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 part of *Modelling Computing Systems* [@modelling]; 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* [@semantics], Frank Pfenning's [-@pfenning] lecture notes on natural deduction, and Gabriele Keller and Liam O'Connor-Davis's [-@keller] lecture notes on inference rules and rule induction.

Wouter Swierstra

February 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 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 $w \in W$, $\varepsilon \leqslant w$;
- 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 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 "lego piece" 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 isPalindrom(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 is the set of all words built from the open bracket character "[' and closing bracket character ']". Now consider the words over this alphabet, 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

TODO: add further exercises and solutions

Define isEven

Define less than or equal prove that 2 < 4

Define parity check

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.

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?

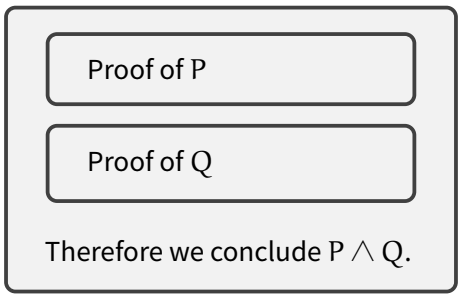
We can define the set of propositional logical formulas over a set of atomic variables P using the following BNF equation:

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

This equation enables us to distinguish between those strings of symbols that correspond to well-formed formulas, such as $\mathrm{p} \vee \neg \mathrm{q}$, and those that do not, such as $\neg \vee \vee \mathrm{p}$. But this does not yet tell us why a formula such as $\mathrm{p} \Rightarrow \mathrm{p}$ is always true, but $\mathrm{p} \vee \mathrm{p}$ 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 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:

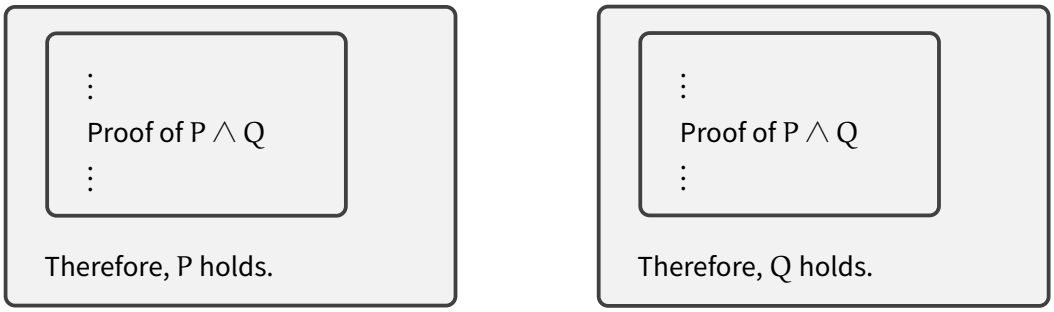
$$\frac{\text{isTrue}(P) \qquad \text{isTrue}(Q)}{\text{isTrue}(P \wedge Q)} \wedge\text{-I}$$

In this style, we can try to define a unary relation isTrue on the formulas of propositional logic; 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:

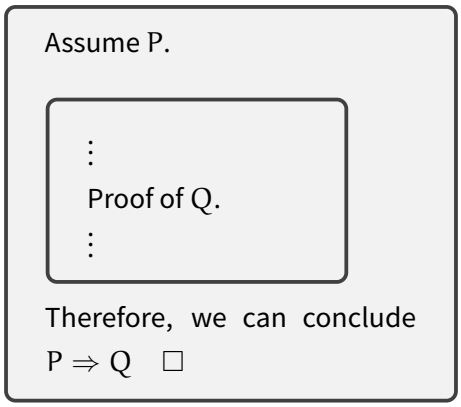
$$\frac{P \wedge Q}{P} \wedge\text{-E}_1 \qquad\qquad \frac{P \wedge Q}{Q} \wedge\text{-E}_2$$

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, written $\Gamma \vdash p$ that states that we can find a proof of p from the 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*. In the rest of this section, we will complete the definition of a relation on $\Gamma \times P$, written as $\Gamma \vdash P$, that 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. 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. This is the essence of formal proof.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{P \wedge Q \in P \wedge Q}{P \wedge Q \vdash P \wedge Q} \text{Assumption}}{P \wedge Q \vdash Q} \wedge\text{-}E_2 \qquad \dfrac{\dfrac{P \wedge Q \in P \wedge Q}{P \wedge Q \vdash P \wedge Q} \text{Assumption}}{P \wedge Q \vdash P} \wedge\text{-}E_1}{P \wedge Q \vdash Q \wedge P} \wedge\text{-}I}{\vdash P \wedge Q \Rightarrow Q \wedge P} \Rightarrow\text{-}I$$

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

$$\frac{\dfrac{}{P \wedge Q \vdash P \wedge Q}}{P \wedge Q \vdash Q} \wedge\text{-}E_2$$

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:

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

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:

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

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 by appear correct is by abusing assumptions. For example, here is an incorrect derivation showing $\vdash\ (P \Rightarrow P) \Rightarrow P$

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

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:

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

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.

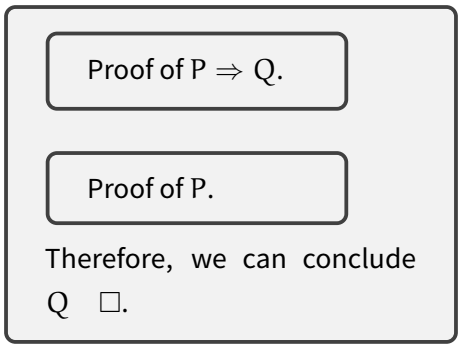
Exercise 2.2

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 [-@gentzen]. 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 ordifferent 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:

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

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

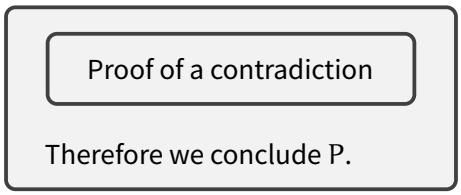
Most logic textbooks use $\top$ and $\bot$ rather than true and false respectively. We use the same notation in our inference rules. To give semantics to $\top$ and $\bot$, we need to define their elimination and introduction rules. It turns out we only need *two* rules to define both connectives.

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 $P \Rightarrow \bot, P \vdash Q$:

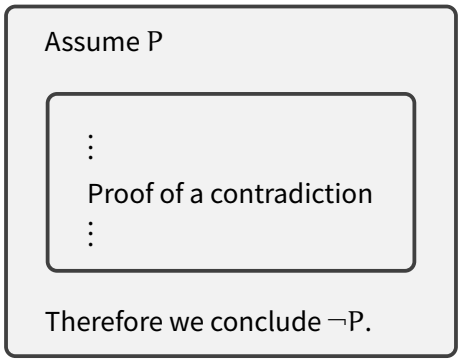
$$\dfrac{\dfrac{\overline{P \Rightarrow \bot, P \vdash P} \qquad \overline{P \Rightarrow \bot, P \vdash P \Rightarrow \bot}}{P \Rightarrow \bot, P \vdash \bot}\ \Rightarrow\text{-E}}{P \Rightarrow \bot, P \vdash Q}\ \bot\text{-E}$$

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 $P \Rightarrow Q \wedge P \Rightarrow P$, 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 $\vdash P \wedge \top \Leftrightarrow P$.

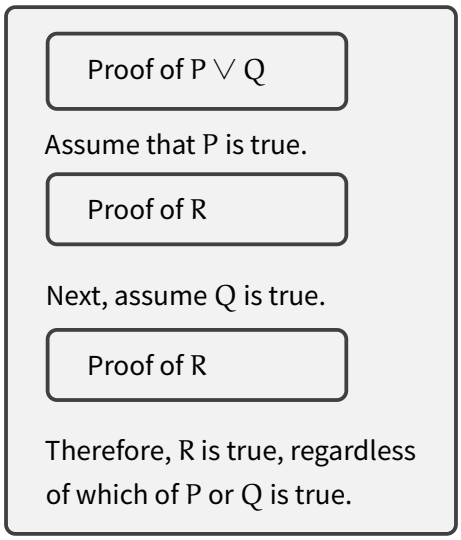
Rules for disjunction

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 or 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* of P or Q holds.

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 $P \vee Q \vdash Q \vee P$:

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

Exercise 2.6

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

Reductio ad absurdum

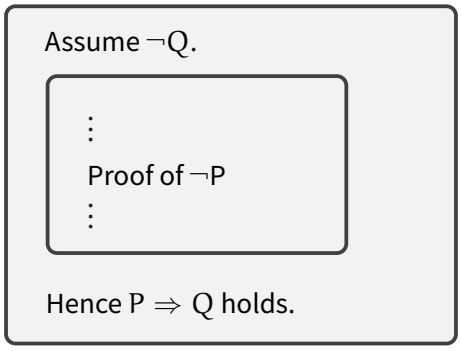
To complet our natural deduction rules for classical propositional logic, we need one final rule:

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

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—as we will discuss at the end of this chapter.

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:

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

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.

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 true/false value.

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 value of p and q, we can check the corresponding row to see the value of the entire proposotional formula.

Can we make this more precise?

We call a function $\nu : \mathrm{P} \to$ **Bool** a *truth assignment*.

Such a function chooses the values of associated with each atomic propositional variables.

Claim Given any truth assignment ν and propositional logic formula p, we can calculate the truth value of a p.

Booleans vs propositions

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

Any propositional logic formula p gives rise to a semantics:

$$[\![\mathrm{p}]\!] : (\mathrm{P} \to \textbf{Bool}) \to \textbf{Bool}$$

Before giving the semantics of propositional logic formulas, what are the booleans?

Bool, &&, ||, not w

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

We can do this by induction on p. Recall that the propositional logic formulas are given by the following BNF:

- if p is true, we return **T**;
- if p is false, we return **F**;
- if p is of the form $\neg \mathrm{q}$, we can compute the value associated with q. If this is **T**, we return **F**; if it is **F**, we return **T**.
- if p is of the form $\mathrm{q}_1 \wedge \mathrm{q}_2$, we can compute the value associated with q_1 and q_2. If this both are **T**, we return **T**; otherwise we return **F**.
- if p is of the form $\mathrm{q}_1 \vee \mathrm{q}_2$, we can compute the value associated with q_1 and q_2. If this both are **F**, we return **F**; otherwise we return **T**.
- similar cases exist for implication and logical equivalence.
- but what about variables?
- if p is an atomic propositional variable P, we return $\nu(\mathrm{P})$.

Our truth assignment tells us exactly how to treat atomic propositions.

This defines the semantics of all propositional logic formulas, usually written $[\![\mathrm{p}]\!]$.

$$[\![\mathrm{p}]\!] : (\mathrm{P} \to \textbf{Bool}) \to \textbf{Bool}$$

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.

But what does this have to do with truth tables?

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.

Suppose I'm teaching a class with 5 students

S = {Alice, Bob, Carroll, David, Eve }.

I can define a functions `marks` mapping $S \to \{1..10\}$ by giving each student their mark:

marks(Alice) = 8

marks(Bob) = 6

marks(Carroll) = 7

…

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*.

For any formula p, there are $2^{|\mathrm{fv}(\mathrm{p})|}$ possible truth assignments for the free variables in p.

Hence, you can give the semantics for p, that is the function:

$$[\![\mathrm{p}]\!] : (\mathrm{P} \to \textbf{Bool}) \to \textbf{Bool}$$

as a truth table with $2^{|\mathrm{fv}(\mathrm{p})|}$ rows.

Truth tables are simply the tabulation of this semantics.

Relating natural deduction and semantics

In the previous section, we showed how to assign any propositional logic formula p the following semantics:

$$[\![\mathrm{p}]\!] : (\mathrm{P} \to \textbf{Bool}) \to \textbf{Bool}$$

This captures the “truth table semantics” with which started the course. 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?

We would expect that for each derivation $\Gamma \vdash \mathrm{p}$,

And can we be sure that we haven’t left out any inference rules?

Given an truth assignment ν we write $\nu \models \mathrm{p}$ if $[\![\mathrm{p}]\!]\nu = \mathbf{T}$.

If for all truth assignments ν, we have $\nu \models \mathrm{p}$ we say that $\models \mathrm{p}$ (and p is a tautotology).

It turns out that natural deduction inference rules above 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 of p consists of only **T**.

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

The proofs of soundness and completeness are a subject of a more advanced course on formal logic…

…but in principle you have the reasoning techniques to understand them.

- Soundness is relatively easy to show: given a derivation of some formula p, we can do induction on this derivation. If we can show each of our inference rules is safe to use, we can trust each proof built using them.
- Completeness is harder: we don't have a derivation to do induction on; instead we need to create a derivation for some arbitrary formula p… The proof of completeness is usually much harder; the lecture notes from last year give one proof, going via a Hilbert-style proof system.

These results show just how clean and simple propositional logic is…

But they break down as soon as you study richer predicate logics…

I've presented the rules for propositional logic – but we can extend these rules to handle *predicate* logic.

Rather than introduce a more complicated system for natural deduction for handling quantifiers, I'd rather relate the natural deduction rules to truth tables…

Curry Howard

The rules of natural deduction: an overview

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercises

Exercise 2.11

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

Exercise 2.12

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

Exercise 2.13

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

Exercise 2.14

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

Exercise 2.15

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

Exercise 2.16

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

Exercise 2.17

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

Exercise 2.18

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

Exercise 2.19

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

Exercise 2.20

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

Exercise 2.21

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

Exercise 2.22

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

Solutions to 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

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

Exercise 2.3

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

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

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

Exercise 2.7

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

Exercise 2.8

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

TODO - fix overfull hbox

Exercise 2.9

TODO

Exercise 2.10

TODO

3 Reasoning about programs

We have already seen the syntax of a (toy) programming language, While – but what is its semantics?

Semantics of expressions

$$e \quad ::= \quad n \mid x \mid e + e \mid e \times e \mid \dots$$

$$b \quad ::= \quad \text{true} \mid \text{false} \mid b_1 \,||\, b_2 \mid b_1 \,\&\&\, b_2 \mid e_1 < e_2 \mid \dots$$

Idea We can write a pair of inductively defined functions that take *syntax*, evaluate it to a number or boolean.

But – this doesn't quite work: what is the value of `x + 3`?

. . .

This depends on the last value we assigned to the variable `x` – we need to keep track of the computer's memory.

Memory

We can mode the contents of the computer's memory as a function $V \to$ **Int** this function tells us for each variable in V what its current value is.

We can use this function to write a pair of inductively defined functions that take *syntax*, evaluate it to a number or boolean.

$$[\![e]\!] : (V \to \textbf{Int}) \to \textbf{Int} \qquad [\![b]\!] : (V \to \textbf{Int}) \to \textbf{Bool}$$

Just as we saw for the semantics of propositional logic, we use this function to associate meaning with variables.

Example

Previously we didn't know the meaning of `x + 3` – but what if we are given the current memory $\sigma : V \to$ **Int** and we know that $\sigma(x) = 7$:

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

We can compute the integer associated with expressions and the boolean value associated with boolean expressions provided we know the current *state* of the computer's memory.

Semantics for programs

$$\begin{aligned} p \quad ::= \quad & x := e \\ & \mid p_1; p_2 \\ & \mid \text{if } b \text{ then } p_1 \text{ else } p_2 \\ & \mid \text{while } b \text{ do } p \end{aligned}$$

How should I define a semantics?

A statement such as:

$$x := 17$$

doesn't return any interesting result – but rather *modifies the state of our program*

...

Any semantics for our language should carefully describe how the state changes…

Example execution

```
x := 3;
p := 0;
i := 1;
while (i <= x)
{
    p := p+i;
    i := i+1;
}
```

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 which satisfies:

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

Now let's run this program step by step…

This gives some idea of how a program is executed.

But this example raises some interesting questions:

- What would have happened if we would have used a different initial state? Would the results have been the same?
- Does every program terminate in a finite number of steps?
- Can our program "go wrong" somehow – dividing by zero or accessing unallocated memory?

My goal isn't to answer all these questions – but just to highlight the kind of issues you need to address when making the semantics of programming languages precise.

Let's try to give a mathematical account of program execution.

Modelling state

We model the current state of our computer's memory (storing the value of all our variables) as function:

$$\sigma : V \to \textbf{Int}$$

If we want to know the value of a given variable x, we can simply look it up $\sigma(x)$;

We will sometimes also need to *update* the current memory.

We write $\sigma[x \mapsto n]$ for the memory that is the same as σ for all variables in V **except** x, where it stores the value n.

In other words, this updates the current memory at one location, setting the value for x to n.

The meaning of our programs

Using the *inference rule notation* from the previous lecture, we can formalize the semantics of our language.

The key idea is that we define a relation on (While $\times$ State) $\times$ (While $\times$ State) – that is given the current state of the computer's memory and the program that we're executing, this relation determines the next state and remaining program to execute…

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.

Notation

We will write use the following notation:

$\langle$ p , σ $\rangle \rightarrow \langle$ p' , σ' $\rangle$

To mean that the program p running with the current state σ can perform a single step of execution, yielding a new state σ' and remaining program to execute p'.

If running p for one step causes our program to terminate we write:

$\langle$ p , σ $\rangle \rightarrow \sigma$'

To mean the program p running in the state σ terminates in one step, producing the final state σ'.

This relation gives an **operational semantics** for our programs, describing how to execute a program step by step.

Operational semantics

$$\mathrm{p} \quad ::= \quad \mathrm{x} := e \quad | \quad \mathrm{p}_1; \mathrm{p}_2 \quad | \quad \text{if } \mathrm{b} \text{ then } \mathrm{p}_1 \text{ else } \mathrm{p}_2 \text{ fi} \quad | \quad \text{while } \mathrm{b} \text{ do } \mathrm{p} \text{ end}$$

We have four language constructs – we'll only need very few rules to describe their behaviour (in contrast to, say, natural deduction rules for propositional logic).

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

Assignment

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

There is one rule for handling assignment.

The assignment statement always terminates in one step.

Starting in the state σ, executing $\mathrm{x} := e$ produces a new state, σ[x $\mapsto$ n], that updates the memory location for x to store the value of e.

For example, given a state σ satisfying $\sigma(\mathrm{y}) = 3$, we can execute the command $\mathrm{x} := \mathrm{y} + 2$ by:

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

Conditionals

$$\frac{[\![b]\!]_\sigma = true}{\langle \text{ if } b \text{ then } p_1 \text{ else } p_2 \text{ fi }, \sigma \rangle \rightarrow \langle \, p_1 \, , \sigma \, \rangle} \text{ If-true}$$

$$\frac{[\![b]\!]_\sigma = false}{\langle \text{ if } b \text{ then } p_1 \text{ else } p_2 \text{ fi }, \sigma \rangle \rightarrow \langle \, p_2 \, , \sigma \, \rangle} \text{ If-false}$$

There are two rules for evaluating if-then-else statements:

- if the guard b is true, we continue evaluating the then branch, leaving the state unchanged;
- if the guard b is false, we continue evaluating the else branch, leaving the state unchanged;

Example

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:

$$\frac{[\![x < y]\!]_\sigma = true}{\langle \text{ if } x < y \text{ then } r := x \text{ else } r := y \text{ fi }, \sigma \rangle \rightarrow \langle \, r := x \, , \sigma \, \rangle} \text{ If-true}$$

$$\frac{[\![x < y]\!]_\sigma = true}{\langle \, r := x \, , \sigma \, \rangle \rightarrow \sigma[r \mapsto 3]} \text{ Assignment}$$

Notation

It's often clear enough which rule is being applied.

For reasons of space, I may sometimes write:

$\langle \text{ if } x < y \text{ then } r := x \text{ else } r := y \text{ fi }, \sigma \rangle \rightarrow \langle \, r := x \, , \sigma \, \rangle \rightarrow \sigma[r \mapsto 3]$

In other words, our original progam halts in the state where r has become 3.

Sequential composition

$$\frac{\langle\, \mathrm{p}_1 \,,\, \sigma \,\rangle \to \sigma'}{\langle\, (\mathrm{p}_1; \mathrm{p}_2) \,,\, \sigma \,\rangle \to \langle\, \mathrm{p}_2 \,,\, \sigma' \,\rangle} \text{ seq-a}$$

$$\frac{\langle\, \mathrm{p}_1 \,,\, \sigma \,\rangle \to \langle\, \mathrm{p}_1', \sigma' \,\rangle}{\langle\, (\mathrm{p}_1; \mathrm{p}_2) \,,\, \sigma \,\rangle \to \langle\, (\mathrm{p}_1'; \mathrm{p}_2) \,,\, \sigma' \,\rangle} \text{ seq-b}$$

There are two rules for sequential composition:

- if the first program, p_1, stops after one step in the state σ', we continue executing the second program p_2 from σ';
- otherwise, we continue evaluating the remaining program p_1' until it is done.

Sequential composition

$$\frac{\langle\, \mathrm{p}_1 \,,\, \sigma \,\rangle \to \sigma'}{\langle\, (\mathrm{p}_1; \mathrm{p}_2) \,,\, \sigma \,\rangle \to \langle\, \mathrm{p}_2 \,,\, \sigma' \,\rangle} \text{ seq-a}$$

$$\frac{\langle\, \mathrm{p}_1 \,,\, \sigma \,\rangle \to \langle\, \mathrm{p}_1', \sigma' \,\rangle}{\langle\, (\mathrm{p}_1; \mathrm{p}_2) \,,\, \sigma \,\rangle \to \langle\, (\mathrm{p}_1'; \mathrm{p}_2) \,,\, \sigma' \,\rangle} \text{ seq-b}$$

There are two rules for sequential composition:

- if p_1 is done in one step (like an assignment) – we'll generally use the first rule;
- if p_1 needs more steps, like the if-then-else rules or loops, we'll use the second rule.

Loops

$$\frac{[\![\mathrm{b}]\!]_\sigma = \mathit{false}}{\langle\, \text{while } \mathrm{b} \text{ do } \mathrm{p} \text{ od} \,,\, \sigma \,\rangle \to \sigma} \text{ While-false}$$

$$\frac{[\![\mathrm{b}]\!]_\sigma = \mathit{true}}{\langle\, \text{while } \mathrm{b} \text{ do } \mathrm{p} \text{ od} \,,\, \sigma \,\rangle \to \langle\, \mathrm{p} \,;\, \text{while } \mathrm{b} \text{ do } \mathrm{p} \text{ od} \,,\, \sigma \,\rangle} \text{ While-true}$$

Just as we saw for conditionals, we need two rules to handle loops:

- if the guard b is false, we do not enter the loop body or change the state – but rather halt in the current state σ;

- if the guard b is true, we execute the loop body p, and then continue executing the main while loop.

These seven rules determine precisely how a program is executed.

Given any initial state σ and program p, we can repeatedly apply these rules to determine if the program terminates or not.

This formalizes the process I went through with large example I did at the beginning of the lecture: showing how to evaluate an example program from some initial state.

Let's extend our operational semantics to handle many execution steps

More than one step

We say a given program p and starting state σ **terminates** in τ precisely if:

- $\langle\, \mathrm{p}\,, \sigma \,\rangle \to \tau$
- or $\langle\, \mathrm{p}\,, \sigma \,\rangle \to \langle\, \mathrm{p}'\,, \sigma' \,\rangle$ and $\langle\, \mathrm{p}'\,, \sigma' \,\rangle$ terminates in τ;

Our initial example showed how the following program terminates

```
x := 3; p:= 0; i := 1;
while (i <= x) {
  p:=p+i; i:=i+1; }
```

in the state

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

by repeatedly applying the rules from our operational semantics.

From operational semantics to program logic

These operational semantics determine how a program is executed from a given initial state σ.

But consider the following mini-program:

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

Can we prove that after execution `r` will store the minimal value of `x` and `y`?

This requires reasoning about *all* possible states – rather than *one* initial state.

To perform this kind of reasoning, we need a logic to reason about **all possible** executions.

. . .

This motivates the shift from *operational semantics* to *program logic*.

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 development cost, etc.

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

Is this program doing what it should?

We will give specifications in the form of a pre- and post-condition that are predicates on our states.

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.

Notation

To define our logic for reasoning about programs, we introduce the following notation:

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

pre-condition programme post-condition

For each state σ that satisfies the precondition P,

if executing $\langle p, \sigma \rangle$ terminates in some final state τ, then τ must satisfy Q.

We'll define this – once again – using inference rules. But's let look at some examples first.

Examples

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

Unsurprising: if $x = 3$, after executing `x := x + 1`, we know $x = 4$.

- { x = A ∧ y = B} z:= x; x := y; y := z { x = B ∧ y = A}

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.

- { true } while true do p := 0 od {p = 500 }

Note that the postcondition only makes a statement about the final state. If the program never terminates, it trivially satisfies any postcondition!

Examples

{ true }

```
x := 3;
p := 0;
i := 1;
while i <= x do
      p := p + i;
      i := i+1
od
```

{ p = 6 }

How can we write a derivation proving this? What are the *inference rules* that we can use?

Hoare logic

We'll give a handful of inference rules for proving statements of the form $\{P\}\ p\ \{Q\}$.

Together these define a logic known as *Hoare logic* – named after Tony Hoare, a British computer scientist who pioneered the approach together with Edsger Dijkstra, Robert Floyd, and others.

Hoare logic – assignment

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

{ x = 3} `x := x + 1` { x = 4}

We could generalise this:

{ x = N} `x := x + 1` { x = N + 1}

. . .

But what if we want to assign another expression than x + 1?

Or what if the pre- and postcoditions are not a simple equality?

. . .

What's the most general rule?

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

- We write $Q[x\backslash e]$ for the result of replacing all the occurrences of x with e in Q.
- This rule seems backwards! It helps to read it back to front: in order for Q to hold after the assignment x := e, the precondition $Q[x\backslash e]$ should already hold.

Let's look at some examples…

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

Here are three different examples of this rule in action:

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

$$\frac{}{\{\, x = N + 1 \,\}\ x := x - 1\ \{\, x = N \,\}}\ \text{Assign}$$

$$\frac{}{\{\, x + y = V \,\}\ z := x + y\ \{\, z = V \,\}}\ \text{Assign}$$

Hoare logic – conditional

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

What happens when we execute an if statement?

We will continue executing either the "then-branch" or the "else-branch"; if both branches manage to end in a state satisfying Q, the entire if-statement will.

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

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.

. . .

Question

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

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

Hoare logic – 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 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.

Hoare logic – rule of consequence

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

If you look at this rule though, you may need to be very lucky to be able to use it: the postcondition of p_1 and precondition of p_2 must match **exactly**…

This rarely happens in larger derivations.

To still be able to use such rules, we need an additional "bookkeeping" rule.

$$\frac{P' \Rightarrow P \qquad \{P\}\ p\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ p\ \{Q'\}}\ \text{Consequence}$$

The **rule of consequence** states that we can change the pre- and postcondition provided:

- the **precondition** is **stronger** – that is, $P' \Rightarrow P$;
- the **postcondition** 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 $\langle p, \sigma \rangle$ terminates in some final state τ, then τ must satisfy Q.

Hoare logic – while

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

The general structure of the rule for loops should be along these lines:

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

But how should we fill in the question marks?

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

When we first enter the loop body, we know that P still holds.

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

After completing the loop body, we may need to execute the loop body again (and again and again and again).

The precondition of p should *continue to hold during execution*.

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

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 throughout the execution of the while loop.

Example

Give a derivation of the following statement:

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

. . .

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

Soundness and completeness

How can we be sure that we chose the right set of inference rules?

Once again, we can show that these rules are **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 $\langle\ p\ ,\ \sigma\ \rangle \to \tau$ then $Q(\tau)$

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

We can reason about all possible program behaviours using the rules of Hoare logic.

Put differently, we never need to *execute* code to prove its correctness.

From While to C#

We still need to consider a bucketload of missing features to turn our simple imperative language into a more realistic programming language:

- Classes, objects, inheritance, abstract classes, virtual methods, …
- Strings, arrays, and other richer types
- Exceptions;
- Concurrency;
- Recursion;
- Shared memory;
- Standard libraries;
- Compiler primitives;
- Foreign function interfaces;
- …

Program calculation

Problem

Given a precondition P and postcondition Q, find a program p such that $\{P\}\ p\ \{Q\}$ holds.

There is a rich field of research on **program calculation** that tries to solve this problem.

Approaches include the refinement calculus, pioneered by people such as Edsger Dijkstra, Tony Hoare, and many others.

4 References