\}$

Unsurprising: if $x = 3$, after executing $x := x + 1$, we should be able to establish that $x = 4$.

- $\{x = N \wedge y = M\}\quad z := x; x := y; y := z\quad \{x = M \wedge y = N\}$

This is more interesting: it works for *any* values of A and B – this describes many possible executions, starting from some state for which the precondition holds.

- $\{\text{true}\}\quad \text{while true do } p := 0\quad \{p = 500\}$

This final example is more devious: remember that we need to show that *if* our program terminates in a state σ', the postcondition holds—-but this program never terminates! Hence we can make any claim we like about the final state, as it is never reached.

There are variations of the logic we will present here that can be used to establish that a program terminates and satisfies the postcondition. This is sometimes referred to as *total correctness*; for the sake of simplicity, however, we ignore termination issues for the moment.

Hoare logic

We have given a handful of examples of Hoare triples that we expect to hold, but we haven't yet defined *when* a given Hoare triple holds, or how it can be established. This is akin to giving formulas in predicate logic that should be true, without making precise how they can be proven.

In this chapter, we will define a handful of inference rules for proving Hoare triples of the form $\{\mathrm{P}\}\ \mathrm{p}\ \{\mathrm{Q}\}$. Once again, we present the rules one by one for all of the constructs in the While 'programming language.'

Hoare logic – assignment

What rule should we use for assignment? We've seen one example:

$$\{\,\mathsf{x} = 3\,\}\quad \mathsf{x}\ :=\ \mathsf{x}\ +\ 1\quad \{\,\mathsf{x} = 4\,\}$$

We could generalise this:

$$\{\,\mathsf{x} = \mathsf{N}\,\}\quad \mathsf{x}\ :=\ \mathsf{x}\ +\ 1\quad \{\,\mathsf{x} = \mathsf{N} + 1\,\}$$

But what if we want to assign another expression than $\mathsf{x}\ +\ 1$? We should try to find a single rule capable of reasoning about *all* assignments. This leads to the following general rule for reasoning about assignment statements:

$$\frac{}{\{\,\mathrm{Q}[x\backslash e]\,\}\quad \mathsf{x} := \mathsf{e}\quad \{\,\mathrm{Q}\,\}}\ \text{Assign}$$

Here we write $\mathrm{Q}[x\backslash e]$ for the result of replacing all the occurrences of x with e in Q. The easiest way to read this rule is starting from the postcondition. Under what precondition does the postcondition Q hold, after performing the assignment $x := e$? Well, the assigment statement will not magically cause Q to hold; the only thing the statement changes is the value of x, replacing it with the value associated with the expression e. As a result, the Q should already hold, where the occurrences of x are replaced with e, which is exactly what the precondition $\mathrm{Q}[x\backslash e]$ states.

Let's look at a few examples of this rule in action.

$$\frac{}{\{\,y = 3\,\}\,x := 3\,\{\,y = x\,\}}\ \text{Assign}$$

This example derivation shows that if the precondition $y = 3$ holds, after the assignment $x := 3$, we can conclude that x and y are equal. To be more precise, the precondition should read

$$(x = y)[x\backslash 3]$$

but we have simplified this to the equivalent predicate, $y = 3$.

Exercise 3.6

What is the result of performing the following substitutions?

1. $(x + y \geqslant 3)[x \backslash 3]$
2. $(x + y \geqslant 3)[y \backslash x + 2]$
3. $(x + y \geqslant 3)[z \backslash y]$
4. $(x + y \geqslant 3)[y \backslash z]$
5. $(x + x \geqslant x)[x \backslash 0]$
6. $(x + x \geqslant y)[x \backslash y]$
7. $(x + x \geqslant y)[y \backslash x]$

Exercise 3.7

Use the Assignment rule to find a suitable precondition so that each of the following Hoare triples holds::

1. { ? } x := x + 1 {x $\geqslant$ 10}
2. { ? } x := y + 1 {x $\geqslant$ y}
3. { ? } x := y + 1 {x $\leqslant$ y}
4. { ? } x := x + 1 {x = x}

Can you simplify the preconditions you found any further?

Hoare logic – if statements

Next, we consider the inference rule that can be used to reason about if-statements:

$$\frac{\{ P \wedge b \} \;\; p_1 \;\; \{Q\} \qquad \{ P \wedge \neg b \} \;\; p_2 \;\; \{Q\}}{\{ P \} \quad \text{if } b \text{ then } p_1 \text{ else } p_2 \quad \{ Q \}} \text{ If}$$

A conditional statement of the form 'if b then p_1 else p_2' will execute either p_1 or p_2. In order for the postcondition Q to hold after executing the conditional statement, we require that Q holds after executing both p_1 and p_2—but the associated preconditions change. By checking whether the guard b holds or not, we learn something. As a result, the precondition changes in both branches of the if-statement: we may assume that $P \wedge b$ holds when taking the then branch; and similarly, that $P \wedge \neg b$ holds when taking the else branch.

Note that these rules use the guard b as if it is a predicate on states, just like P. It would be a bit more precise to write $\{P \wedge [\![b]\!](\sigma)\}$ to indicate that the boolean expression b should be evaluated to a truth value.

Example Use the two rules we have seen so far to show that:

$$\{\, 0 \leqslant x \leqslant 5 \,\} \quad \text{if } x < 5 \text{ then } x := x+1 \text{ else } x := 0 \text{ fi} \quad \{\, 0 \leqslant x \leqslant 5 \,\}$$

If use the variable P to refer to the condition that $0 \leqslant x \leqslant 5$, the derivation has the following structure:

$$\dfrac{\dfrac{}{\{\, P \wedge x < 5 \,\}\ x := x+1\ \{\, P \,\}}\ \text{Assign} \qquad \dfrac{}{\{\, P \wedge x > 5 \,\}\ x := 0\ \{\, P \,\}}\ \text{Assign}}{\{\, P \,\} \quad \text{if } x < 5 \text{ then } x := x+1 \text{ else } x := 0 \text{ fi} \quad \{\, P \,\}}\ \text{If}$$

While the If-rule is used correctly here, it is less obvious that the two assignment rules are correct. Indeed, neither is precisely of the form:

$$\dfrac{}{\{\, Q[x\backslash e] \,\} \quad x := e \quad \{\, Q \,\}}\ \text{Assign}$$

So is there a problem with this proof?

Hoare logic – rule of consequence

The example above shows that oftentimes the pre- and postconditions that arise when writing out a derivation do not line up *exactly* with the expected pre- and postconditions. We have already seen an example of this in the assignment rule, where we wrote $y = 3$, rather than the precondition that our rule for assignments requires:

$$(x = y)[x\backslash 3]$$

Clearly, we should be allowed to replace our pre- and postconditions with logically equivalent conditions. More formally, the following rule should hold:

$$\dfrac{P \Leftrightarrow P' \qquad \{\, P' \,\}\ p\ \{Q'\} \qquad Q \Leftrightarrow Q'}{\{\, P \,\}\ p\ \{\, Q \,\}}$$

However, we there is an even more general rule in Hoare logic, sometimes referred to as the *rule of consequence*:

$$\dfrac{P \Rightarrow P' \qquad \{\, P' \,\}\ p\ \{Q'\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ p\ \{\, Q \,\}}\ \text{Consq}$$

This rule says that, in order to establish that $\{\,P\,\}\ p\ \{\,Q\,\}$ holds, it suffices to show that:

- $\{\,P'\,\}\ p\ \{Q'\,\}$ holds;
- for some *precondition* P' that is *stronger*, that is, $P' \Rightarrow P$;
- for some *postcondition* Q' is *weaker*, that is, $Q \Rightarrow Q'$.

We can justify this rule by thinking back to what a statement of the form $\{P\}\ p\ \{Q\}$ means:

> For each state σ that satisfies the precondition P, if executing our program p in the initial state σ terminates in some state σ', that is, $\langle p, \sigma\rangle \to \sigma'$, then σ' must satisfy Q.

This intuition can be used to motivate why the rule of consequence holds. Let us focus on the postcondition for the moment.

If $\{P\}\ p\ \{Q'\}$ holds and $Q' \Rightarrow Q$. If we execute our program p, yielding some final state σ', we know that σ' satisfies Q'. From our second assumption, however, we also know that σ' satisfies Q. Hence we can conclude that $\{P\}\ p\ \{Q\}$ as required.

While we will sometimes silently rewrite pre- and postconditions to equivalent logical expressions, we will try to be explicit about where we apply the rule of consequence.

Exercise 3.8

Give a similar argument showing that if $\{P'\}\ p\ \{Q\}$ holds and $P \Rightarrow P'$, then $\{P\}\ p\ \{Q\}$ also holds.

Exercise 3.9

Suppose that the following Hoare triple holds:

$$\{\,P\,\}\ \ \text{if } b \text{ then } p_1 \text{ else } p_2\ \ \{R\}$$

Prove that the following statement also holds:

$$\{\,P\,\}\ \ \text{if } \neg b \text{ then } p_2 \text{ else } p_1\ \ \{R\}$$

Hoare logic – sequential composition

To compose larger programs, we need a rule for sequential composition:

$$\frac{\{\,P\,\}\ \ p_1\ \ \{R\} \qquad \{\,R\,\}\ \ p_2\ \ \{Q\}}{\{\,P\,\}\ \ \ p_1; p_2\ \ \ \{\,Q\,\}}\ \text{Seq}$$

The rule for composition of programs is particularly beautiful – it may remind you of function composition.

If we know that P holds of our initial state, we can run p_1 to reach a state satisfying R; but now we can run p_2 on this state, to produce a state satisfying Q. In this way, we can break the verification of a big program into smaller parts.

Exercise 3.10

Use the Assignment and Seq rules to find a suitable precondition so that each of the following Hoare triples holds:

1. { ? } y := x; z := y {z $\geqslant$ 10}
2. { ? } x := y; z := x + 5 {z $\geqslant$ 10}
3. { ? } x := y; z := y + 5 {z $\geqslant$ 10}
4. { ? } x := y + 3; z := z + x {z $\geqslant$ 10}

Can you simplify the preconditions you found any further?

Exercise 3.11

In the question above, you are given a *postcondition* and are asked to compute a *precondition*. Give an example explaining why it is not possible using these rules, to compute the postcondition given the precondition.

Exercise 3.12

In a previous exercise, we used the operational semantics to argue that $(\mathrm{p}_1; \mathrm{p}_2); \mathrm{p}_3$ and $\mathrm{p}_1; (\mathrm{p}_2; \mathrm{p}_3)$ always behave the same. Argue that these two programs are the same using the Seq-rule from Hoare logic described above.

Hoare logic – while-loops

The final rule we need to complete our treatment of Hoare logic is the rule to handle while-loops. It turns out that the while-rule is one of the most subtle rules in Hoare logic. We would expect this rule to have the following form:

$$\frac{\{\,??? \wedge \mathrm{b}\,\}\quad \mathrm{p}\quad \{???\}}{\{\,\mathrm{P}\,\}\quad \text{while } \mathrm{b} \text{ do } \mathrm{p}\quad \{\,??? \wedge \neg \mathrm{b}\,\}}\ \text{While}$$

Given what we know about the behaviour of while-loops and the rule for conditionals we saw previously, we would expect that:

- some precondition P should hold initially;
- the loop body may assume that the guard b is true;
- after having run to completion, we know that the guard b is no longer true.

But how should we fill in the question marks? When we first enter the loop body, we know that the precondition of the entire while-loop, P, still holds:

$$\frac{\{ P \wedge b \} \quad p \quad \{???\}}{\{ P \} \quad \text{while } b \text{ do } p \quad \{ ??? \wedge \neg b \}} \text{ While}$$

After running the loop body once, we may need to execute the loop body again (and again and again and again). As a result, we observe that the condition P *must hold after every iteration of the loop body*:

$$\frac{\{ P \wedge b \} \quad p \quad \{P\}}{\{ P \} \quad \text{while } b \text{ do } p \quad \{ ??? \wedge \neg b \}} \text{ While}$$

This leaves just one set of question marks: the postcondition of the entire loop. After

$$\frac{\{ P \wedge b \} \quad p \quad \{P\}}{\{ P \} \quad \text{while } b \text{ do } p \quad \{ P \wedge \neg b \}} \text{ While}$$

After running the loop body over and over again, the postcondition of the entire while statement says that both P and $\neg b$ hold. We call P the **loop invariant** – it continues to hold after every execution of the body of the while loop. Finding the right loop invariant is one of the key creative steps necessary when reasoning about programs.

By itself, it may seem strange that a while-loop as (almost) the same pre- and postcondition. How can we ever use it to compute anything interesting? Remember, however, that the body of the while loop may modify the current state of our program; even if the condition P must remain true during execution, it may refer to variables whose values change after an assignment in the body of the loop.

Example To illustrate how loops work, we give a derivation proving the following statement:

$\{ x \geqslant 5 \}$ `while x > 5 do x := x - 1 od` $\{ x \geqslant 5 \wedge x \leqslant 5 \}$

This loops assumes that x is at least 5 and counts down until x is precisely 5. Note how the postcondition is equivalent to $\{ x = 5 \}$. We have kept it in this form, to illustrate how the rule for while-loops works.

$$\dfrac{\dfrac{}{\{ x - 1 \geqslant 5 \} \quad x := x - 1 \quad \{ x \geqslant 5 \}} \text{ Assignment}}{\{ x \geqslant 5 \wedge x > 5 \} \quad x := x - 1 \quad \{ x \geqslant 5 \}} \text{ Consq}}{\{ x \geqslant 5 \} \quad \text{while } x > 5 \text{ do } x := x - 1 \{ x \geqslant 5 \wedge x \leqslant 5 \}} \text{ While}$$

In several points during this proof, we have not written the simplest pre- or postcondition. For example, clearly $\{ x \geqslant 5 \wedge x > 5 \}$ can equally well be written as $\{ x > 5 \}$; writing out the conjunction explicitly, however, makes it easier to establish that the While-rule has been applied correctly.

Case study: gcd

To illustrate how to use the rule for while-loops, let's consider small—but non-trivial—program p:

```
while x != y {
  if (x > y) {
    x := x-y}
  else {
    y := y-x}
}
```

What does this program compute? In this section, I will argue that this program computes the *greatest common divisor* of x and y. More precisely, provided x and y are greater than zero in the initial state σ, this program terminates in a final state σ' such that:

$$\sigma'(x) = \sigma'(y) = \mathrm{gcd}(\sigma(x),\sigma(y))$$

How can we prove this? I will not give the complete derivation, but explain how to use the rules of Hoare logic we saw previously to establish this program is correct.

First, let us make precise what the pre- and postconditions are that we wish to establish for our program p. The precondition is a bit unusual. The program may repeatedly updates both x and y; as the value of x and y in the *final state* should be equal to the greatest common divisor of the value of x and y in the *initial state*, we will need a way to record the initial value of x and y. One way to achieve this is by introducing so-called 'ghost variables', N and M, that do not occur in our program but can be used to relate the initial and final values of our variables. We therefore propose the following precondition:

$$x = N \wedge y = M$$

The postcondition is a bit easier. Upon completion, x and y should be equal to the greatest common divisor of N and M, which we formalize in the following postcondition:

$$x = \mathrm{gcd}(N,M) \wedge x = y$$

We would now like to prove the following statement:

$$\{ x = N \wedge y = M \} \quad \mathrm{p} \quad \{ x = \mathrm{gcd}(N,M) \wedge x = y \}$$

As p starts with a while-loop, we need to apply our while-rule to establish this statement. What invariant should we choose? This is not at all obvious and requires some creativity. The key insight that makes this algorithm work, however, is that during execution the greatest common divisor of x and y is is

equal to the greatest common divisor of N and M. Using the rule of consequence, we therefore rephrase the Hoare triple we wish to establish as:

$$\{ \gcd(N,M) = \gcd(x,y) \} \quad p \quad \{ \gcd(N,M) = \gcd(x,y) \wedge x = y \}$$

Now that we have identified the loop invariant, we can use the while-rule. Doing so, leaves the following proof obligation:

$$\{ \gcd(N,M) = \gcd(x,y) \wedge x \neq y \} \text{ if } (x > y) \text{ then } x := x-y \text{ else } y := y-x \ \{ \gcd(N,M) = \gcd(x,y) \}$$

Here we can apply the if-rule to leave two proof obligations. For the moment, we focus on the then-branch:

$$\{ \gcd(N,M) = \gcd(x,y) \wedge x > y \} \ x := x-y \ \{ \gcd(N,M) = \gcd(x,y) \}$$

To prove this, we would like to use the Hoare logic rule for assignments, that is, we need to show that if $x > y$ then

$$\gcd(x,y) = \gcd(x - y, x)$$

At this point, we have boiled the verification of the program p down to establishing the above property of greatest common divisors. We no longer need Hoare logic to reason about p, but can now rely on some simple algebra.

To prove the above property, suppose that k divides both x and y. More formally, there are numbers n and m such that:

$$x = k \times n \wedge y = k \times m$$

But then k also divides x - y since:

$$x - y = (k \times n) - (k \times m) = k \times (n - m)$$

Hence if k is a divisor of both x and y it is also a divisor of x - y as required.

To complete the proof, there are still a few steps missing, which we leave as exercises to the reader.

Exercise 3.13

Show that k is the *greatest* common divisor of x and y if and only if it also the greatest common divisor a divisor of x - y and y.

Exercise 3.14

Give a similar argument to establish that the else-branch of p is correct.

The above proof is not a *formal* proof. It does not provide a complete derivation in terms of the rules of Hoare logic that we have seen—but arguably it is still a valid proof. Most proofs in mathematics are not given by providing a derivation using natural deduction, but rather explain to the reader why a

statement holds, appealing to their intuition; with enough effort, such proofs can be phrased in terms of the rules of natural deduction. The same is true of this proof: using the intuition we have developed previously, we can reason about a program's behaviour. This reasoning can be made precise in the form of a derivation. These derivations, however, are easy for a machine to check, but hide some of the thought process that went into their construction. For that reason, it can be more illuminating to give a 'proof' that leaves out some of the details that a formal derivation must include, but focuses no presenting the key creative steps instead.

Exercise 3.15

Consider the following program:

```
r := 0
while (r × r < n)
  { r := r + 1}
```

Give a similar argument for why this program computes the integer square root of n. Can you think of a more efficient search strategy for finding the integer square root?

Soundness and completeness

How can we be sure that we chose the right set of inference rules? We could have made a mistake in the definition of our inference rules for Hoare logic. A good check to convince ourselves that our rules are the right ones, is by relating them to our operational semantics somehow. In particular, we can show that the inference rules for Hoare logic that we have presented are both *sound* and *complete* with respect to our operational semantics.

Soundness If we can prove $\{P\}\ p\ \{Q\}$ then for all states σ such that $P(\sigma)$, if $\langle\, p\ ,\ \sigma\,\rangle \to \sigma$' then $Q(\sigma')$

Completeness For all states σ and τ and programs p, such that $\langle\, p\ ,\ \sigma\,\rangle \to \sigma$'. Then for all preconditions P and postconditions Q for which $P(\sigma) \Rightarrow Q(\sigma')$, there exists a derivation showing $\{P\}\ p\ \{Q\}$.

In other words, the rules for Hoare logic can be used to reason about *all possible program behaviours*! Where the operational semantics make explicit how a program is executed, these properties establish that we can prove a program meets its specification *without having to execute it*.

Exercise 3.16

Argue that for all programs p and propositions P, the following Hoare triples hold:

1. $\{\,\text{false}\,\}\ \ p\ \ \{\,P\,\}$
2. $\{\,P\,\}\ \ p\ \ \{\,\text{true}\,\}$

Discussion

The While programming language describes the core of many imperative programming languages, but it hopelessly incomplete. There are numerous language features in modern programming languages such as C# or Java that are missing, such as:

- *Objects* – Many modern programming languages have some notion of *class*, with both methods and variables associated. These classes may be abstract or contain virtual methods; both methods and variables may be inherited from super-classes. None of these features is present in our While language.
- *Primitive types* – In the While language, every variable is an integer. There is no support for other types such as strings, arrays, characters, bytes, or binary words.
- *Exceptions* – programs in the While language may loop, but cannot throw an exception. The control flow associated with throwing and catching exceptions can be quite subtle, and the rules for Hoare logic that we have presented here cannot be used to reason about code that may throw an exception.
- *Concurrency* – many programs run using separate *threads*, that execute at the same time. These threads may read and write from the same memory locations. Reasoning about concurrent programs is notoriously hard, but there are extensions to the Hoare logic rules we have seen here that make this possible.
- *Et cetera* – While programs cannot print or read from the command line, let alone open a window or observe a mouse click. There is no interaction with the operating system; there are no network primitives; almost every feature imaginable that modern programming languages support, cannot be handled by these rules.

So what is the point? Reading this list, you may feel like Hoare logic is of little no practical use—but the contrary is true! The ideas presented here form the basis of many modern verification tools and static analyzers. These tools try to detect all kinds of bugs automatically; if a user annotates a method with pre- and postconditions, these verification tools try to prove that the code satisfies its specification—using rules like those presented here. Even if there are no pre- and postconditions, these tools can identify possible null pointer exceptions or memory safety issues. The logic presented here powers modern verification tools that are built and used by some of the largest software development companies in the world.

Solutions to selected exercises

Exercise 3.1

$$
\begin{aligned}
[\![\mathrm{b}]\!] &: (\textbf{Var} \to \textbf{Int}) \to \textbf{Bool} \\
[\![\mathsf{true}]\!](\sigma) &= \mathsf{T} \\
[\![\mathsf{false}]\!](\sigma) &= \mathsf{F} \\
[\![e_1 = e_2]\!](\sigma) &= \begin{cases} \mathbf{T} & \text{if } [\![e_1]\!](\sigma) = [\![e_2]\!](\sigma) \\ \mathbf{F} & \text{otherwise} \end{cases} \\
[\![e_1 \leqslant e_2]\!](\sigma) &= \begin{cases} \mathbf{T} & \text{if } [\![e_1]\!](\sigma) \leqslant [\![e_2]\!](\sigma) \\ \mathbf{F} & \text{otherwise} \end{cases} \\
[\![\neg \mathrm{p}]\!](\sigma) &= \mathsf{not}([\![\mathrm{p}]\!](\sigma)) \\
[\![\mathrm{p} \&\& \mathrm{q}]\!](\sigma) &= \mathsf{and}([\![\mathrm{p}]\!](\sigma), [\![\mathrm{q}]\!](\sigma))
\end{aligned}
$$

Exercise 3.2

Define σ be a state such that: $\sigma'(\mathrm{x}) = 2 \quad \sigma'(\mathrm{y}) = 1$

And τ be a state such that: $\tau(\mathrm{x}) = 2 \quad \tau(\mathrm{y}) = 5$

Now we can give the following derivation:

$$
\dfrac{\dfrac{}{\langle\, \mathsf{y := x + 3} \,,\, \sigma \,\rangle \to \tau}\ \textsf{Assign} \qquad \dfrac{}{\langle\, \mathsf{x:= x + y} \,,\, \sigma \,\rangle \to \sigma'}\ \textsf{Assign}}{\langle\, \mathrm{p} \,,\, \sigma \,\rangle \to \tau}\ \textsf{Seq}
$$

We leave it to the reader to check that σ' and τ are used correctly in the assignment rules.

Exercise 3.3

If n is greater than one, a derivation will apply the While-true rule n - 1 times. In each iteration, $\mathtt{i}$ is incremented and x is overwritten with a new value. When the program terminates, x will have the value $1 \times 2 \times 3 \times \ldots \times \mathrm{n}$, that is, n!.

Exercise 3.4

To give a derivation of the form$\langle\, \mathrm{p} \,,\, \sigma \,\rangle \to \sigma$' we need to use one of the rules for while-loops. Given that the guard associated with the while loop in p is always true, we can only use the While-true rule. But since the While-true rules requires that we can (eventually) end the derivation using the While-false rule, no such derivation can exist.

Exercise 3.5

The associativity does not matter: $(\mathrm{p}_1; \mathrm{p}_2); \mathrm{p}_3$ and $\mathrm{p}_1; (\mathrm{p}_2; \mathrm{p}_2)$ will always behave the same. More formally, for all states σ and σ', $\langle (\mathrm{p}_1; \mathrm{p}_2); \mathrm{p}_3, \sigma \rangle \to \sigma'$ if and only if $\langle \mathrm{p}_1; (\mathrm{p}_2; \mathrm{p}_3), \sigma \rangle \to \sigma'$. To see why this holds, we will prove the implication in one direction. Consider the possible derivations of $\langle (\mathrm{p}_1; \mathrm{p}_2); \mathrm{p}_3, \sigma \rangle \to \sigma'$; these must use the rule for sequential composition twice. As a result, it has the following form:

$$\dfrac{\dfrac{\langle\, \mathrm{p}_1 \,,\, \sigma \,\rangle \to \tau_1 \qquad \langle\, \mathrm{p}_2 \,,\, \tau_1 \,\rangle \to \tau_2}{\langle\, \mathrm{p}_1; \mathrm{p}_2 \,,\, \sigma \,\rangle \to \tau_2}\,\text{Seq} \qquad \dfrac{}{\langle\, \mathrm{p}_3 \,,\, \tau_2 \rangle \to \sigma'}}{\langle\, (\mathrm{p}_1; \mathrm{p}_2); \mathrm{p}_3 \,,\, \sigma \,\rangle \to \sigma'}\,\text{Seq}$$

But then we can always construct the following derivation:

$$\dfrac{\langle\, \mathrm{p}_1 \,,\, \sigma \rangle \to \tau_1 \qquad \dfrac{\langle\, \mathrm{p}_2 \,,\, \tau_1 \,\rangle \to \tau_2 \qquad \langle\, \mathrm{p}_3 \,,\, \tau_2 \,\rangle \to \sigma'}{\langle\, \mathrm{p}_2; \mathrm{p}_3 \,,\, \tau_1 \,\rangle \to \sigma'}\,\text{Seq}}{\langle\, \mathrm{p}_1; (\mathrm{p}_2; \mathrm{p}_3) \,,\, \sigma \,\rangle \to \sigma'}$$

This shows that the placement of the parentheses does not impact the semantics of such expressions.

In general, however, $\mathrm{p}_1; \mathrm{p}_2$ and $\mathrm{p}_2; \mathrm{p}_1$ are not the same. Consider for example:

x := x + 1 ; x := 3

and

x := 3 ; x := x + 1

In a state where x is initially 0, these two programs behave differently.

Exercise 3.6

1. $3 + \mathrm{y} \geqslant 3$

2. $x + (x + 2) \geqslant 3$
3. $x + y \geqslant 3$
4. $x + z \geqslant 3$
5. $0 + 0 \geqslant 0$
6. $y + y \geqslant y$
7. $x + x \geqslant x$

Exercise 3.7

1. {x + 1 ⩾ 10} x := x + 1 {x ⩾ 10} or {x ⩾ 9} x := x + 1 {x ⩾ 10}
2. {y + 1 ⩾ y } x := y + 1 {x ⩾ y} or { true } x := y + 1 {x ⩾ y}
3. {y + 1 ⩽ y} x := y + 1 {x ⩽ y} or { false } x := y + 1 {x ⩽ y}
4. {x + 1 = x + 1} x := x + 1 {x = x} or { true } x := x + 1 {true}

Exercise 3.8

Assume $\{P'\}\ p\ \{Q\}$ holds and $P \Rightarrow P'$. Then we know that if σ satisfies P' and execution terminates, then Q will hold. If we start from some initial state σ that satisfies P, our second assumption guarantees that P' also holds. Hence, from our first assumption, we know that if p terminates, Q will hold in the final state. Hence we conclude $\{P\}\ p\ \{Q\}$.

Exercise 3.9

If we assume that the following statement holds:

$$\{\,P\,\}\ \ \text{if } b \text{ then } p_1 \text{ else } p_2\ \ \{R\}$$

It must be constructed using a derivation using the if-rule as follows:

$$\frac{\{\,P \wedge b\,\}\ \ p_1\ \ \{Q\} \qquad \{\,P \wedge \neg b\,\}\ \ p_2\ \ \{Q\}}{\{\,P\,\}\ \ \text{if } b \text{ then } p_1 \text{ else } p_2\ \ \{\,Q\,\}}\ \text{If}$$

By swapping the branches and negating the guard, we can construct the following derivation:

$$\frac{\{\,P \wedge \neg b\,\}\ \ p_2\ \ \{Q\} \qquad \{\,P \wedge \neg\neg b\,\}\ \ p_1\ \ \{Q\}}{\{\,P\,\}\ \ \text{if } \neg b \text{ then } p_2 \text{ else } p_1\ \ \{\,Q\,\}}\ \text{If}$$

Now note that $\neg\neg b$ is equivalent to b, which gives us the required proof.

Exercise 3.10

1. $\{ y \geqslant 10 \}\ y := x; z := y\ \{z \geqslant 10\}$
2. $\{ y \geqslant 5 \}\ x := y; z := x + 5\ \{z \geqslant 10\}$
3. $\{ y \geqslant 5 \}\ x := y; z := y + 5\ \{z \geqslant 10\}$
4. $\{ y + z \geqslant 7 \}\ x := y + 3; z := z + x\ \{z \geqslant 10\}$

Exercise 3.11

Consider the following (incomplete) Hoare triple:

$\{ 5 \geqslant y \}\ \ x := 5\ \ \{ ? \}$

We don't know what the postcondition should be. For example, either of the following might be possible:

- $\{ x \geqslant y \}$
- $\{ 5 \geqslant y \}$

Without further information, we do not know which of these is preferable.

Exercise 3.12

Similar to our previous argument, we can show that for any precondition P and postcondition S, we have $\{P\}\ (p_1; p_2); p_3\ \{S\}$ if and only if $\{P\}\ p_1; (p_2; p_3)\ \{S\}$. To see why this is the case, consider the a possible derivation of $\{P\}\ (p_1; p_2); p_3\ \{S\}$:

$$\dfrac{\dfrac{\{ P \}\ \ p_1\ \ \{R\} \qquad \{ R \}\ \ p_2\ \ \{Q\}}{\{ P \}\ \ p_1; p_2\ \ \{ Q \}}\ \textsf{Seq} \qquad \{ Q \}\ \ p_3\ \ \{S\}}{\{ P \}\ \ (p_1; p_2); p_3\ \ \{ S \}}\ \textsf{Seq}$$

We can always reorganize this proof as follows:

$$\dfrac{\{ P \}\ \ p_1\ \ \{R\} \qquad \dfrac{\{ R \}\ \ p_2\ \ \{Q\} \qquad \{ Q \}\ \ p_3\ \ \{S\}}{\{ R \}\ \ p_2; p_3\ \ \{ S \}}\ \textsf{Seq}}{\{ P \}\ \ (p_1; p_2); p_3\ \ \{ S \}}\ \textsf{Seq}$$

Exercise 3.16

1. { false } p {P} makes a statement about all the states that satisfy the precondition false. As no such states exist, the statement is trivially true.
2. { P } p {true} guarantees that if p terminates in a state σ', the postcondition true will hold in σ'. As the postcondition true always holds, this statement is trivially true.

4 References

Gentzen, G. 1935. “Untersuchungen über Das Logische Schließen I.” *Mathematische Zeitschrift* 39: 176–210. http://eudml.org/doc/168546.

Keller, Gabriele, and Liam O’Connor-Davis. 2019. “Inference Rules and Induction.” Concepts of Programming Languages lecture notes. http://www.cs.uu.nl/docs/vakken/mcpd/.

Moller, Faron, and Georg Struth. 2013. *Modelling Computing Systems: Mathematics for Computer Science*. Undergraduate Topics in Computer Science. Springer.

Nielson, Hanne Riis, and Flemming Nielson. 2007. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer.

Pfenning, Frank. 2004. “Natural Deduction.” Automated Theorem Proving lecture notes. https://www.cs.cmu.edu/~fp/courses/atp/handouts.html.

Van Dalen, Dirk. 1994. *Logic and Structure*. Springer.