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:

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

TODO: unclear which rule needs to be applied a priori. Sometimes it's easier to work bottom-up.

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?

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

Define isEven

Define less than or equal prove that 2 < 4

Define parity check

Solutions to exercises

Exercise 1.1

Wat denk je zelf?

Exercise 1.2

Wat denk je zelf?

Exercise 1.3

Wat denk je zelf? Dan?

2 Natural deduction

So far, we have encountered propositional logic in several lectures:

- The first lecture defined the syntax of propositional logic informally
- Later, we saw how to define this syntax formally as an inductively defined set
- We have studied the semantics of propositional logic using truth tables.
- We have seen the semantics of propositional logic informally using proof strategies

Can we not give a more precise definition of proof?

And relate it to the "truth table semantics" we saw in the first lecture?

What is a proof?

Given a formula in propositional logic p, we can check when p holds for all possible values of its atomic propositional variables – this is what we do when we write a truth table.

We can also give a "proof sketch" using proof strategies – but we haven't made precise what these strategies are, relying on an informal diagrammatic description.

Can we define a set of all proofs of some propositional logic formula?

After all, we managed to define the syntax of propositionial logic as inductively defined set – can we do the same for its semantics?

What is a proof?

Given the following set of propositional logical formulas over a set of atomic variables P:

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

Can we give inference rules that capture precisely the tautologies?

These inference rules, sometimes called *natural deduction*, formalize the proof strategies that we have seen previously.

Most logical textbooks do not introduce an explicit name for the relation capturing "truthfulness" of a given propositional logical formula, writing:

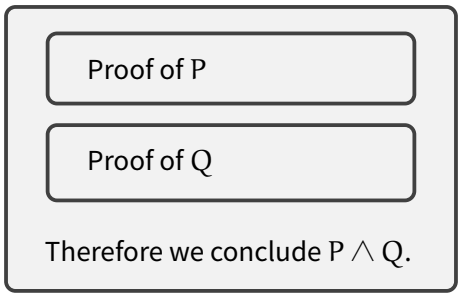
$$\frac{P \qquad Q}{P \wedge Q} \wedge\text{-I}$$

Rather than the more explicit:

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

Proof strategies vs natural deduction

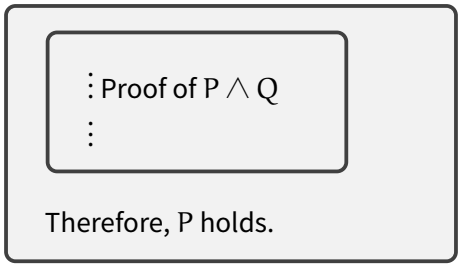
Compare the proof strategy for conjunction introduction:



And the inference rule for conjunction introduction:

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

Conjuction elimination



What is the corresponding elimination rule for conjunction?

. . .

$$\frac{P \wedge Q}{P} \wedge\text{-E}_{\text{L}}$$

Assumptions

Most textbooks in logic define natural deduction as a *unary* relation on propositional formulas.

$$\frac{P \wedge Q}{P} \wedge\text{-E}_l$$

This rule states that from the assumption $P \wedge Q$, you can deduce P.

Once you have completed a derivation, we can read off all the assumptions from the "leaves" of our proof tree.

Example derivation

Combining the rules we have seen so far, we can prove that if $P \wedge Q$ holds, so does $Q \wedge P$.

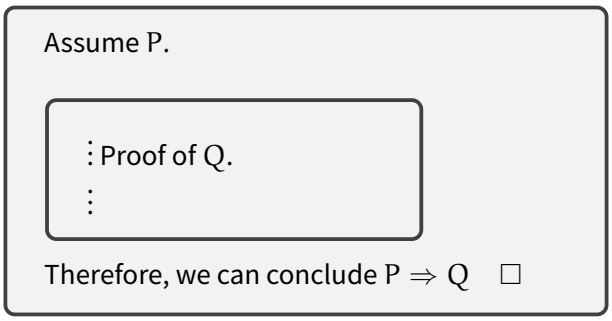
$$\dfrac{\dfrac{P \wedge Q}{Q} \wedge\text{-E}_r \qquad \dfrac{P \wedge Q}{P} \wedge\text{-E}_l}{Q \wedge P} \wedge\text{-I}$$

But how can we manage these assumptions?

Wouldn't it be nicer to show that $(P \wedge Q) \Rightarrow (Q \wedge P)$ (without making any further assumptions)?

To prove this, we need the *implication introduction* rule.

Implication introduction – proof strategy



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.

How can keep track of the assumptions in natural deduction proofs?

Assumptions

$$\dfrac{\dfrac{P \wedge Q}{Q}\,\wedge\text{-E2} \qquad \dfrac{P \wedge Q}{P}\,\wedge\text{-E1}}{Q \wedge P}\,\wedge\text{-I}$$

In the proof tree above, we have $P \wedge Q$ as *axioms* – propositions that we assume must hold.

Implication introduction – inference rule

$$\dfrac{\begin{array}{c}\overline{P^{1}}\\ \vdots \\ Q\end{array}}{P \Rightarrow Q}\,\Rightarrow\text{-I}\,1$$

The *implication introduction* rule takes a proof of Q that is built using P as assumptions.

To conclude $P \Rightarrow Q$, we *discharge* all the occurrences of P as axioms *in the current subtree*.

We number each usage of the implication introduction rule; the assumptions discharged are also numbered – indicating which rule discharged them.

Example: $P \Rightarrow P$

$$\dfrac{\overline{P^{1}}}{P \Rightarrow P}\,\Rightarrow\text{-I}\,1$$

This proof is *closed* – meaning there are no open assumptions that it is making.

Note: when using the implication elimination rule more than once, you'll need to assign a unique number to each application of this inference rule.

Example: $(P \wedge Q) \Rightarrow (Q \wedge P)$

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

. . .

$$\dfrac{\dfrac{\overline{(P \wedge Q)^{1}}}{Q}\,\wedge\text{-E2} \qquad \dfrac{\overline{(P \wedge Q)^{1}}}{P}\,\wedge\text{-E1}}{\dfrac{Q \wedge P}{(P \wedge Q) \Rightarrow (Q \wedge P)}\,\Rightarrow -\text{I}\,1}\,\wedge\text{-I}$$

Wrong proofs

The statement $(\mathrm{P} \Rightarrow \mathrm{P}) \Rightarrow \mathrm{P}$ is not true in general.

We previously saw how we "abused" proof strategies to come up with an incorrect proof.

What kind of mistakes can we make when we writing a proof using natural deduction?

. . .

$$\frac{\overline{\mathrm{P}^{1}}}{(\mathrm{P} \Rightarrow \mathrm{P}) \Rightarrow \mathrm{P}} \Rightarrow -\mathrm{I}\,1$$

Here we can make the previous mistake more explicit: we are discharging the assumption P, whereas we should be discharging $\mathrm{P} \Rightarrow \mathrm{P}$.

Implication elimination

Proof of $\mathrm{P} \Rightarrow \mathrm{Q}$.

Proof of P.

Therefore, we can conclude Q $\square$.

What is the rule for implication elimination?

. . .

$$\frac{\mathrm{P} \qquad \mathrm{P} \Rightarrow \mathrm{Q}}{\mathrm{Q}} \Rightarrow -\mathrm{E}$$

Natural deduction

We'll go through the rules for natural deduction for propositional logic.

Many of these rules closely mirror the proof strategies that we have seen previously – which is no coincidence of course.

They should be fairly familiar.

Once we've seen the rules for natural deduction proofs – we can try to relate them to the *truth table semantics* from our first lecture.

Truth and falsity

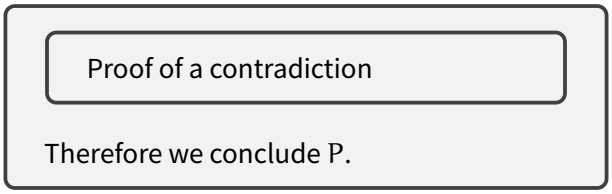
Most logic textbooks use $\top$ for **T** (truth) and $\bot$ for **F** (falsity).

The introduction rule for truth is trivial:

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

There is no introduction rule for falsity.

Falsity elimination



Or written as an inference rule:

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

Negation rules

Recall that $\neg P$ behaves just like $P \Rightarrow \bot$.

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

$$\dfrac{\dfrac{\dfrac{}{P^1}}{\vdots}}{\dfrac{\bot}{\neg P}}\ \neg\text{-I}\,1$$

Note: the negation introduction rule also discharges assumptions! Remember: keep the numbering of such rules unique throughout the entire proof tree to avoid confusion.

That is – don't use rule number 1 for both introduction introduction and negation introduction rules.

-

Equivalence rules

Similarly, $P \Leftrightarrow Q$ behaves the same as $P \Rightarrow Q \wedge P \Rightarrow P$.

$$\frac{P \Rightarrow Q \qquad P \Rightarrow Q}{P \Leftrightarrow Q} \Leftrightarrow\text{-I}$$

$$\frac{P \Leftrightarrow Q}{P \Rightarrow Q} \Leftrightarrow\text{-E}_l$$

$$\frac{P \Leftrightarrow Q}{Q \Rightarrow P} \Leftrightarrow\text{-E}_r$$

Exercise

Prove that $P \Rightarrow (Q \Rightarrow (Q \wedge P))$

. . .

$$\dfrac{\dfrac{\dfrac{\overline{Q^2} \qquad \overline{P^1}}{Q \wedge P}}{Q \Rightarrow (Q \wedge P)} \Rightarrow\text{-I } 2}{P \Rightarrow (Q \Rightarrow (Q \wedge P))} \Rightarrow\text{-I } 1$$

Discharging more than once

Consider the following proof that $P \Rightarrow (P \wedge P)$

$$\dfrac{\dfrac{\overline{P^1} \qquad \overline{P^1}}{P \wedge P}}{P \Rightarrow (P \wedge P)} \Rightarrow\text{-I } 1$$

This example shows how we need to discharge **all** the occurrences of the assumption P in the current proof subtree.

-

Exercise

Prove that $P \wedge \top \Leftrightarrow P$.

. . .

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P^1} \quad \dfrac{}{\top}\,\top\text{-I}}{P \wedge \top}\,\wedge\text{-I}}{P \Rightarrow (P \wedge \top)}\,\Rightarrow\text{-I}\,1 \qquad \dfrac{\dfrac{\dfrac{}{(P \wedge \top)^2}}{P}\,\wedge\text{-E}_l}{(P \wedge \top) \Rightarrow P}\,\Rightarrow\text{-I}\,2}{P \wedge \top \Leftrightarrow P}\,\Leftrightarrow\text{-I}$$

What's missing?

The only thing remaining are the rules for disjunction.

The *introduction* rules are easy:

. . .

$$\dfrac{P}{P \vee Q}\,\vee\text{-I}_l$$

$$\dfrac{Q}{P \vee Q}\,\vee\text{-I}_r$$

Disjuction elimination: proof strategy

Proof of $P \vee Q$

Assume that P is true.

Proof of R

Next, assume Q is true.

Proof of R

Therefore, R is true, regardless of which of P or Q is true.

Disjuction elimination

$$\dfrac{P \vee Q \qquad \begin{array}{c}\overline{P^1}\\ \vdots\\ R\end{array} \qquad \begin{array}{c}\overline{Q^1}\\ \vdots\\ R\end{array}}{R}\ \vee\text{-E } 1$$

If we know $P \vee Q$ holds…

… and we know that R holds whenever P does;

… and we know that R holds whenever Q does;

… we can conclude that R must always hold.

Exercise

Give a proof that $(P \vee \bot) \Rightarrow P$.

. . .

$$\dfrac{\dfrac{\overline{(P \vee \bot)^1} \qquad \overline{P^2} \qquad \overline{\bot^2}}{P}\ \vee\text{-E } 2}{P \vee \bot \Rightarrow P}\ \Rightarrow\text{-I } 1$$

Final rules

We need one final rule:

$$\dfrac{\dfrac{\dfrac{\overline{\neg P^1}}{\vdots}}{\bot}\ \wedge\text{-E1}}{P}\ \text{RAA}$$

This rule, sometimes called *reductio ad absurdum*, states that if $\neg P$ leads to a contradiction, P must hold.

(Notice how it is the only rule that is not an introduction-elimination rule for a logical operator?)

Beyond propositional logic...

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...

But before I can do that, let's revisit what "proof-by-truth-table" really means...

Semantics of propositional logic

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.

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

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

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

...

- 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**.

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

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

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

. . .

- 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?

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

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

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

- if p is an atomic propositional variable P, we return $\nu(\mathrm{P})$.

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

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

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

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

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:

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

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

Given any propositional logic formula p, we can assign it semantics:

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

But how is this semantics related to our natural deduction rules?

Our inference rules for natural deduction all seem perfectly "logical".

But can we be sure that any propositional formula proven using this inference rules always holds?

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

Given a set of propositional logic formulas, Γ, we will write $\Gamma \vdash \mathrm{p}$ whenever we can find a natural deduction proof of the formula p using the assumptions from Γ.

When we do not need any assumptions to show p, we write $\vdash \mathrm{p}$.

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 $\nu \models p$ we say that $\models p$ (and p is a tautotology).

It turns out that natural deduction inference rules above satisfy two important properties:

Soundness If $\vdash p$ then $\models p$. 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 $\models p$ then $\vdash p$. 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…

Exercises

Solutions to exercises

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 $V \to \mathbf{Int}$ 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.

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

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

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

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

$$\frac{\langle\, p_1 \,,\, \sigma \,\rangle \to \sigma'}{\langle\, (p_1; p_2) \,,\, \sigma \,\rangle \to \langle\, p_2 \,,\, \sigma' \,\rangle} \text{ seq-a}$$

$$\frac{\langle\, p_1 \,,\, \sigma \,\rangle \to \langle\, p_1', \sigma' \,\rangle}{\langle\, (p_1; p_2) \,,\, \sigma \,\rangle \to \langle\, (p_1'; p_2) \,,\, \sigma' \,\rangle} \text{ seq-b}$$

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

$$\frac{\langle\, p_1 \,,\, \sigma \,\rangle \to \sigma'}{\langle\, (p_1; p_2) \,,\, \sigma \,\rangle \to \langle\, p_2 \,,\, \sigma' \,\rangle} \text{ seq-a}$$

$$\frac{\langle\, p_1 \,,\, \sigma \,\rangle \to \langle\, p_1', \sigma' \,\rangle}{\langle\, (p_1; p_2) \,,\, \sigma \,\rangle \to \langle\, (p_1'; p_2) \,,\, \sigma' \,\rangle} \text{ seq-b}$$

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

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

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

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:

- $\langle p, \sigma \rangle \to \tau$
- or $\langle p, \sigma \rangle \to \langle p', \sigma' \rangle$ and $\langle p', \sigma' \rangle$ 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

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

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

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

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

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

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

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

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.

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

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