Semantics of Programming Languages - Autumn 2004

Matthew Hennessy

Course Notes by Guy McCusker

Note: Not all the topics in these notes will be covered in Autumn 2007 course

1 Introduction

As computer scientists, we are constantly talking about programs. We want to write programs that are "right", and we want to be able to describe programs to other people, without having to show them the code. Some ways in which we might describe programs are given in Slide 1.

Describing Programs

Syntax: what sequences of characters constitute programs? Grammars, lexers, parsers, automata theory...

Pragmatics: what does a given program make the computer *do*? Informal descriptions. Compilers?

Semantics: what does a program mean? When are two programs *equivalent*? When does a program satisfy its specification?

Slide 1

This course is about a formal, mathematical approach to semantics. The idea is to assign to a program a *precise* description of its meaning. A very important notion is that of equivalence between programs: we can hardly claim to know what a program means if we are not sure when two programs mean the same! Related issues include correctness of programs with respect to specifications, and the intuitively obvious notion of one program being an improvement of another.

Slide 2 gives a pair of programs that might be considered equivalent. But are they? What does that mean anyway?

Equivalent Programs?

Is the method

```
int add1(int x, int y)
{ return (x + y);
}
```

equivalent to

```
int add2(int x, int y)
{ return (y + x);
}
```

Slide 2

It turns out to be annoyingly hard to give a precise description of a program, or rather of the collection of programs in a given language, as we will see. However, it is a worthwhile activity: see Slide 3 for some reasons.

The disadvantages of reliance on informal descriptions should be obvious: descriptions in the English language are usually ambiguous and can be hard to understand. See Slide 4 for an example from the Algol 60 report [NBB$^+$63].

That description is awful, for several reasons. Here are two.

- It is gobbledygook.
- Despite its best efforts, it is imprecise: notice the word "suitable" towards the end. What is suitable is left to the reader's interpretation.

You can see people arguing about the definition of Algol 60 on `comp.compilers` to this day.

Benefits of Formal Semantics

Implementation: correctness of compilers, including optimisations, static analyses etc.

Verification: semantics supports reasoning about programs, specifications and other properties, both mechanically and by hand.

Language design: subtle ambiguities in existing languages often come to light, and cleaner, clearer ways of organising things can be discovered.

Slide 3

Informal Descriptions

An extract from the Algol 60 report:

> Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If a procedure is called from a place outside the scope of any nonlocal quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

Slide 4

1.1 Styles of Semantics

There are several different, complementary styles of formal semantics. Three of the most important are *denotational*, *operational* and *axiomatic* semantics. We shall have at least a quick look at each of these styles.

Styles of Semantics

Denotational: a program's meaning is given abstractly as an element of some mathematical structure (some kind of set).

Operational: a program's meaning is given in terms of the steps of computation the program makes when you run it.

Axiomatic: a program's meaning is given indirectly in terms of the collection of properties it satisfies; these properties are defined via a collection of axioms and rules.

Slide 5

The effort to make semantics precise has been underway since the late 1960s, and many deep and interesting discoveries have been made. However, we are a very long way from having a complete, usable theory of programming language semantics which accommodates the prevalent features of modern languages. So be warned: the languages we consider on this course will be very simplistic.

2 A Language of Expressions

Let us begin by considering a very simple language of arithmetic expressions. This example will serve to illustrate some of the ideas behind the various kinds of semantics, and will provide a simple setting in which to introduce inductive definitions and proofs slightly later on.

A grammar for a simple language *Exp* of expressions is given on Slide 9.

We all know what these things mean intuitively, and we expect that, for example, $(\mathtt{3} + \mathtt{7})$ and $(\mathtt{5} \times \mathtt{2})$ mean the same thing.

The state of the art

Denotational: successful research has focused on very simple imperative languages and vastly complex but seldom-used functional languages.

Operational: most sequential languages and some concurrent languages can be given operational semantics, but the theory tends to be hard to use.

Axiomatic: beyond simple imperative languages, little has been done.

Slide 6

This course: semantics

We will consider:

- operational, axiomatic and denotational semantics for a very simple imperative language;
- ways of proving facts about the semantics; and
- connections between the various semantics.

Slide 7

This course: maths

There will be some mathematics along the way:

- mathematical induction; and
- structural induction.

Slide 8

Syntax of *Exp*

$$E \in \mathit{Exp} ::= \mathtt{n} \mid (E + E) \mid (E \times E) \mid \cdots$$

where `n` ranges over the numerals `0`, `1`, ... We can add more operations if we need to.

We will always work with abstract syntax. That is, we assume all programs have already been parsed, so the grammar above defines syntax trees rather than concrete syntax.

Slide 9

Numbers vs Numerals Notice that we use typewriter font for the numerals, to distinguish them from the *numbers* 0, 1 and so on. The numbers are the mathematical entities which we use in everyday life, while the numerals are just syntax for describing these numbers. Thus, we can add numbers together, but not numerals; but we can write a program in *Exp* which computes the result of "adding a pair of numerals".

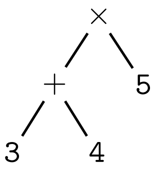
The distinction between numerals and numbers is subtle, but important, because it is one manifestation of the difference between syntax and semantics. The difference should become clearer once we have studied the various semantics which follow.

Abstract vs Concrete Syntax Saying that we are using *abstract syntax* means that all our programs are parsed *before* we start to worry about them. In this course, we will *never* be worried about where the brackets are in an expression like

$$\mathtt{3} + \mathtt{4} \times \mathtt{5}$$

because we will *never deal with such unparsed expressions.*

Using abstract syntax in fact means we're dealing with trees such as



although we use linear syntax like $((\mathtt{3} + \mathtt{4}) \times \mathtt{5})$ for them.

In this course, *brackets don't matter* because we're always using the linear syntax as a shorthand for the abstract, tree-based syntax. But of course, when there's ambiguity about what abstract syntax tree is meant by a particular piece of linear "shorthand", insert brackets if you think it will help.

Note that taking an abstract, tree-view of syntax makes it clear that +, × and so on are *program-forming operations*: they take two programs and give you a new one. One of the points of semantics, particularly denotational semantics, is to show that these operations on programs have corresponding operations on meanings.

2.1 Operational Semantics for *Exp*

An operational semantics for *Exp* will tell us how to evaluate an expression to get a natural number. This can be done in two ways:

- *small-step*, or *structural*, operational semantics gives a method for evaluating an expression step-by-step

- *big-step*, or *natural*, operational semantics ignores the intermediate steps and gives the result immediately.

Let us consider big-step semantics first. The big-step semantics for *Exp* takes the form of a relation $\Downarrow$ between expressions and *values*, which are those expressions we deem to be a “final answer”. In this case, it seems obvious that final answers should be numerals, so we will define a relation of the form

$$E \Downarrow \mathtt{n}$$

where E is an expression and $\mathtt{n}$ is a numeral.

Should I have set this up as a relation between expressions and numbers? Perhaps, but for more sophisticated languages, it will be necessary to have pieces of syntax on the right hand side of the relation, so this is a more consistent approach.

Big-Step Semantics of *Exp*

$$\text{(B-NUM)} \quad \frac{}{\mathtt{n} \Downarrow \mathtt{n}}$$

$$\text{(B-ADD)} \quad \frac{E_1 \Downarrow \mathtt{n}_1 \qquad E_2 \Downarrow \mathtt{n}_2}{(E_1 + E_2) \Downarrow \mathtt{n}_3} \; n_3 = n_1 + n_2$$

There will be similar rules for multiplication and any other operations we add.

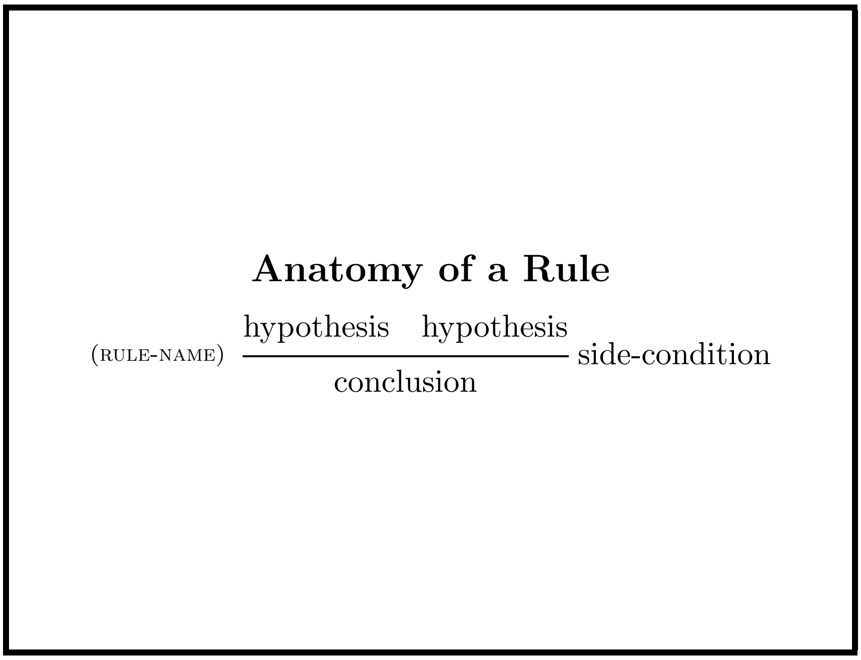
Slide 10

Notice that the side-condition for the rule for addition, (B-ADD), talks about the addition operation on *numbers* in order to define the semantics of addition of *numerals*. Note also that we are assuming that the correspondence between numerals and numbers is understood. If we wanted to give a semantics in which the numeral `3` denoted the number 918, we would have to say so somewhere! Strictly speaking we ought to say that `3` denotes 3, and so on.

What does this definition mean? Intuitively, a *rule* such as

$$\frac{E_1 \Downarrow V_1 \quad E_2 \Downarrow V_2}{E_3 \Downarrow V_3}$$

means that if it is the case that $E_1 \Downarrow V_1$ and also $E_2 \Downarrow V_2$, then it is the case that $E_3 \Downarrow V_3$. When there are no entries above the line, the rule is an *axiom*, which is to say, it always holds.



Anatomy of a Rule

$$\text{(RULE-NAME)}\ \frac{\text{hypothesis} \quad \text{hypothesis}}{\text{conclusion}}\ \text{side-condition}$$

Slide 11

The rules define a relation $\Downarrow$ which says when an expression evaluates to a final answer. $E \Downarrow \mathtt{n}$ is in this relation, or *holds*, *only* if it can be established from the axioms and rules. So, if we want to assert that, for example, $(\mathtt{3} + (\mathtt{2} + \mathtt{1})) \Downarrow \mathtt{6}$, we need to show this to be the case by applying the axioms and rules given in the definition of $\Downarrow$. A proof is given on Slide 15; note that each step in the proof is justified by a reference to a rule in the big-step semantics, from Slide 10

2.2 Small-step Semantics

The big-step semantics given above tells us what the final value of an expression is straight away. The rules give us a clue as to how to compute the answer, but sometimes it is desirable to be more explicit about exactly how programs are evaluated. A small-step semantics lets us do just this.

We shall define a relation

$$E \to E'$$

saying what you get when performing *one step* of evaluation of E. The definition is given on Slide 16.

How to Read Axioms

The axiom

$$\text{(B-NUM)} \quad \frac{}{\mathtt{n} \Downarrow \mathtt{n}}$$

says:

> for every numeral $\mathtt{n}$, it is the case that $\mathtt{n} \Downarrow \mathtt{n}$.

Notice that $\mathtt{n}$ is a kind of *variable*: you can put any numeral you like in its place. These are called *metavariables*.

Slide 12

How to Read Rules

The rule

$$\text{(B-ADD)} \quad \frac{E_1 \Downarrow \mathtt{n}_1 \qquad E_2 \Downarrow \mathtt{n}_2}{(E_1 + E_2) \Downarrow \mathtt{n}_3} \; n_3 = n_1 + n_2$$

says

> for any expressions E_1 and E_2,
> **if** it is the case that $E_1 \Downarrow \mathtt{n}_1$
> **and** if it is the case that $E_2 \Downarrow \mathtt{n}_2$
> **then** it is the case that $(E_1 + E_2) \Downarrow \mathtt{n}_3$
> where $\mathtt{n}_3$ is the numeral such that $n_3 = n_1 + n_2$.

Slide 13

Rules are Schemas

Because the Es and $\mathtt{n}$s in these rules are metavariables, each rule is really a *schema* (pattern) for an infinite collection of rules. Some of these *instances* are a bit silly, for example:

$$\frac{\mathsf{3} \Downarrow \mathsf{4} \qquad \mathsf{4} \Downarrow \mathsf{5}}{(\mathsf{3} + \mathsf{4}) \Downarrow \mathsf{9}}$$

This rule is *valid*, but is *useless*, because it is *not* the case that $\mathsf{3} \Downarrow \mathsf{4}$. That is to say, the hypotheses of the rule are not satisfied.

Slide 14

A proof that $(\mathsf{3} + (\mathsf{2} + \mathsf{1})) \Downarrow \mathsf{6}$

$$\frac{\dfrac{}{\mathsf{3} \Downarrow \mathsf{3}}\,(\textsc{b-num}) \qquad \dfrac{\dfrac{}{\mathsf{2} \Downarrow \mathsf{2}}\,(\textsc{b-num}) \qquad \dfrac{}{\mathsf{1} \Downarrow \mathsf{1}}\,(\textsc{b-num})}{(\mathsf{2} + \mathsf{1}) \Downarrow \mathsf{3}}\,(\textsc{b-add})}{(\mathsf{3} + (\mathsf{2} + \mathsf{1})) \Downarrow \mathsf{6}}\,(\textsc{b-add})$$

Slide 15

Small-step semantics of *Exp*

$$\text{(S-LEFT)}\quad \frac{E_1 \rightarrow E_1'}{(E_1 + E_2) \rightarrow (E_1' + E_2)}$$

$$\text{(S-RIGHT)}\quad \frac{E \rightarrow E'}{(\mathtt{n} + E) \rightarrow (\mathtt{n} + E')}$$

$$\text{(S-ADD)}\quad \frac{}{(\mathtt{n_1} + \mathtt{n_2}) \rightarrow \mathtt{n_3}} \text{ where } n_3 = n_1 + n_2$$

Slide 16

These rules say: to evaluate an addition, first evaluate the left hand argument; when you get to a numeral, evaluate the right hand argument; when you get to a numeral there too, add the two together to get a numeral. Note that there are *no* rules to evaluate numerals, because they have already been fully evaluated.

Consider the expression $(\mathtt{3} + (\mathtt{2} + \mathtt{1}))$. By the axiom, the rule (S-ADD), we have $\mathtt{2} + \mathtt{1} \rightarrow \mathtt{3}$, so by the second rule, (S-RIGHT), $(\mathtt{3} + (\mathtt{2} + \mathtt{1})) \rightarrow (\mathtt{3} + \mathtt{3})$. The axiom (S-ADD) also says that $(\mathtt{3} + \mathtt{3}) \rightarrow \mathtt{6}$, so we have

$$(\mathtt{3} + (\mathtt{2} + \mathtt{1})) \rightarrow (\mathtt{3} + \mathtt{3}) \rightarrow \mathtt{6}.$$

It is important to realise that the order of evaluation is fixed by this semantics, so that

$$((\mathtt{1} + \mathtt{2}) + (\mathtt{3} + \mathtt{4})) \rightarrow (\mathtt{3} + (\mathtt{3} + \mathtt{4}))$$

and *not* to $((\mathtt{1} + \mathtt{2}) + \mathtt{7})$. The big-step semantics did not, and could not, make such a stipulation.

Exercise Write down all the expressions E in the language *Exp* (with addition as the only operator) such that $E \rightarrow ((\mathtt{1} + \mathtt{2}) + \mathtt{7})$. (There are not too many of them!)

2.2.1 Getting the final answer

While the intermediate expressions of a computation are interesting, we are ultimately concerned with the final answer yielded by evaluating an expression. To capture this mathematically, we need to consider the relation which expresses multiple-step evaluations. See Slide 17.

Many steps of evaluation

Definition Given a relation → we define a new relation $\rightarrow^*$ as follows. $E \rightarrow^* E'$ holds if and only if either $E = E'$ (so no steps of evaluation are needed to get from E to E') or there is a finite sequence

$$E \rightarrow E_1 \rightarrow E_2 \cdots \rightarrow E_k \rightarrow E'.$$

This is called the *reflexive transitive closure* of →.

For our expressions, we say that `n` is the final answer of E if $E \rightarrow^* \texttt{n}$.

Slide 17

2.3 Denotational Semantics

As we have seen, operational semantics talks about how an expression is evaluated to an answer. Denotational semantics, on the other hand, has grander aspirations. A denotational model attempts to say what a piece of program text "really means".

In the case of expressions, a piece of program text "is really" a number, so we will define a function $[\![-]\!]$, such that for any expression E, $[\![E]\!]$ is a number, giving the meaning of E. Therefore, $[\![-]\!]$ will be a function from expressions to numbers, and we write

$$[\![-]\!] : \mathit{Exp} \rightarrow \mathbb{N}$$

where $\mathbb{N}$ is the set of natural numbers.

Given a model like this, $\mathbb{N}$ is called the *semantic domain* of *Exp*, which just means it is the place where the meanings live. As we come to study more complex languages, we will find that we need more complex semantic

Denotational semantics

We will define the *denotational semantics* of expressions via a function

$$[\![-]\!] : \mathit{Exp} \to \mathbb{N}.$$

Slide 18

domains. The construction and study of such domains is the subject of *domain theory*, an elegant mathematical theory which provides a foundation for denotational semantics; unfortunately domain theory is beyond the scope of this course.

For now, notice that our choice of semantic domain has certain consequences for the semantics of our language: it implies that every expression will "mean" exactly one number, so without even seeing the definition of $[\![-]\!]$, someone looking at our semantics already knows that the language is (expected to be) *normalising* (every expression has an answer) and *deterministic* (each expression has at most one answer).

It is easy to give a meaning to numerals:

$$[\![\mathtt{n}]\!] = n.$$

Note again the difference between numerals and numbers, or syntax and semantics.

For addition expressions $(E_1 + E_2)$, the meaning will of course be the sum of the meanings of E_1 and E_2:

$$[\![(E_1 + E_2)]\!] = [\![E_1]\!] + [\![E_2]\!].$$

We could make similar definitions for multiplication and so on.

We have defined $[\![-]\!]$ *by induction on the structure of expressions*: see Section 3.

Denotation of expressions

Here is the definition of our semantic function. We will see later on that this is an example of *definition by structural induction.*

$$
\begin{aligned}
[\![\mathtt{n}]\!] &= n. \\
[\![(E_1 + E_2)]\!] &= [\![E_1]\!] + [\![E_2]\!].
\end{aligned}
$$

Slide 19

2.3.1 Remarks

1. The semantic domain is entirely separate from the syntax: the set of natural numbers is a mathematical entity in its own right.

2. The meaning of a compound term like $(E_1 + E_2)$ is given in terms of the meanings of its subterms. Hence we have really given a meaning to the term forming operation $(\ldots + \ldots)$. In this case the meaning of the syntactic + is the usual addition function. We call a semantics *compositional* when it has this property, which lets us calculate meanings bit by bit, starting from the numerals and working up. Slide 20 shows an example of a calculation.

The denotational semantics for expressions is particularly easy to work with, and much less cumbersome than the operational semantics. For example, it is easy to prove simple facts such as the following.

Theorem 1 For all E_1, E_2 and E_3,

$$[\![(E_1 + (E_2 + E_3))]\!] = [\![((E_1 + E_2) + E_3)]\!].$$

Proof See Slide 21. ■

Exercise Show a similar fact using the operational semantics.

Calculating Semantics

$$\begin{aligned}
[\![(\mathsf{1} + (\mathsf{2} + \mathsf{3}))]\!] &= [\![\mathsf{1}]\!] + [\![(\mathsf{2} + \mathsf{3})]\!] \\
&= 1 + [\![(\mathsf{2} + \mathsf{3})]\!] \\
&= 1 + (2 + 3) \\
&= 6.
\end{aligned}$$

Slide 20

Associativity of addition

$$\begin{aligned}
[\![(E_1 + (E_2 + E_3))]\!] &= [\![E_1]\!] + [\![(E_2 + E_3)]\!] \\
&= [\![E_1]\!] + ([\![E_2]\!] + [\![E_3]\!]) \\
&= ([\![E_1]\!] + [\![E_2]\!]) + [\![E_3]\!] \\
&= [\![(E_1 + E_2)]\!] + [\![E_3]\!] \\
&= [\![((E_1 + E_2) + E_3)]\!].
\end{aligned}$$

Slide 21

2.4 Contextual Equivalence

We shall now introduce a very important idea in semantics, that of *contextual equivalence.*

One thing we might expect of equivalent programs is that they can be used interchangeably. That is, if $P_1 \cong P_2$ and P_1 is used in some *context*, $C[P_1]$, then we should get the same effect if we replace P_1 with P_2: we expect $C[P_1] \cong C[P_2]$.

To make this more precise, we say that a *context* $C[-]$ is a program with a *hole* where you would ordinarily expect to see a sub-program. Some contexts for *Exp* are given on Slide 22.

Some *Exp* contexts

- $C_1[-] = -$.
- $C_2[-] = (- + \mathtt{2})$.
- $C_3[-] = ((- + \mathtt{1}) + -)$.

Slide 22

Notice that the hole can appear more than once. Given any expression E we can *fill the hole* with E simply by writing E wherever the hole appears, yielding a new expression. The results of filling the holes with the expression $(\mathtt{3} + \mathtt{4})$ are given on Slide 23.

Contextual equivalence is usually defined in terms of an operational semantics. The definition in terms of big-step semantics is given on Slide 24.

For a simple language like *Exp*, contextual equivalence doesn't mean very much—it turns out that two expressions are contextually equivalent if and only if they have the same final answer. In general, though, it is a very important notion. To see this, think about the following two pieces of code which compute factorials: see Slides 25 and 26.

Filling the holes

- $C_1[(\mathtt{3}+\mathtt{4})] = (\mathtt{3}+\mathtt{4})$.
- $C_2[(\mathtt{3}+\mathtt{4})] = ((\mathtt{3}+\mathtt{4})+\mathtt{2})$.
- $C_3[(\mathtt{3}+\mathtt{4})] = (((\mathtt{3}+\mathtt{4})+\mathtt{1})+(\mathtt{3}+\mathtt{4}))$.

Slide 23

Contextual equivalence

Expressions E_1 and E_2 are *contextually equivalent* with respect to the big-step semantics if for all contexts $C[-]$ and all numerals $\mathtt{n}$,

$$C[E_1] \Downarrow \mathtt{n} \iff C[E_2] \Downarrow \mathtt{n}.$$

Slide 24

Factorial 1

```
int fact(int x) {
  int i = 1;
  int j = x;
  while (j > 0) {
  i = i * j;
  j = j - 1;
  }
  return i;
}
```

Slide 25

Factorial 2

```
int fact(int x) {
  if (x <= 0)
   { return 1; }
  else
   { return (x * fact(x - 1));}
}
```

Slide 26

These two pieces of code apparently do the same thing: they each take an integer argument and return its factorial. Whether these pieces of code are contextually equivalent or not depends on *what contexts are available*, which of course depends on the programming language.

If these two pieces of code, with syntax suitably altered, were in ML, they would indeed be equivalent. In Java, on the other hand, they are not.

Exercise Give a Java context which distinguishes these two pieces of code. (Hint: think about overriding the `fact()` method.)

2.4.1 Compositionality and Contextual Equivalence

Recall that the denotational semantics is compositional, that is, the meaning of a large phrase is built out of the meanings of its subphrases. It follows that each context determines a "function between meanings" i.e. for each $C[-]$ there is a function $f : \mathbb{N} \to \mathbb{N}$ such that

$$[\![C[E]]\!] = f([\![E]\!])$$

for any expression E.

For us, the most important consequence of this is that

$$\text{if } [\![E_1]\!] = [\![E_2]\!] \text{ then } [\![C[E_1]]\!] = [\![C[E_2]]\!] \text{ for all } C[-].$$

Therefore, if we can show something like

$$[\![E]\!] = n \iff E \Downarrow \mathtt{n}$$

we can use our semantics to reason about contextual equivalence; that is, we will know that denotationally equivalent phrases are in fact contextually equivalent. For *Exp*, this is indeed the case.

Theorem 2 For all expressions E, $[\![E]\!] = n$ if and only if $E \Downarrow \mathtt{n}$.

For more interesting languages, the relationship between operational and denotational semantics can be more subtle, but the principle of compositionality allows the denotational model to be used to reason about contextual equivalence in just the same way.

3 Induction

In the definition of the denotational semantics above, we used the principle of structural induction for abstract syntax trees. We are going to use a lot of

Our first correspondence theorem

For all expressions E, $[\![E]\!] = n$ if and only if $E \Downarrow \mathtt{n}$.

Slide 27

inductive techniques in this course, both to give definitions and to prove facts about our semantics. So, it's worth taking a little while to set out exactly what a proof by induction is, what a definition by induction is, and so on.

Very often in computer science, and (less often) in life in general, we come up against the problem of reasoning about unknown entities. For example, when designing an algorithm to solve a problem, we want to know that the result produced by the algorithm is correct, regardless of the input:

> The quicksort algorithm takes a list of numbers and puts them into ascending order.

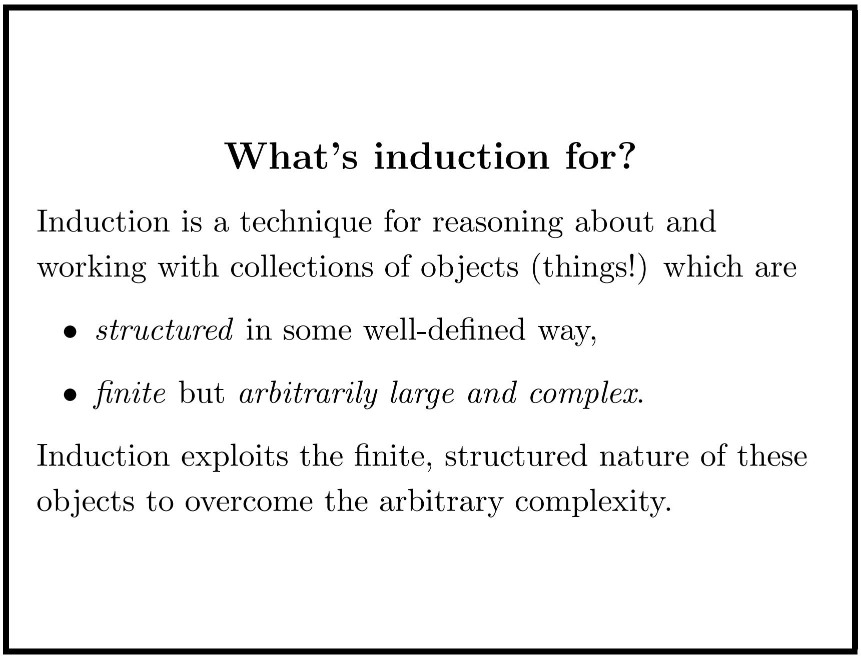
In this example, we know that the algorithm operates on *a list of numbers*, but we do not know how long that list is or exactly what numbers it contains. Similarly one may raise questions about depth-first search of a tree: how do we know it always visits all the nodes of a tree if we do not know the exact size and shape of the tree?

In examples such as these, there are two important facts about the input data which allow us to reason about arbitrary inputs:

- the input is *structured* in a known way; for example, a non-empty list has a first element and a "tail", which is the rest of the list, and a binary tree has a root node and two subtrees.

- the input is *finite.*

In this situation, the technique of *structural induction* provides a principle by which we may formally reason about arbitrary lists, trees and so on.



What's induction for?

Induction is a technique for reasoning about and working with collections of objects (things!) which are

- *structured* in some well-defined way,
- *finite* but *arbitrarily large and complex.*

Induction exploits the finite, structured nature of these objects to overcome the arbitrary complexity.

Slide 28

These kinds of structured, finite objects arise in many areas of computer science. Data structures as above are a common example, but in fact programs themselves can be seen as structured finite objects. This means that induction can be used to prove facts about *all programs in a certain language.* In semantics, we use this very frequently. We will also make use of induction to reason about purely semantic notions, such as *derivations* of assertions in the operational semantics of a language.

3.1 Mathematical Induction

The simplest form of induction is mathematical induction, that is to say, induction over the natural numbers. The principle can be described as follows.

Given a property $P(-)$ of natural numbers, to prove that $P(n)$ holds for *all* natural numbers n, it is enough to

- prove that $P(0)$ holds, and
- prove that if $P(k)$ holds for an arbitrary natural number k, then $P(k+1)$ holds too.

It should be clear why this principle is valid: if we can prove the two things above, then we know

You can use induction...

...to reason about things like

- *natural numbers*: each one is finite, but natural numbers could be arbitrarily big
- *data structures* such as trees, lists and so on
- *programs* in a programming language: again, you can write arbitrarily large programs, but they are always finite
- *derivations* of semantic assertions like $E \Downarrow \texttt{4}$: these derivations are finite trees of axioms and rules.

Slide 29

Proof by Mathematical Induction

Let $P(-)$ be a property of natural numbers. The principle of mathematical induction states that if

$$P(0) \wedge [\forall k. P(k) \implies P(k+1)]$$

holds then

$$\forall n. P(n)$$

holds.

k is the **induction parameter**

Slide 30

Writing an inductive proof

To prove that $P(n)$ holds for all natural numbers n, we must do two things.

Base Case: prove that $P(0)$ holds, any way you like

Inductive Step: let k be an arbitrary number, and assume that $P(k)$ holds. This assumption is called the *inductive hypothesis* or IH, with parameter k. Using this assumption, prove that $P(k+1)$ holds.

Slide 31

- $P(0)$ holds.
- Since $P(0)$ holds, $P(1)$ holds.
- Since $P(1)$ holds, $P(2)$ holds.
- Since $P(2)$ holds, $P(3)$ holds.
- And so on...

Therefore, $P(n)$ holds for any n, regardless of how big n is.

This conclusion can *only* be drawn because every natural number can be reached by starting at zero and adding one repeatedly. The two elements of the induction can be read as saying

- Prove that P is true at the place where you start, i.e. zero.
- Prove that the operation of adding one *preserves* P, that is, if $P(k)$ is true then $P(k+1)$ is true.

Since every natural number can be “built” by starting at zero and adding one repeatedly, every natural number has the property P: as you build the number, P is true of everything you build along the way, and it’s still true when you’ve built the number you’re really interested in.

3.1.1 Induction in Practice

So, how do we write down a proof by induction? If we need to prove $P(n)$ for all natural numbers n, we do the following:

Base case: Prove directly that $P(0)$ holds. This can be done any way you like!

Inductive Step: Prove that $P(k+1)$ holds, using the assumption that $P(k)$ holds. That is to say

> assume $P(k)$, and use this assumption to prove $P(k+1)$.

In the second step above, the assumption, $P(k)$, is called the *inductive hypothesis* or IH. The idea is that k is some natural number about which we know nothing except that $P(k)$ holds. Our task is to use only this information to show that $P(k+1)$ also holds.

Another way to think of this is

> try to *reduce* the problem of showing that $P(k+1)$ holds to the problem of showing that $P(k)$ holds.

3.1.2 Example

Here is perhaps the simplest example of a proof by mathematical induction. We shall show that

$$\sum_{i=0}^{n} i = \frac{n^2+n}{2}.$$

So here our property $P(n)$ is

> the sum of numbers from 0 to n inclusive is equal to $\dfrac{n^2+n}{2}$.

Base case: The base case, $P(0)$, is

> the sum of numbers from 0 to 0 inclusive is equal to $\dfrac{0^2+0}{2}$, which is 0.

This is obviously true, so the base case holds.

Inductive Step: Here the inductive hypothesis, IH for parameter k, is the statement $P(k)$:

> the sum of numbers from 0 to k inclusive is equal to $\dfrac{k^2+k}{2}$.

From this inductive hypothesis, with parameter k, we must prove that

> the sum of numbers from 0 to $k+1$ inclusive is equal to $\dfrac{(k+1)^2+(k+1)}{2}$.

The proof is a simple calculation.

$$\begin{array}{rcll}
\sum_{i=0}^{k+1} i & = & (\sum_{i=0}^{k} i) + (k+1) & \\
 & = & \dfrac{k^2+k}{2} + (k+1) & \text{using IH for k} \\
 & = & \dfrac{k^2+k+2k+2}{2} & \\
 & = & \dfrac{(k^2+2k+1)+(k+1)}{2} & \\
 & = & \dfrac{(k+1)^2+(k+1)}{2} &
\end{array}$$

which is what we had to prove.

3.1.3 Defining Functions and Relations

As well as using induction to prove properties of natural numbers, we can use it to define functions which operate on natural numbers.

Just as proof by induction proves a property $P(n)$ by considering the case of zero and the case of adding one to a number known to satisfy P, so definition of a function f by induction works by giving the definition of $f(0)$ directly, and building the value of $f(k+1)$ out of $f(k)$.

All this is saying is that if you define the value of a function at zero, by giving some a, and you show how to calculate the value at $k+1$ from that at k, then this does indeed define a function. This function is "unique", meaning that it is completely defined by the information you have given—there is no choice about what f can be.

Roughly, the fact that we use $f(k)$ to define $f(k+1)$ in this definition corresponds to the fact that we assume $P(k)$ to prove $P(k+1)$ in a proof by induction.

For example, Slide 33 gives an inductive definition of the factorial function over the natural numbers.

Slide 34 contains another definitional use of induction. We have already seen, in Slide 16, the effect of one computation step on expressions E from *Exp*. This is represented as a relation $E \to E'$ over expressions. Suppose

Definition by induction

You can define a function f on natural numbers by

Base Case: giving a value for $f(0)$ directly

Inductive Step: giving a value for $f(k+1)$ in terms of $f(k)$.

Slide 32

Inductive definition of factorial

- $\mathsf{fact}(0) = 1$.
- $\mathsf{fact}(k+1) = (k+1) \times \mathsf{fact}(k)$.

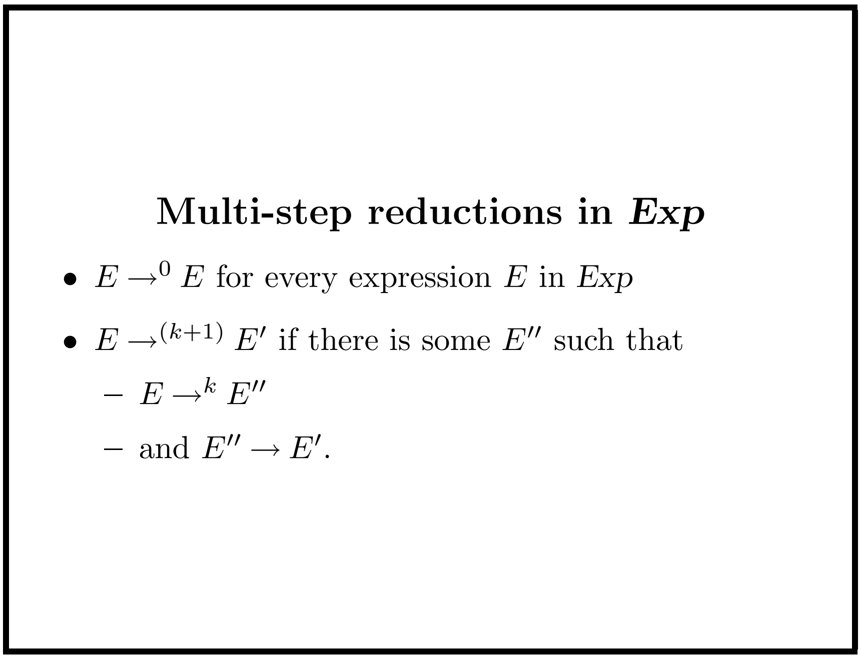
Slide 33

we wanted to define what is the effect of k reduction steps, for any natural number k. This would mean defining a family of relations $\rightarrow^k$, one for each natural number k. Intuitively $E \rightarrow^k E'$ is supposed to mean that by applying exactly k computation rules to E we obtain E'.

A formal definition of these relations can be given by induction on k. In Slide 34 we see a definition with two clauses. The first defines the relation $\rightarrow^0$ outright. In zero steps an expression remains untouched, so $E \rightarrow^0 E$, for every expression E. In the second clause the relation $\rightarrow^{(k+1)}$ is defined in terms of $\rightarrow^k$ It says that E reduces to E' in $(k+1)$ steps if

- there is some intermediary expression E'' to which E reduces to in k steps
- this intermediary expression E'' reduces to E' in one step.

The principle of induction now says that each of the infinite collection of relations $\rightarrow^k$ are well-defined.



Multi-step reductions in *Exp*

- $E \rightarrow^0 E$ for every expression E in *Exp*
- $E \rightarrow^{(k+1)} E'$ if there is some E'' such that
  - $E \rightarrow^k E''$
  - and $E'' \rightarrow E'$.

Slide 34

3.2 A Structural View of Mathematical Induction

We said in the last section that mathematical induction is a valid principle because every natural number can be “built” using zero as a starting point and the operation of adding one as a method of building new numbers from old. We can turn mathematical induction into a form of structural induction by viewing numbers as elements of the following grammar:

$$N ::= \mathsf{zero} \mid \mathsf{succ}(N).$$

Here succ, short for *successor*, should be thought of as the operation of adding one. Therefore zero represents 0, and 3 is represented by

$$\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))).$$

With this view, it really is the case that a number is built by starting from zero and repeatedly applying succ. Numbers, when thought of like this, are finite, structured objects. The structure can be described as follows.

> A number is either zero, which is indecomposable, or has the form $\mathsf{succ}(N)$, where N is another number.

(We might refer to this N as a *sub-number*, since it is a substructure of the bigger number; but in this case that nomenclature is very unusual and clumsy.)

The principle of induction now says that to prove $P(N)$ for all numbers N, it suffices to do two things.

Base case: Prove that $P(\mathsf{zero})$ holds.

Inductive Step: The IH (inductive hypothesis) is that $P(K)$ holds for some number K. From this IH prove that $P(\mathsf{succ}(K))$ also holds.

This is summarised in Slideslide:structural-mathematical-induction

3.2.1 In Practice...

Again, in practice an inductive proof looks like this:

Base case: Prove that $P(\mathsf{zero})$ holds.

Inductive Step: Assume $P(K)$ holds for some K; this is the inductive hypothesis for K. From this assumption prove that $P(\mathsf{succ}(K))$ also holds.

Note that when trying to prove $P(\mathsf{succ}(K))$, the inductive hypothesis tells us that we may assume P holds of the *substructure* of $\mathsf{succ}(K)$, that is, we may assume $P(K)$ holds.

This principle is *identical* to the one above, but written in a structural way. The reason it is valid is the same as before:

Structural view Mathematical Induction

A grammar for natural numbers

$$N ::= \mathsf{zero} \mid \mathsf{succ}(N).$$

Base case: Prove $P(\mathsf{zero})$ holds.

Inductive Step: IH is that $P(K)$ holds for some K. Assuming IH, prove that $P(\mathsf{succ}(K))$ follows.

Conclusion: $P(N)$ is true for every number N

Slide 35

- $P(\mathsf{zero})$ holds,
- so $P(\mathsf{succ}(\mathsf{zero}))$ holds,
- so $P(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))$ holds,
- so $P(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))))$ holds,
- and so on…

That is to say, we have shown that every way of building a number preserves the property P, and that P is true of the basic building block zero, so P is true of every number.

3.2.2 Defining Functions

The principle of defining functions by induction works for this representation of the natural numbers in exactly the same way as before. To define a function f which operates on these numbers, we must

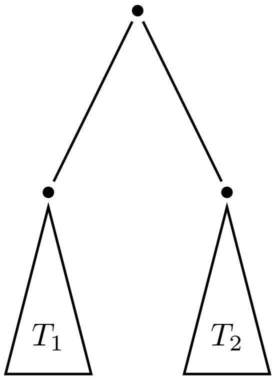
- Define $f(\mathsf{zero})$ directly
- Define $f(\mathsf{succ}(K))$ in terms of $f(K)$.

In this presentation, the definition of $f(\mathsf{succ}(K))$ looks very much like a recursive definition in ML, with the proviso that the recursive call must be to $f(K)$.

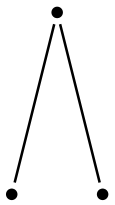
3.3 Structural Induction for Binary Trees

Binary trees are a commonly used data structure. Roughly, a binary tree is either a single *leaf node*, or a *branch node* which has two *subtrees*. That is, trees take the form

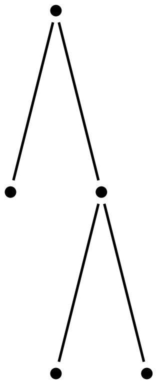
or



Here T_1 and T_2 are the two subtrees of the bigger tree. For example,



is one such *composite tree*, in which both of the subtrees are leaf nodes. Another example is



Here the left subtree is a single leaf node, while the right subtree is the simple composite tree from above.

To make it easier to talk about trees like this, let us introduce a BNF-like syntax for them, similar to that for arithmetic expressions. See Slide 36.

A syntax for binary trees

$$\mathsf{bTree} ::= \mathsf{Node} \mid \mathsf{Branch}(\mathsf{bTree}, \mathsf{bTree})$$

Note similarity with arithmetic expressions

Slide 36

In this syntax, the four trees above are written as

$$\mathsf{Node},$$

$$\mathsf{Branch}(T_1, T_2),$$

$$\mathsf{Branch}(\mathsf{Node}, \mathsf{Node}),$$

and

$$\mathsf{Branch}(\mathsf{Node}, \mathsf{Branch}(\mathsf{Node}, \mathsf{Node}))$$

respectively.

The principle of *structural induction over binary trees* states that to prove a property $P(T)$ for all trees T, it is sufficient to do the following two things:

Base case: Prove that $P(\mathsf{Node})$ holds.

Inductive Step: The inductive hypothesis IH is that $P(T_1)$ and $P(T_2)$ hold for some arbitrary trees T_1 and T_2. Then from this assumption prove that $P(\mathsf{Branch}(T_1, T_2))$ also holds.

Again, in the inductive step, we *assume* that the property holds of T_1 and T_2 and use this assumption to prove that it holds of $\mathsf{Branch}(T_1, T_2)$. The conclusion is that $P(T)$ is true for every tree T.

To put this another way: to do a proof by induction on the structure of trees, consider all possible cases of what a tree can look like. The grammar above tells us that there are two cases.

Structural Induction on Binary Trees

To prove a property $P(-)$ of all binary trees, we must do two things.

Base case: Prove that $P(\mathsf{Node})$ holds.

Inductive Step: The inductive hypothesis IH is that $P(T_1)$ and $P(T_2)$ hold for some arbitrary trees T_1 and T_2. Assuming IH prove that $P(\mathsf{Branch}(T_1, T_2))$ follows.

The conclusion is that $P(T)$ is true of all trees T.

Slide 37

- The case of Node. Prove that $P(\mathsf{Node})$ holds directly.
- The case of $\mathsf{Branch}(T_1, T_2)$. In this case, the inductive hypothesis says that *we may assume that* $P(T_1)$ *and* $P(T_2)$ *hold* while we are trying to prove $P(\mathsf{Branch}(T_1, T_2))$. We do not know anything else about T_1 and T_2: they could be any size or shape, as long as they are binary trees which satisfy P.

3.3.1 Defining Functions

Using exactly the same principle as before, we may give definitions of functions which take binary trees as their arguments, by induction on the structure of the trees.

As you can probably guess by now, to define a function f which takes an arbitrary binary tree, we must

- Define $f(\mathsf{Node})$ directly.
- Define $f(\mathsf{Branch}(T_1, T_2))$ in terms of $f(T_1)$ and $f(T_2)$.

This is summarised in Slide 38

Again this definition looks like a recursive function definition in ML, with the proviso that we may make recursive calls only to $f(T_1)$ and $f(T_2)$. That

Defining functions over binary trees

To define a function f on binary trees we must

Base Case: give a value for $f(\mathsf{Node})$ directly

Inductive Step: define $f(\mathsf{Branch}(T_1, T_2)$, using (if necessary) $f(T_1)$ and $f(T_2)$.

$f(T)$ is then defined for every tree T.

Slide 38

is to say, the recursive calls must be with the *immediate subtrees* of the tree we are interested in.

Another way to think of such a function definition is that it says how to build up the value of $f(T)$ for any tree, in the same way that the tree is built up. Since any tree can be built starting with some Nodes and putting things together using $\mathsf{Branch}(-, -)$, a definition like this lets us calculate $f(T)$ bit-by-bit.

3.3.2 Example

Here is an example of a pair of inductive definitions over trees, and a proof of a relationship between them.

We first define the function leaves which returns the number of leaf Nodes in a tree.

Base case: $\mathsf{leaves}(\mathsf{Node}) = 1$.

Inductive Step: $\mathsf{leaves}(\mathsf{Branch}(T_1, T_2)) = \mathsf{leaves}(T_1) + \mathsf{leaves}(T_2)$.

We now define another function, branches, which counts the number of $\mathsf{Branch}(-, -)$ nodes in a tree.

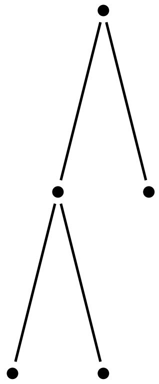
Base case: $\mathsf{branches}(\mathsf{Node}) = 0$.

Inductive Step: $\mathsf{branches}(\mathsf{Branch}(T_1, T_2)) = \mathsf{branches}(T_1) + \mathsf{branches}(T_2) + 1$.

Let us illustrate how branches works. Consider the tree

$$\mathsf{Branch}(\mathsf{Branch}(\mathsf{Node}, \mathsf{Node}), \mathsf{Node})$$

which looks like



This clearly has two branch nodes. Let us see how the function branches calculates this by building this tree up from the bottom.

First, the left sub-tree is built by taking two Nodes and putting them together with a Branch. The definition of branches says that the value on a Node is zero, while the value of a Branch is obtained by adding together the values for the things you're putting together, and adding one. Therefore, the value of branches on the left subtree is $0 + 0 + 1 = 1$.

The value of branches on the right subtree is 0, since this tree is just a Node.

The whole tree is built by putting the left and right subtrees together with a Branch. The definition of branches again tells us to add together the values for each subtree, and add one. Therefore, the overall value is $1 + 0 + 1 = 2$, as we expected.

The purpose of this discussion is of course just to show you how the value of an inductively defined function on a tree is built from the bottom up, in the same way the tree is built. You can also see it as going from the top down, in the usual way of thinking about recursively defined functions: to calculate $f(\mathsf{Branch}(T_1, T_2))$, we break the tree down into its two subtrees, calculate $f(T_1)$ and $f(T_2)$ with a recursive call, and combine the values of those in some way to get the final value.

Let us now prove, by induction on the structure of trees, that for any tree T,

$$\mathsf{leaves}(T) = \mathsf{branches}(T) + 1.$$

Let us refer to this property as $P(T)$. To show that $P(T)$ is true of all binary trees T the principle of induction says that we must do two things.

Base case: Prove that $P(\mathsf{Node})$ is true; that is that $\mathsf{leaves}(\mathsf{Node}) = \mathsf{branches}(\mathsf{Node}) + 1$.

Inductive Step: The inductive hypothesis IH is that $P(T_1)$ and $P(T_2)$ are both true, for some T_1 and T_2. So we can assume IH, namely that

$$\mathsf{leaves}(T_1) = \mathsf{branches}(T_1) + 1$$

and

$$\mathsf{leaves}(T_2) = \mathsf{branches}(T_2) + 1$$

From this assumption we have to derive $P(\mathsf{Branch}(T_1, T_2))$, namely that

$$\mathsf{leaves}(\mathsf{Branch}(T_1, T_2)) = \mathsf{branches}(\mathsf{Branch}(T_1, T_2)) + 1.$$

Proof

Base case: By definition,

$$\mathsf{leaves}(\mathsf{Node}) = 1 = 1 + \mathsf{branches}(\mathsf{Node})$$

as required.

Inductive Step: By definition,

$$\mathsf{leaves}(\mathsf{Branch}(T_1, T_2)) = \mathsf{leaves}(T_1) + \mathsf{leaves}(T_2).$$

By the inductive hypothesis IH,

$$\mathsf{leaves}(T_1) = \mathsf{branches}(T_1) + 1$$

and

$$\mathsf{leaves}(T_2) = \mathsf{branches}(T_2) + 1.$$

We therefore have

$$\mathsf{leaves}(\mathsf{Branch}(T_1, T_2)) = \mathsf{branches}(T_1) + 1 + \mathsf{branches}(T_2) + 1.$$

By definition of $\mathsf{branches}$,

$$\mathsf{branches}(\mathsf{Branch}(T_1, T_2)) = \mathsf{branches}(T_1) + \mathsf{branches}(T_2) + 1.$$

It is therefore the case that

$$\mathsf{leaves}(\mathsf{Branch}(T_1, T_2)) = \mathsf{branches}(\mathsf{Branch}(T_1, T_2)) + 1.$$

■

3.4 Structural Induction over the Language of Expressions

The syntax of our illustrative language *Exp* of expressions also gives a collection of structured, finite, but arbitrarily large objects over which induction may be used.

The syntax is given below.

$$E \in \mathit{Exp} ::= \mathtt{n} \mid (E + E) \mid (E \times E).$$

Recall that $\mathtt{n}$ ranges over the numerals $\mathtt{0}$, $\mathtt{1}$, $\mathtt{2}$ and so on. This means that in this language there are in fact an infinite number of indecomposable expressions; contrast this with the cases above, where 0 is the only "indecomposable natural number", and Node is the only indecomposable binary tree.

In the previous examples, there was only one way of building new things from old: in the case of natural numbers, we built a new one from old by adding one (applying succ); and in the case of binary trees, we built a new tree from two old ones using $\mathsf{Branch}(-,-)$.

Here, on the other hand, we can build new expressions from old in two ways: using + and using ×.

The principle of induction for expressions reflects these differences as follows. If P is a property of expressions, then to prove that $P(E)$ holds for any E, it suffices to do the following.

Base cases: Prove that $P(\mathtt{n})$ holds for every numeral $\mathtt{n}$.

Inductive Step: Here the inductive hypothesis IH is that $P(E_1)$ and $P(E_2)$ hold for some E_1 and E_2. Assuming IH we must show that both $P((E_1 + E_2))$ and $P((E_1 \times E_2))$ follow.

The conclusion will then be that $P(E)$ is true of *every* expression E.

Again, this induction principle can be seen as a case-analysis: expressions come in two forms:

- numerals, which cannot be decomposed, so we have to prove $P(\mathtt{n})$ directly for each of them; and
- composite expressions $(E_1 + E_2)$ and $(E_1 \times E_2)$, which can be decomposed into subexpressions E_1 and E_2. In this case, induction says that we may assume $P(E_1)$ and $P(E_2)$ when trying to prove $P((E_1 + E_2))$ and $P((E_1 \times E_2))$.

Structural Induction for Terms of *Exp*

To prove that property $P(-)$ holds for all terms of *Exp*, it suffices to prove

base cases: $P(\mathtt{n})$ holds for all n, and

inductive step: The inductive hypothesis IH is that $P(E_1)$ and $P(E_2)$ both hold for some arbitrary expressions E_1 and E_2. From IH we must prove that $P(E_1 + E_2)$ and $P(E_1 \times E_2)$ follow.

Slide 39

3.4.1 Example

We shall prove by induction on the structure of expressions that for any expression E, there is some numeral $\mathtt{n}$ for which $E \Downarrow \mathtt{n}$. This property is called *normalisation*: it says that all programs in our language have a final answer or so-called "normal form". It goes hand in hand with another property, called *determinacy*, which we shall prove later.

Proposition 3 (Normalisation) For every expression E, there is some n such that $E \Downarrow \mathtt{n}$.

Proof By structural induction on E. The property $P(E)$ of expressions we wish to prove is

> $P(E)$ - there is some numeral $\mathtt{n}$ such that $E \Downarrow \mathtt{n}$.

The principle of structural induction says that to prove $P(E)$ holds for every expression E we are required to establish two facts:

Base cases: $P(\mathtt{n})$ holds for every numeral $\mathtt{n}$.

> For any numeral $\mathtt{n}$, the axiom of the big-step semantics, (B-NUM), gives us that $\mathtt{n} \Downarrow \mathtt{n}$, so the property is true of every $\mathtt{n}$, as required.

Determinacy and Normalisation

Determinacy says that an expression cannot evaluate to more than one answer:

> For any expression E, if $E \Downarrow \mathtt{n}$ and $E \Downarrow \mathtt{n}'$ then $n = n'$.

Normalisation says that an expression evaluates to at least one answer:

> For every expression E, there is some n such that $E \Downarrow \mathtt{n}$.

Slide 40

Inductive Step: The inductive hypothesis IH is that $P(E_1)$ and $P(E_2)$ hold for some arbitrary E_1 and E_2. From IH we are required to prove both $P(E_1 + E_2)$ and $P(E_1 \times E_2)$ follow. We shall consider the case of $(E_1 + E_2)$ in detail; the case of $(E_1 \times E_2)$ is similar.

We must show $P(E_1 + E_2)$, namely that for some $\mathtt{n}$, it is the case that $(E_1 + E_2) \Downarrow \mathtt{n}$.

By the *inductive hypothesis* IH , we may assume that there are numerals $\mathtt{n}_1$ and $\mathtt{n}_2$ for which $E_1 \Downarrow \mathtt{n}_1$ and $E_2 \Downarrow \mathtt{n}_2$. We can then apply the rule (B-ADD) to obtain

$$\frac{E_1 \Downarrow \mathtt{n}_1 \quad E_2 \Downarrow \mathtt{n}_2}{(E_1 + E_2) \Downarrow \mathtt{n}_3}$$

where $n_3 = n_1 + n_2$. So $\mathtt{n}_3$ is the required numeral which makes $P(E_1 + E_2)$ true.

■

3.4.2 Defining Functions over Expressions

We may also use the principle of induction to define functions which operate on expressions. To define a function f which can take an arbitrary expression as an argument, we must

- define $f(\mathtt{n})$ directly, for each numeral $\mathtt{n}$,

- define $f((E_1 + E_2))$ *in terms of* $f(E_1)$ and $f(E_2)$, and
- define $f((E_1 \times E_2))$ *in terms of* $f(E_1)$ and $f(E_2)$.

Definition by Induction for *Exp*

To define a function on all terms of *Exp*, it suffices to do the following.

- define $f(\mathtt{n})$ directly, for each numeral $\mathtt{n}$,
- define $f((E_1 + E_2))$ *in terms of* $f(E_1)$ and $f(E_2)$, and
- define $f((E_1 \times E_2))$ *in terms of* $f(E_1)$ and $f(E_2)$.

Slide 41

The *denotational semantics* of *Exp* is an example of such an inductive definition. In the notes, the denotational semantic function is written as $[\![-]\!]$, but here we will write it as den to make the definition look more like the ones we have seen before.

The idea is to define, for each expression E, a number $\mathsf{den}(E)$ which is the "meaning" or in this case the "final answer" of E. The definition is a straightforward inductive one.

- $\mathsf{den}(\mathtt{n}) = n$ for each numeral $\mathtt{n}$. That is to say, $\mathsf{den}(\mathsf{0}) = 0$, $\mathsf{den}(\mathsf{7}) = 7$ and so on.
- $\mathsf{den}((E_1 + E_2)) = \mathsf{den}(E_1) + \mathsf{den}(E_2)$.
- $\mathsf{den}((E_1 \times E_2)) = \mathsf{den}(E_1) \times \mathsf{den}(E_2)$.

Again, this definition should be thought of as showing how to build up the "meaning" of a complex term as the term itself is built up from numerals and uses of + and ×.

3.5 Structural Induction over Derivations

The final example of a collection of finite, structured objects which we have seen is the collection of *proofs* of statements $E \Downarrow \mathtt{n}$ in the big-step semantics of *Exp*. In general, an operational semantics given by axioms and proof rules defines a collection of proofs of this kind, and induction is available to us in reasoning about them.

To clarify the presentation, let us refer to such proofs as *derivations* in this section.

Here is the derivation of $(\mathsf{3} + (\mathsf{2} + \mathsf{1})) \Downarrow \mathsf{6}$.

$$\frac{\dfrac{}{\mathsf{3} \Downarrow \mathsf{3}} \qquad \dfrac{\dfrac{}{\mathsf{2} \Downarrow \mathsf{2}} \qquad \dfrac{}{\mathsf{1} \Downarrow \mathsf{1}}}{(\mathsf{2} + \mathsf{1}) \Downarrow \mathsf{3}}}{(\mathsf{3} + (\mathsf{2} + \mathsf{1})) \Downarrow \mathsf{6}}$$

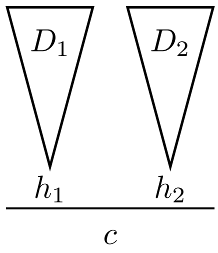
This derivation has three key elements: the *conclusion* $(\mathsf{3} + (\mathsf{2} + \mathsf{1})) \Downarrow \mathsf{6}$, and the two *subderivations*, which are

$$\frac{}{\mathsf{3} \Downarrow \mathsf{3}}$$

and

$$\frac{\dfrac{}{\mathsf{2} \Downarrow \mathsf{2}} \qquad \dfrac{}{\mathsf{1} \Downarrow \mathsf{1}}}{(\mathsf{2} + \mathsf{1}) \Downarrow \mathsf{3}}$$

We can think of a complex derivation like this as a structured object:



Here we see a derivation whose last rule is

$$\frac{h_1 \quad h_2}{c}$$

where h_1 and h_2 are the *hypotheses* of the rule and c is the *conclusion* of the rule; c is also the conclusion of the whole derivation. Since the hypotheses themselves must be derived, there are *subderivations* D_1 and D_2 with conclusions h_1 and h_2.

The only derivations which do not decompose into a last rule and a collection of subderivations are those which are simply axioms. Our principle of induction will therefore treat the axioms as the base cases, and the more complex proofs as the inductive step.

The principle of structural induction for derivations says that to prove a property $P(D)$ for every derivation D, it is enough to do the following.

Base cases: Prove that $P(A)$ holds for every axiom A. In the case of the big-step semantics, we must prove that every derivation

$$\frac{}{\mathtt{n} \Downarrow \mathtt{n}}$$

satisfies property P.

Inductive Step: For each rule of the form

$$\frac{h_1 \cdots h_n}{c}$$

prove that any derivation ending with a use of this rule satisfies the property. Such a derivation has *subderivations* with conclusions h_1, ..., h_n, and we may assume that property P holds for each of these subderivations. These assumptions form the *inductive hypothesis*.

3.5.1 Example

Let us now treat a simple example. Consider the language of expressions, restricted so that the only arithmetic operation is +.

Given a derivation in the big-step semantics, there are obvious notions of

- the number of + symbols in the conclusion of the derivation, and
- the number of rules used in the derivation.

We shall now prove that these two numbers are equal. (This is rather a pointless thing to prove, but it is mildly more straightforward than the more useful example which follows.)

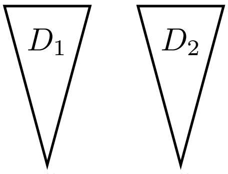
Proof

Base cases: Consider a derivation consisting just of an axiom

$$\frac{}{\mathtt{n} \Downarrow \mathtt{n}}$$

This derivation contains no rules, since it is just an axiom; and the conclusion contains no + symbols, so the two numbers in question are indeed equal.

Inductive Step: Consider a derivation which ends in a use of the rule for +. This derivation must have the form



$$\frac{E_1 \Downarrow \mathtt{n}_1 \quad E_2 \Downarrow \mathtt{n}_2}{(E_1 + E_2) \Downarrow \mathtt{n}_3}$$

The inductive hypothesis tells us that the number of rules used in D_1 is the same as the number of + symbols in E_1; and D_2 and E_2 are similarly related.

The number of + symbols in the conclusion of this derivation is clearly equal to the number in E_1, plus the number in E_2, plus 1. Similarly, the number of rules used in this derivation is the number in D_1 plus the number in D_2 plus 1. Therefore, these two numbers are identical, as required.

■

3.5.2 A Harder Example

As a second example, consider the property of *determinacy* of the big-step semantics of *Exp*.

Proposition 4 (Determinacy) For any expression E, if $E \Downarrow \mathtt{n}$ and $E \Downarrow \mathtt{n}'$ then $n = n'$.

Proof We prove this by induction on the structure of the *proof* that $E \Downarrow \mathtt{n}$. This in itself requires a little thought. The property we wish to prove is:

> For any derivation D,
> **if** the conclusion of D is $E \Downarrow \mathtt{n}$, and it is also the case that $E \Downarrow \mathtt{n}'$ is derivable,
> **then** $n = n'$.

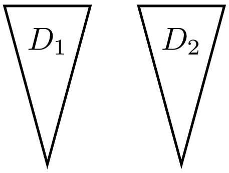
So, during this proof, we will consider

- a derivation D of a statement $E \Downarrow \mathtt{n}$, and
- another statement $E \Downarrow \mathtt{n}'$ which is *derivable*

and try to show that $n = n'$. We will apply induction to the derivation D, and *not* to the derivation of $E \Downarrow \mathtt{n}'$.

Base case: $E \Downarrow \mathtt{n}$ is an axiom. In this case, $E = \mathtt{n}$. We also have $E \Downarrow \mathtt{n}'$, that is, $\mathtt{n} \Downarrow \mathtt{n}'$. By examining the rules of the big-step semantics, it is clear that this can only be the case if $\mathtt{n} \Downarrow \mathtt{n}'$ is an axiom. It follows that $n = n'$.

Inductive step: If the proof is not an axiom, it must have the form



$$\frac{E_1 \Downarrow \mathtt{n}_1 \quad E_2 \Downarrow \mathtt{n}_2}{(E_1 + E_2) \Downarrow \mathtt{n}}$$

where $E = (E_1 + E_2)$ and $n = n_1 + n_2$. Call this whole derivation D.

The inductive hypothesis applies to the subderivations D_1 and D_2. In the case of D_1, it says

> Since D_1 has conclusion $E_1 \Downarrow \mathtt{n}_1$, if a statement $E_1 \Downarrow \mathtt{n}''$ is derivable, then we may assume that $n_1 = n''$.

We will use this in a moment.

We must show that if $E \Downarrow \mathtt{n}'$ then $n = n'$. So suppose that $E \Downarrow \mathtt{n}'$, i.e. $(E_1 + E_2) \Downarrow \mathtt{n}'$ is derivable. This could not be derived using an axiom, so it must be the case that it was derived using a rule

$$\frac{E_1 \Downarrow \mathtt{n}_3 \quad E_2 \Downarrow \mathtt{n}_4}{(E_1 + E_2) \Downarrow \mathtt{n}'}$$

where $n' = n_3 + n_4$.

This means that $E_1 \Downarrow \mathtt{n}_3$ is derivable, and $E_2 \Downarrow \mathtt{n}_4$ is derivable.

Using the inductive hypothesis as spelled out above, we may assume that $n_1 = n_3$, and by applying IH to D_2, we may assume that $n_2 = n_4$. Since we have the equations

$$n = n_1 + n_2 \quad \text{and} \quad n' = n_3 + n_4$$

it follows that $n = n'$ as required. ■

This is a tricky proof, because we do induction on the derivation of $E \Downarrow \mathtt{n}$, but we must perform some analysis on the derivation of $E \Downarrow \mathtt{n}'$ too. Make sure you understand all the previous examples before getting too worried about this one; but do attempt to understand this technique, because it crops up all the time in semantics.

3.6 Some proofs about the small-step semantics

We have seen how to use induction to prove simple facts about the big-step semantics of *Exp*. In this section we will see how to carry out similar proofs for the small-step semantics, both to reassure ourselves that we're on the right course and to make some intuitively obvious facts about our language into formal theorems.

An important property of the small-step semantics is that it is *deterministic* in a very strong sense: not only does each expression have at most one final answer, as in the big-step semantics, but also each expression can be evaluated to its final answer in exactly one way. Slide 42 shows three properties that we will prove. We will in fact give two different proofs for the first of these properties.

Properties of evaluation

For any expression E,

- if $E \rightarrow E_1$ and $E \rightarrow E_2$ then $E_1 = E_2$;
- if $E \rightarrow^* \mathtt{n}$ and $E \rightarrow^* \mathtt{n}'$ then $n = n'$; and
- there is some n such that $E \rightarrow^* \mathtt{n}$.

Slide 42

Lemma 5 If $E \rightarrow E_1$ and $E \rightarrow E_2$ then $E_1 = E_2$.

Proof In this particular proof we use structural induction on the *derivation* that $E \rightarrow E_1$.

Base case The axiom for this semantics is the case where E is $(\mathtt{n}_1 + \mathtt{n}_2)$ and E_1 is $\mathtt{n}_3$, where $n_3 = n_1 + n_2$. Consider the derivation that $E \rightarrow E_2$, that is $(\mathtt{n}_1 + \mathtt{n}_2) \rightarrow E_2$. If this derivation is just an axiom, then E_2 must be $\mathtt{n}_3$ as required. Otherwise, the last rule of this derivation is either

$$\frac{\mathtt{n}_1 \rightarrow E'}{(\mathtt{n}_1 + \mathtt{n}_2) \rightarrow (E' + \mathtt{n}_2)}$$

or

$$\frac{\mathtt{n}_2 \rightarrow E'}{(\mathtt{n}_1 + \mathtt{n}_2) \rightarrow (\mathtt{n}_1 + E').}$$

This implies that there is a derivation of $\mathtt{n}_1 \rightarrow E'$ or $\mathtt{n}_2 \rightarrow E'$, but it is easy to see that no such derivation exists. Therefore this case can't happen!

Inductive Step If $E \rightarrow E_1$ was established by a more complex derivation, we must consider two possible cases, one for each rule that may have been used last in the derivation.

1. For some E_3, E_4 and E_3', it is the case that $E = (E_3 + E_4)$ and the last line of the derivation is

   $$\frac{E_3 \rightarrow E_3'}{(E_3 + E_4) \rightarrow (E_3' + E_4).}$$

   We also know that $E \rightarrow E_2$, i.e. $(E_3 + E_4) \rightarrow E_2$. Since $E_3 \rightarrow E_3'$, E_3 cannot be a numeral, so the last line in the derivation of this reduction must have the form

   $$\frac{E_3 \rightarrow E_3''}{(E_3 + E_4) \rightarrow (E_3'' + E_4).}$$

   But the derivation of $E_3 \rightarrow E_3'$ is a subtree of that of $E \rightarrow E_1$, so our inductive hypothesis allows us to assume that $E_3' = E_3''$. It therefore follows that $E_1 = E_2$.

2. In the second case, $E = (\mathtt{n} + E_3)$ and the derivation of $E \rightarrow E_1$ has

   $$\frac{E_3 \rightarrow E_3'}{(\mathtt{n} + E_3) \rightarrow (\mathtt{n} + E_3')}$$

   as its last line. Again we know that E_3 is not a numeral, so the derivation that $E \rightarrow E_2$ must also end with a rule of the form

   $$\frac{E_3 \rightarrow E_3''}{(\mathtt{n} + E_3) \rightarrow (\mathtt{n} + E_3'').}$$

   Again we may apply the inductive hypothesis to deduce that $E_3' = E_3''$, from which it follows that $E_1 = E_2$.

■

Lemma 5 (Revisited) If $E \rightarrow E_1$ and $E \rightarrow E_2$ then $E_1 = E_2$.

Proof Here we will prove the result using structural induction on the *structure* of E.

Let $P(E)$ be the property:

> For any F_1, F_2, if $E \rightarrow F_1$ and $E \rightarrow F_2$, then $F_1 = F_2$

We prove $P(E)$ holds for every expression E by structural induction on E. There are two cases.

Base case We have to show $P(\mathtt{n})$ is true for every numeral $\mathtt{n}$.

But this is vacuously true because according to the rules of the small-step semantics of *Exp*, on Slide 16, there are no possible F_1 or F_2 such that $\mathtt{n} \rightarrow F_1$ or $\mathtt{n} \rightarrow F_2$.

Inductive case Here we assume the inductive hypothesis IH, namely that for some (unknown) E_1 and E_2 both $P(E_1)$ and $P(E_2)$ are true. Using IH we have to show that $P(E_1 + E_2)$ is true.

So suppose $E_1 + E_2 \rightarrow F_1$ and $E_1 + E_2 \rightarrow F_2$; we have to show $F_1 = F_2$. There are two possibilities.

- E_1 is a numeral, say $\mathtt{n}_1$. In this case how can $\mathtt{n}_1 + E_2 \rightarrow F_1$ be derived? Examining the rules in Slide 16 we see that there are two possibilities.
  - Suppose the rule which was applied was (S-RIGHT), giving a derivation of the form

$$\frac{E_2 \rightarrow G_1}{(\mathtt{n}_1 + E_2) \rightarrow (\mathtt{n}_1 + G_1)}\ \text{(S-RIGHT)}$$

    In other words F_1 must take the form $\mathtt{n}_1 + G_1$, for some G_1 such that $E_2 \rightarrow G_1$.

    Since there is a derivation $E_2 \rightarrow G_1$ it means that E_2 is not a numeral. This is turn means any derivation from E_2 must also use (S-RIGHT). So a similar analysis can be made of the derivation $E_1 + E_2 \rightarrow F_2$; this must take the form

$$\frac{E_2 \rightarrow G_2}{(\mathtt{n}_1 + E_2) \rightarrow (\mathtt{n}_1 + G_2)}\ \text{(S-RIGHT)}$$

where F_2 is $\mathtt{n}_1 + G_2$.

But now we can use part of the inductive hypothesis, namely $P(E_2)$; this tells us that $G_1 = G_2$. From this it follows that $F_1 = F_2$.

- The second possibility is that $\mathtt{n}_1 + E_2 \rightarrow F_1$ is derived by an application of the rule (S-ADD). So E_2 must be a numeral, say $\mathtt{n}_2$ and so F_1 is also a numeral, $\mathtt{n}_3$, where $n_1 + n_2 = n_3$.

  Now looking at the second derivation $\mathtt{n}_1 + E_2 \rightarrow F_2$, since E_2 is the numeral $\mathtt{n}_2$, again the only possible rule which can be applied is (S-ADD), with the result that F_2 is also $\mathtt{n}_3$; in other words, $F_1 = F_2$.

- So let us suppose that E_1 is not a numeral. Here, what rules from Slide 16 can be used to infer $E_1 + E_2 \rightarrow F_1$? The rules (S-RIGHT) and (S-ADD) require that E_1 be a numeral. So the only possibility is (S-LEFT), giving a derivation of the form the form

$$\frac{E_1 \rightarrow G_1}{(E_1 + E_2) \rightarrow (G_1 + E_2)} \;\text{(S-RIGHT)}$$

So F_1 must be of the form $G_1 + E_2$ for some G_1 such that $E_1 \rightarrow G_1$.

A similar analysis gives that F_2 must also be of the form $G_2 + E_2$ for some G_2 such that $E_1 \rightarrow G_2$.

Now we can apply $P(E_1)$, which is part of the inductive hypothesis, to obtain that $G_1 = G_2$. It follows that $F_1 = F_2$.

■

This result says that the one-step relation is deterministic. Let us now see how from this we can prove that there can be at most one final answer. First we show that the k-step reduction relation, defined in Slide 34, is also deterministic.

Corollary 6 For every natural number k and every expression E, if $E \rightarrow^k E_1$ and $E \rightarrow^k E_2$ then $E_1 = E_2$.

Proof We prove this by *mathematical induction* on the natural number k. So let $P(k)$ be the statement

$$E \rightarrow^k E_1 \text{ and } E \rightarrow^k E_2 \text{ implies } E_1 = E_2$$

By mathematical induction to prove $P(n)$ holds for every n we need to establish two facts.

Base case Here we establish $P(0)$, namely that if $E \rightarrow^0 E_1$ and $E \rightarrow^0 E_2$ then $E_1 = E_2$.

But this is trivial. Looking at the definition of $\rightarrow^k$ in Slide 34 we see that the only possibility for E_1 and E_2 is that they are E itself, and therefore must be equal.

Inductive case Here we assume the inductive hypothesis, namely $P(k)$. From this we must prove $P(k{+}1)$, namely that if $E{\rightarrow}^{(k+1)}E_1$ and $E{\rightarrow}^{(k+1)}E_2$ then $E_1 = E_2$.

Again looking at the definition of $\rightarrow^{(k+1)}$ in Slide 34 we know that there must exist some expressions E_1' and E_2' such that

$$\begin{aligned} E &\rightarrow^k E_1' \rightarrow E_1 \\ E &\rightarrow^k E_2' \rightarrow E_2 \end{aligned}$$

But the inductive hypothesis gives that $E_1' = E_2'$, and the determinism of the one-step relation, proved in the previous Lemma, gives the required $E_1 = E_2$. ■

This corollary leads directly to the determinacy of the final result.

Lemma 7 If $E \rightarrow^* \mathtt{n}$ and $E \rightarrow^* \mathtt{n}'$ then $\mathtt{n} = \mathtt{n}'$.

Proof The statement $E \rightarrow^* \mathtt{n}$ means that E reduces to $\mathtt{n}$ in some finite number of steps. So there is some natural number k_1 such that $E \rightarrow^{k_1} \mathtt{n}_1$. Similarly we have some k_2 such that $E \rightarrow^{k_2} \mathtt{n}_2$. Now either $k_1 \leq k_2$ or $k_2 \leq k_1$. Let us assume the former; the proof in the latter case is completely symmetric. Then these derivations take the form

$$\begin{aligned} E &\rightarrow^{k_1} \mathtt{n}_1 \\ E &\rightarrow^{k_1} E' \rightarrow^{(k_2-k_1)} \mathtt{n}_2 \end{aligned}$$

for some intermediary expression E'.

But by the previous Corollary E' must be the same as $\mathtt{n}$. According to the rules in Slide 16 no reductions can be made from numerals. So the reduction $E' \rightarrow^{(k_2-k_1)} \mathtt{n}_2$ must be the trivial one $\mathtt{n}_1 \rightarrow^0 \mathtt{n}_2$. In other words $\mathtt{n}_1 = \mathtt{n}_2$. ■

We now know that every term reaches at most one final answer; of course for this simple language we can show that *normalisation* also holds, i.e. there is a final answer for every expression.

Lemma 8 For all E there is some n such that $E \rightarrow^* \mathtt{n}$.

Proof By induction (!!!) on the structure of E.

Base Case E is a numeral $\mathtt{n}$. Then $\mathtt{n} \rightarrow^* \mathtt{n}$ as required.

Inductive Step E is $(E_1 + E_2)$. By the inductive hypothesis, we have numbers n_1 and n_2 such that $E_1 \rightarrow^* \mathtt{n_1}$ and $E_2 \rightarrow^* \mathtt{n_2}$. For each step in the reduction

$$E_1 \rightarrow E_1' \rightarrow E_1'' \cdots \rightarrow \mathtt{n_1}$$

applying the rule for reducing the left argument of an addition gives

$$(E_1 + E_2) \rightarrow (E_1' + E_2) \rightarrow (E_1'' + E_2) \cdots \rightarrow (\mathtt{n_1} + E_2).$$

Applying the other rule to the sequence for $E_2 \rightarrow^* \mathtt{n_2}$ allows us to deduce that

$$(\mathtt{n_1} + E_2) \rightarrow^* (\mathtt{n_1} + \mathtt{n_2}) \rightarrow \mathtt{n_3}$$

where $n_3 = n_1 + n_2$. Hence $(E_1 + E_2) \rightarrow^* \mathtt{n_3}$. ■

Corollary 9 For every expression E there is exactly one n such that $E {\rightarrow^*} \mathtt{n}$.

We now know that our small-step semantics computes exactly one final answer for any given expression. We expect that the final answers given by the small-step and big-step semantics should agree, and indeed they do.

Theorem 10 For any expression E,

$$E \Downarrow \mathtt{n} \text{ if and only if } E \rightarrow^* \mathtt{n}$$

Exercise Prove this theorem by induction on the structure of expressions.

3.7 The correspondence theorem

We previously stated the following theorem which relates the denotational semantics of *Exp* to the big-step semantics. Let us now prove this result.

Theorem 11 For all expressions E, $[\![E]\!] = n$ if and only if $E \Downarrow \mathtt{n}$.

Proof By induction on the structure of E. If E is a numeral $\mathtt{n}$, clearly $[\![E]\!] = n$ and $E \Downarrow \mathtt{n}$, so the base case is trivial.

So suppose E is of the form $(E_1 + E_2)$. We have to prove $(E_1 + E_2) \Downarrow \mathtt{n}$ if and only if $[\![E]\!] = n$.

First suppose $(E_1 + E_2) \Downarrow \mathtt{n}$. This can only be proved using the rule (B-ADD) from Slide 10. In other words there is an application of the rule of the form

$$\frac{E_1 \Downarrow \mathtt{n_1} \quad E_2 \Downarrow \mathtt{n_2}}{(E_1 + E_2) \Downarrow \mathtt{n_3}}$$

for some numbers n_1, n_2 such that $n_3 = n_1 + n_2$. By the inductive hypothesis, $[\![E_i]\!] = n_i$ for $i = 1, 2$, and therefore

$$[\![E_1]\!] + [\![E_2]\!] = n_1 + n_2 = n_3.$$

Conversely suppose $[\![E_1]\!] + [\![E_2]\!] = n$ for some number n. Looking up the definition of the denotational semantics of *Exp* in Slide 19 we see that there must be two numbers n_1 and n_2, namely $[\![E_1]\!]$ and $[\![E_2]\!]$ respectively, such that $n = n_1 + n_2$. Since $[\![E_i]\!] = n_i$ we can apply structural induction to obtain $E_i \Downarrow \mathtt{n_i}$ and an application of the rule (B-ADD) gives the required $(E_1 + E_2) \Downarrow \mathtt{n}$.

■

To sum up the results of this chapter, we have seen three different semantics for the language *Exp*, a denotational one, given in Slide 19, and two operational ones, in Slide 10 and Slide 16. We now know that these three different views actually coincide. For every expression E

$$[\![E]\!] = n \text{ if and only if } E \Downarrow \mathtt{n} \text{ if and only if } E \rightarrow^* \mathtt{n}$$

4 A Simple Programming Language

Let us now consider a "proper" programming language. The language *While* of *while-programs* has a grammar consisting of three syntax-categories: *expressions* as before, which we now denote with N to indicate that they are *numeric* expressions; *booleans*, which are very similar to expressions but represent truth-values rather than numbers; and *commands*, which are imperative statements which affect the store of the computer. The grammar is given on Slide 43.

The collection of expressions now includes a class x of mutable variables; here x ranges over some fixed, infinite set of identifiers. The expression x is intended to mean "the value currently stored in x". The intended meaning of the commands should be obvious to anyone familiar with imperative programming; we use the syntax $x := E$ for the assignment statement, which evaluates E and stores the result in x.

Abstract Syntax A reminder: we always deal with *abstract syntax*, even though the grammar above looks a bit like the kind of concrete syntax you might type in to a computer. So, we're really dealing with *trees* built up out of the term-forming operators given above. The operators we have for commands are:

- assignment, which takes a variable and an expression and gives a command, written $x := E$

Syntax of ***While***

$$
\begin{array}{rcl}
B \in \mathit{Bool} & ::= & \mathtt{true} \mid \mathtt{false} \mid E = E \mid E < E \mid \cdots \\
 & \mid & B \& B \mid \neg\, B \mid \cdots \\
E \in \mathit{Exp} & ::= & x \mid \mathtt{n} \mid (E + E) \mid \cdots \\
C \in \mathit{Com} & ::= & x := E \mid \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C \\
 & \mid & C\,;C \mid \mathtt{skip} \mid \mathtt{while}\ B\ \mathtt{do}\ C
\end{array}
$$

Slide 43

Example program

```
$x := z + y$
while $y > 0$ do
   $x := x + z;$
   $y := y - 1;$
   $z := z - 1;$
```

Slide 44

- the conditional, taking a boolean and two commands and yielding a command, written `if` B `then` C_1 `else` C_2

- sequential composition, which takes two commands and yields a command, written C_1 ; C_2 (note that the semicolon here is an operator joining two commands into one, and not just a piece of punctuation at the end of a command)

- the "do nothing" constant `skip`

- the loop constructor, which takes a boolean and a command and yields a command, written `while` B `do` C.

4.1 Small-step Semantics

How should we give a small-step semantics to *While*? In particular, how do we evaluate a variable:

$$x \rightarrow ?$$

or an assignment

$$x := \mathtt{n} \rightarrow ?$$

Obviously, we need some more information, about the state of the machine's *memory*. Slide 45 gives the definition of a *state* suitable for modelling *While*. Intuitively, a state s tells us what, if anything, is stored in the memory location corresponding to each identifier of the language, and $s[x \mapsto n]$ is the state s updated so that the location corresponding to x contains `n`.

Our small-step semantics will therefore be concerned with programs together with their store, so we define a relation of the form

$$\langle P, s\rangle \rightarrow \langle P', s'\rangle.$$

Expressions and Booleans Expressions and booleans do not present any difficulty. The only really new kind of expression is the variable. Its semantics involves fetching the appropriate value from the store. The corresponding rule, and some of the old rules updated for the new language, are shown in Slide 46.

Exercise Write down the other rules for expressions, and all the rules for booleans.

States

- A *state* is a partial function from identifiers to numerals such that $s(x)$ is defined only for finitely many x.
- The state $s[x \mapsto n]$ is defined by

$$s[x \mapsto n](y) = \begin{cases} \mathtt{n} & \text{if } y = x \\ s(y) & \text{otherwise} \end{cases}$$

Slide 45

Expressions

$$\text{(W-EXP.NUM)} \quad \frac{}{\langle x, s\rangle \to \langle \mathtt{n}, s\rangle}\; s(x) = \mathtt{n}$$

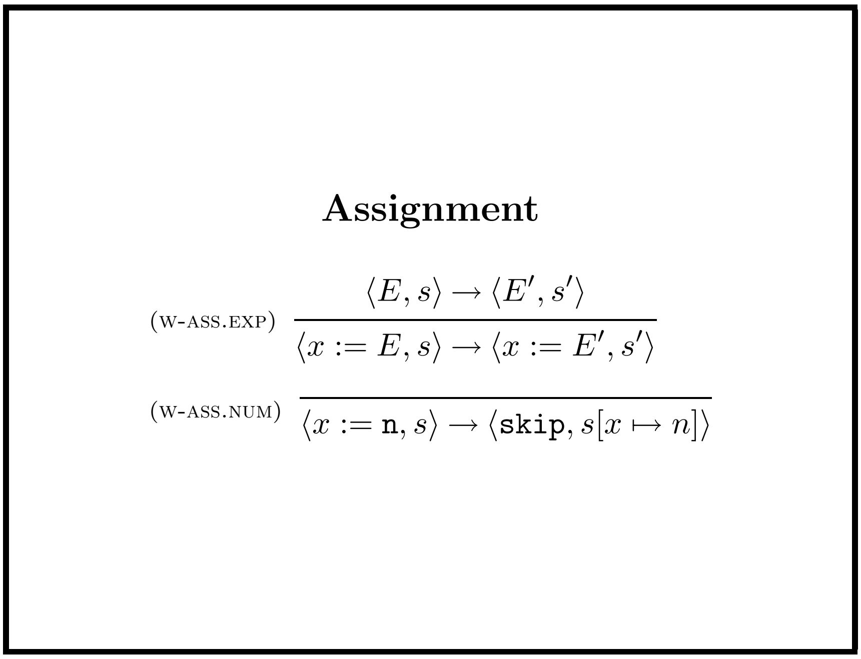
$$\text{(W-EXP.ADD)} \quad \frac{}{\langle (\mathtt{n}_1 + \mathtt{n}_2), s\rangle \to \langle \mathtt{n}_3, s\rangle}\; n_3 = n_1 + n_2$$

$$\text{(W-EXP.LEFT)} \quad \frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle (E_1 + E_2), s\rangle \to \langle (E_1' + E_2), s'\rangle}$$

Slide 46

Commands The rules defining the semantics for commands are different: they will alter the store in interesting ways. Intuitively, we want our rules to show how commands update the store, and we will know that a command has finished its work when it reduces to `skip`. We shall now consider each kind of command in turn and write down the appropriate rules.

For assignment, $x := E$, we first want to evaluate E to some numeral $\mathtt{n}$, and then update the store so that x contains $\mathtt{n}$. See Slide 47.



Assignment

$$\textsc{(w-ass.exp)}\quad \frac{\langle E, s\rangle \rightarrow \langle E', s'\rangle}{\langle x := E, s\rangle \rightarrow \langle x := E', s'\rangle}$$

$$\textsc{(w-ass.num)}\quad \frac{}{\langle x := \mathtt{n}, s\rangle \rightarrow \langle \mathtt{skip}, s[x \mapsto n]\rangle}$$

Slide 47

For sequential composition, $C_1\,; C_2$, we first allow C_1 to run to completion, changing the store as it does so, and then compute C_2. See Slide 48.

For conditionals, we first evaluate the boolean guard; if this returns `true` we take the first branch; if it returns `false` we take the second branch. One rule for this is given on Slide 49.

Exercise Write down the other rules for the conditional.

What about `while`? Obviously, we want to evaluate the boolean guard, and, if true, run the command and then go back to the beginning and start again. Perhaps something like Slide 50 would do the trick?

What's the problem here? The problem is that the only rule we've got which is capable of entering the loop body is the one for `while true do` C, which ought to be an infinite loop. By evaluating the boolean guard "in place" with the rule

$$\frac{\langle B, s\rangle \rightarrow \langle B', s\rangle}{\langle \mathtt{while}\ B\ \mathtt{do}\ C, s\rangle \rightarrow \langle \mathtt{while}\ B'\ \mathtt{do}\ C, s'\rangle}$$

Sequential Composition

$$\text{(W-SEQ.LEFT)}\quad \frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1 \,;\, C_2, s\rangle \to \langle C_1' \,;\, C_2, s'\rangle}$$

$$\text{(W-SEQ.SKIP)}\quad \frac{}{\langle \texttt{skip} \,;\, C_2, s\rangle \to \langle C_2, s\rangle}$$

Slide 48

Conditional

$$\text{(W-COND.TRUE)}\quad \frac{}{\langle \texttt{if true then } C_1 \texttt{ else } C_2, s\rangle \to \langle C_1, s\rangle}$$

Slide 49

Incorrect Semantics for `while`

$$\frac{\langle B, s\rangle \rightarrow \langle B', s\rangle}{\langle \texttt{while } B \texttt{ do } C, s\rangle \rightarrow \langle \texttt{while } B' \texttt{ do } C, s'\rangle}$$

$$\frac{}{\langle \texttt{while false do } C, s\rangle \rightarrow \langle \texttt{skip}, s\rangle}$$

$$\frac{}{\langle \texttt{while true do } C, s\rangle \rightarrow ?}$$

Slide 50

we have made a serious error; the point is that we do not want to evaluate this boolean once and use that value for ever more, but rather to evaluate the boolean every time we go through the loop. So, when we evaluate it the first time, it is vital that we don't throw away the "old" B, which this rule does.

The solution is to make a *copy* of B to evaluate each time, and luckily for us, our syntax allows us to express this in a rather sneaky way. The single rule that we need for `while` is given on Slide 51.

4.1.1 An example

Slide 52 shows a program for computing the factorial of x and storing the answer in variable a.

Let s be the state $(x \mapsto 3, y \mapsto 2, a \mapsto 9)$, using an obvious notation for states. It should be the case that

$$\langle P, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle$$

where $s'(a) = 6$. (Can you predict the final values of x and y?)

Let's check that. First, some abbreviations: we write P' for the sub-program

$$\begin{array}{l} \texttt{while } y > 0 \texttt{ do} \\ \quad a := a \times y; \\ \quad y := y - 1 \end{array}$$

Correct semantics for `while`

$$\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \rightarrow$$
$$\langle \texttt{if}\ B\ \texttt{then}\ (C\,;\texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}, s\rangle$$

All that this rule does is to "unfold" the while loop once. If we could write down the infinite unfolding, there would be no need for the `while` syntax.

Slide 51

A factorial program!

$$\begin{aligned} P = \quad & y := x\,; a := 1; \\ & \texttt{while}\ y > 0\ \texttt{do} \\ & \quad (a := a \times y; \\ & \quad\ \ y := y - 1) \end{aligned}$$

Slide 52

and $s_{i,j,k}$ for the state mapping x to i, y to j and a to k. Thus $s = s_{3,2,9}$. Okay, let's dive in to the evaluation. Each line should really be justified by reference to one of the rules of the operational semantics.

$$\begin{array}{ll}
& \langle y := x \,;\, a := \texttt{1} \,;\, P', s\rangle \\
\longrightarrow & \langle y := \texttt{3} \,;\, a := \texttt{1} \,;\, P', s\rangle \\
\longrightarrow & \langle \texttt{skip} \,;\, a := \texttt{1} \,;\, P', s_{3,3,9}\rangle \\
\longrightarrow & \langle a := \texttt{1} \,;\, P', s_{3,3,9}\rangle \\
\longrightarrow & \langle \texttt{skip} \,;\, P', s_{3,3,1}\rangle \\
\longrightarrow & \langle P', s_{3,3,1}\rangle \\
\longrightarrow & \langle \texttt{if } y > \texttt{0} \texttt{ then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,3,1}\rangle \\
\longrightarrow & \langle \texttt{if 3} > \texttt{0} \texttt{ then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,3,1}\rangle \\
\longrightarrow & \langle \texttt{if true then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,3,1}\rangle \\
\longrightarrow & \langle a := a \times y \,;\, y := y - \texttt{1} \,;\, P', s_{3,3,1}\rangle \\
\longrightarrow & \langle a := \texttt{1} \times y \,;\, y := y - \texttt{1} \,;\, P', s_{3,3,1}\rangle \\
\longrightarrow & \langle a := \texttt{1} \times \texttt{3} \,;\, y := y - \texttt{1} \,;\, P', s_{3,3,1}\rangle \\
\longrightarrow & \langle a := \texttt{3} \,;\, y := y - \texttt{1} \,;\, P', s_{3,3,1}\rangle \\
\longrightarrow & \langle \texttt{skip} \,;\, y := y - \texttt{1} \,;\, P', s_{3,3,3}\rangle \\
\longrightarrow & \langle y := y - \texttt{1} \,;\, P', s_{3,3,3}\rangle \\
\longrightarrow & \langle y := \texttt{3} - \texttt{1} \,;\, P', s_{3,3,3}\rangle \\
\longrightarrow & \langle y := \texttt{2} \,;\, P', s_{3,3,3}\rangle \\
\longrightarrow & \langle \texttt{skip} \,;\, P', s_{3,2,3}\rangle \\
\longrightarrow & \langle P', s_{3,2,3}\rangle \\
\longrightarrow & \langle \texttt{if } y > \texttt{0} \texttt{ then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,2,3}\rangle \\
\longrightarrow & \langle \texttt{if 2} > \texttt{0} \texttt{ then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,2,3}\rangle \\
\longrightarrow & \langle \texttt{if true then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,2,3}\rangle \\
\longrightarrow & \langle a := a \times y \,;\, y := y - \texttt{1} \,;\, P', s_{3,2,3}\rangle \\
\longrightarrow & \langle a := \texttt{3} \times y \,;\, y := y - \texttt{1} \,;\, P', s_{3,2,3}\rangle \\
\longrightarrow & \langle a := \texttt{3} \times \texttt{2} \,;\, y := y - \texttt{1} \,;\, P', s_{3,2,3}\rangle \\
\longrightarrow & \langle a := \texttt{6} \,;\, y := y - \texttt{1} \,;\, P', s_{3,2,3}\rangle \\
\longrightarrow & \langle \texttt{skip} \,;\, y := y - \texttt{1} \,;\, P', s_{3,2,6}\rangle
\end{array}$$

$$\begin{array}{ll}
\rightarrow & \langle y := y - \texttt{1} \,;\, P', s_{3,2,6}\rangle \\
\rightarrow & \langle y := \texttt{2} - \texttt{1} \,;\, P', s_{3,2,6}\rangle \\
\rightarrow & \langle y := \texttt{1} \,;\, P', s_{3,2,6}\rangle \\
\rightarrow & \langle \texttt{skip} \,;\, P', s_{3,1,6}\rangle \\
\rightarrow & \langle P', s_{3,1,6}\rangle \\
\rightarrow & \langle \texttt{if } y > \texttt{0} \texttt{ then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,1,6}\rangle \\
\rightarrow & \langle \texttt{if } \texttt{1} > \texttt{0} \texttt{ then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,1,6}\rangle \\
\rightarrow & \langle \texttt{if true then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,1,6}\rangle \\
\rightarrow & \langle a := a \times y \,;\, y := y - \texttt{1} \,;\, P', s_{3,1,6}\rangle \\
\rightarrow & \langle a := \texttt{6} \times y \,;\, y := y - \texttt{1} \,;\, P', s_{3,1,6}\rangle \\
\rightarrow & \langle a := \texttt{6} \times \texttt{1} \,;\, y := y - \texttt{1} \,;\, P', s_{3,1,6}\rangle \\
\rightarrow & \langle a := \texttt{6} \,;\, y := y - \texttt{1} \,;\, P', s_{3,1,6}\rangle \\
\rightarrow & \langle \texttt{skip} \,;\, y := y - \texttt{1} \,;\, P', s_{3,1,6}\rangle \\
\rightarrow & \langle y := y - \texttt{1} \,;\, P', s_{3,1,6}\rangle \\
\rightarrow & \langle y := \texttt{1} - \texttt{1} \,;\, P', s_{3,1,6}\rangle \\
\rightarrow & \langle y := \texttt{0} \,;\, P', s_{3,1,6}\rangle \\
\rightarrow & \langle \texttt{skip} \,;\, P', s_{3,0,6}\rangle \\
\rightarrow & \langle P', s_{3,0,6}\rangle \\
\rightarrow & \langle \texttt{if } y > \texttt{0} \texttt{ then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,0,6}\rangle \\
\rightarrow & \langle \texttt{if } \texttt{0} > \texttt{0} \texttt{ then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,0,6}\rangle \\
\rightarrow & \langle \texttt{if false then } (a := a \times y \,;\, y := y - \texttt{1} \,;\, P') \texttt{ else skip}, s_{3,0,6}\rangle \\
\rightarrow & \langle \texttt{skip}, s_{3,0,6}\rangle.
\end{array}$$

As you can see, this kind of calculation is agonising. But,

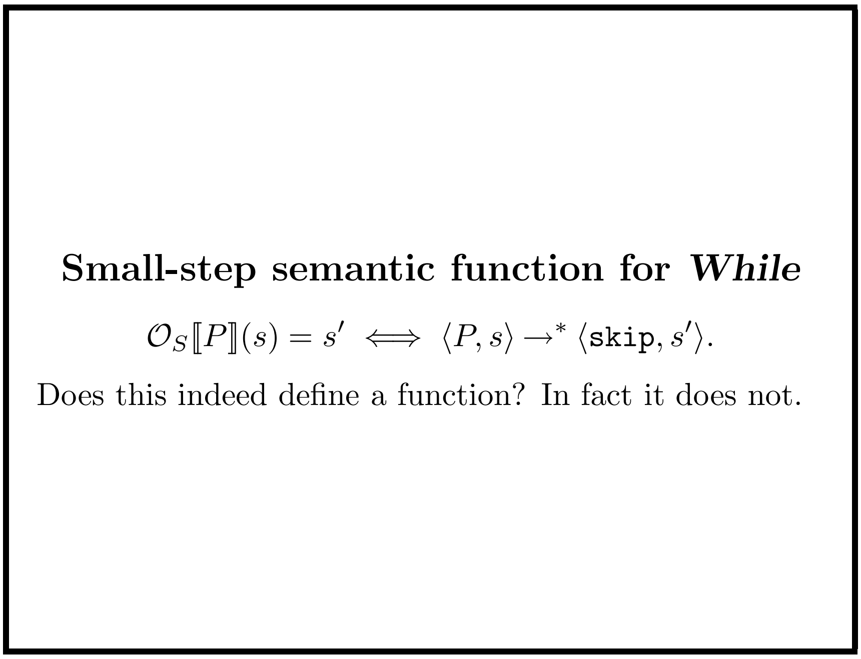
- it can be automated to give a simple *interpreter* for the language, based directly on the semantics.
- it is formal and precise, and there can be no argument about what should happen at a given time.
- it did compute the right answer, thank goodness!

4.1.2 The operational semantic function

To compute a *final answer* in this language, we're really interested in the *state* that is left when a program is evaluated to completion. That is to say, we want to know about s', where

$$\langle P, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle.$$

But what should the initial state s be? We could fix a particular s which we think of as the state in which all programs start. For example, we could decide that the starting state leaves every identifier undefined. This would be a perfectly reasonable approach, but it is probably more useful to define an operational semantic *function*: for each program, this function will take an initial state and return the final state left after running the program. The definition is given on Slide 53.



Small-step semantic function for *While*

$$\mathcal{O}_S[\![P]\!](s) = s' \iff \langle P, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle.$$

Does this indeed define a function? In fact it does not.

Slide 53

One problem with this definition is that in certain states, some programs become “stuck”, which is to say that they are not fully evaluated but have nowhere to go. An example is shown on Slide 54.

This problem is not particularly severe. It seems likely that a type system could be imposed, for example, to prevent programs from attempting to read from variables that have not been initialised. A much more serious obstruction to the definition of $\mathcal{O}_S[\![-]\!]$ is the possibility of a nonterminating computation, that is, an infinite loop; see Slide 55.

Let us prove the claim we made, that `while true do skip` never reaches a result. Before we do so, let us record a familiar fact about this semantics: determinacy.

Lemma 12 The small-step semantics is deterministic, that is to say, for any configuration $\langle P, s\rangle$, there is at most one $\langle P', s'\rangle$ such that $\langle P, s\rangle \rightarrow \langle P', s'\rangle$.

Exercise Prove this claim. You should concentrate on the cases of commands, and just give an indication of any interesting nuances for the cases

A stuck configuration

Let s be the state mapping x to 3 and undefined on every other identifier. Then what is $\mathcal{O}_S[\![y := y + 1]\!](s)$?

$$\langle s, y := y + 1\rangle \to ?$$

We need to evaluate the expression $y + 1$, but there is no rule which can be applied.

Slide 54

Infinite loops

The program `while true do skip` loops forever. Therefore, we expect that there are no states s and s' such that

$$\langle \texttt{while true do skip}, s\rangle \to^* \langle \texttt{skip}, s'\rangle$$

so $\mathcal{O}_S[\![-]\!]$ as we have defined it does not give a function in this case.

Slide 55

of expressions and booleans.

Theorem 13 For any state s, there is no s' such that

$$\langle \texttt{while true do skip}, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle.$$

Proof Let us first calculate a few steps of the evaluation of this program.

$$\begin{array}{ll}
& \langle \texttt{while true do skip}, s\rangle \\
\rightarrow & \langle \texttt{if true then (skip ; while true do skip) else skip}, s\rangle \\
\rightarrow & \langle \texttt{skip ; while true do skip}, s\rangle \\
\rightarrow & \langle \texttt{while true do skip}, s\rangle
\end{array}$$

As you can see, it seems unlikely that this will ever get anywhere! But we need to prove this rigorously.

Suppose on the contrary that it is possible for

$$\langle \texttt{while true do skip}, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle,$$

and let n be the number of steps taken for this evaluation. Note that since the semantics is deterministic, this number n is well-defined.

Again, determinacy tells us that the first three steps of the evaluation must be the steps we calculated above, and then the remaining $n-3$ steps of the evaluation show that

$$\langle \texttt{while true do skip}, s\rangle \rightarrow^* \langle \texttt{skip}, s'\rangle,$$

which is not possible, since this takes n steps! This is a contradiction, so we deduce that no such evaluation can exist. ∎

Therefore, inherent in our language is the fact that some computations do not yield final answers. However, the definition we have given for our semantic function is reasonable; so we just accept it and live with the fact that it is partial.

4.1.3 Discussion: Side-effects and Evaluation Order

Something worth noticing about our language is that the only phrases which affect the store directly are the assignment statements, and these statements are always contained inside commands. Furthermore, commands are strictly sequenced by the ; operator. This means that there is never any confusion about what should be in the store at a given time, and none of the decisions we have made about order of evaluation affect the overall semantic function.

Small-step semantic function for ***While***

For each program P, we define a *partial function* $\mathcal{O}_S[\![P]\!]$ from states to states as follows.

$$\mathcal{O}_S[\![P]\!](s) = s' \iff \langle P, s\rangle \to^* \langle \texttt{skip}, s'\rangle.$$

Slide 56

In more sophisticated languages, this happy situation can be compromised. For example, commands can often creep into the language of expressions via constructs like Java's `return`. For example, the code

$$x := x + 1\,;\texttt{return}(x)$$

which we think of as an expression, because it returns a numerical result, has a side-effect on the store.

We can now write a composite expression like

$$(x := x + 1\,;\texttt{return}(x)) + (x := x \times 2\,;\texttt{return}(x)).$$

It is now *vital* that we pay close attention to the semantics of addition. Does + evaluate its argument left-to-right or right-to-left?

Exercise Write down the sets of rules corresponding to each evaluation strategy for +, and evaluate the above in the state ($x \mapsto 0$) under each set of rules.

Strictness In the case of addition, the only reasonable choices for evaluation are left-to-right and right-to-left, although other choices do exist, such as evaluating both arguments twice! In any case, it is clear that both arguments must be evaluated at least once before the result of the addition can be calculated. For some kinds of expression, this is not the case.

Side-effecting expressions

If we allow expressions like

$$x := x + 1\,;\mathtt{return}(x)$$

then

$$(x := x + 1\,;\mathtt{return}(x)) + (x := x \times 2\,;\mathtt{return}(x))$$

depends on evaluation order.

Slide 57

For example, the logical "and" operator, written & in our syntax, when applied to `false` and any other boolean, must return `false`. It is therefore possible to write a semantics for & as on Slide 58.

The programmer needs to know what the semantics is! We no longer have the equivalence

$$(B_1 \& B_2) \cong (B_2 \& B_1)$$

since for example, in any state

$$\mathtt{false}\&(\mathtt{while\ true\ do\ skip}\,;\mathtt{return}(\mathtt{true})) \rightarrow \mathtt{false}$$

while

$$(\mathtt{while\ true\ do\ skip}\,;\mathtt{return}(\mathtt{true}))\&\mathtt{false}$$

gets into an infinite loop.

Procedure and method calls Though we will not consider languages with procedures or method calls formally on this course, we can informally apply these ideas to obtain a deeper understanding of such languages. The issues of strictness and evaluation order crop up again in this setting.

For example, in a method like

```
void aMethod(int x) {
  return;
}
```

Short-circuit semantics of &

$$\frac{B_1 \to B_1'}{(B_1 \& B_2) \to (B_1' \& B_2)}$$

$$\frac{}{(\texttt{false} \& B_2) \to \texttt{false}} \qquad \frac{}{(\texttt{true} \& B_2) \to B_2}$$

Slide 58

Strictness

An operation is called *strict* in one of its arguments if it always needs to evaluate that argument.

Addition is strict in both arguments, or *bi-strict*.

The semantics of & given above makes & a *left-strict* operator. It is *non-strict* in its right argument.

Slide 59

the argument `x` is never used. So in a call such as

```
aMethod(y := y + 1; return y)
```

do we need to evaluate the argument? Clearly, the outcome of a program containing a call like this will depend on the semantic decision we make here. In a method call such as

```
anotherMethod(exp1, exp2)
```

in what order should we evaluate the arguments, if they are used? If an argument is used twice in the body of the method, should it be evaluated twice? There are plenty of different semantic decisions to be made here. Some popular choices are:

- evaluate all arguments, left to right, and use the *result* of this evaluation each time the argument is used; this is called *call-by-value*, and is roughly what Java and ML do.
- replace each use of an argument in the method body by the text of the actual parameter the programmer has supplied, so that each time the argument is called, it is re-evaluated. This is *call-by-name*, and is what Algol 60 did (does?).
- evaluate an argument only when it is actually used in the body of the method, but then remember the result so that if the argument is used again, it is not re-evaluated. This is *call-by-need*, and is what a large number of functional languages like Haskell do.

Exercise Write the code of a method `myMethod(exp1, exp2)` and a particular call of this method, such that the three evaluation strategies above all give different results.

The purpose of this discussion is to alert you to the fact that there may be several reasonable but nonetheless crucially different choices to be made about the semantics of various language constructs. One role of a formal study of semantics is to help discover where such choices and ambiguities lie, and to resolve them in a fixed, clear and well-documented way.

4.2 Denotational Semantics

We shall now try to give a denotational semantics to the simple language of `while`-programs. The first step is to choose our *semantic domains*.

4.2.1 Semantic Domain for Commands

Let us focus first of all on commands, which are the biggest difference between *While* and *Exp*. The small-step operational semantics gave us a partial function of the form

$$\mathcal{O}_S[\![C]\!](s) = s'$$

that is, given a command, we have a partial function $\mathcal{O}_S[\![C]\!]$ from states to states. This suggests that we might use the set of partial functions from states to states, or something like it, as our semantic domain for commands.

Let Σ be the set of all states. Define $\Sigma_\perp$ to be the set $\Sigma \cup \{\perp\}$, that is, the set Σ together with an extra element $\perp$, called *undefined* or *bottom*, which represents a stuck computation or an infinite loop. Then the set of *state transformers* is defined on Slide 60.

State Transformers

The set of state transformers is defined to be

$$ST = [\Sigma \to \Sigma_\perp]$$

that is, the set of (total) functions which take a starting state and return either $\perp$, to indicate that the computation got stuck or looped forever, or a final state.

Slide 60

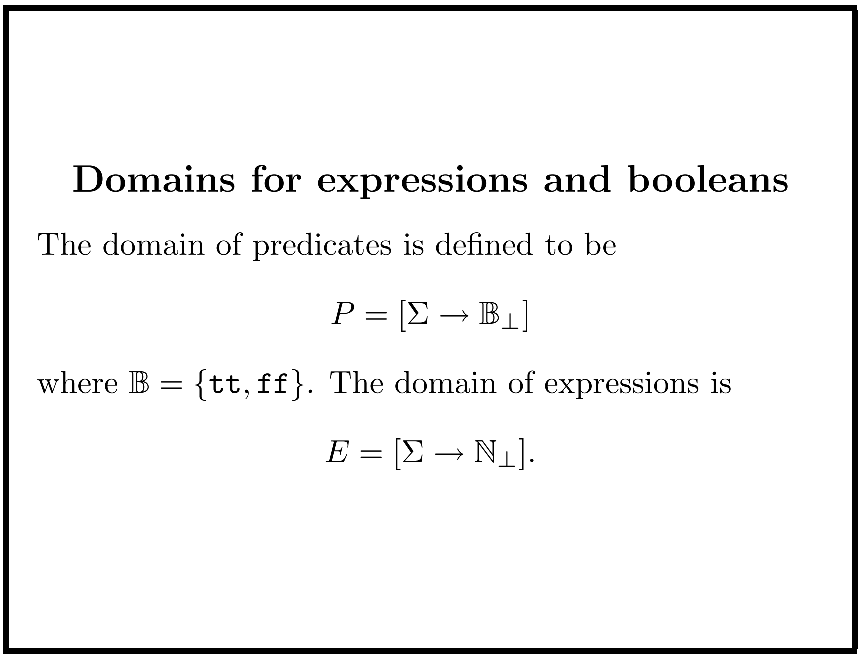
The set ST will be our semantic domain for commands.

A note on notation The metavariable s, and variants of it like s' and so on, will be used to range over *proper states*, not including $\perp$. So, if we say that $f(s) = s'$, it is implicit that $s' \neq \perp$.

4.2.2 Other Semantic Domains

For booleans and expressions, note that our language allows a boolean or an expression to *depend upon* the store, but *not* to change it. Also, though expressions and booleans cannot get into infinite loops, they may become stuck, so we have to account for this in our choice of semantic domain.

We model booleans with the domain of *predicates* P and expressions with the domain E given on Slide 61.



Domains for expressions and booleans

The domain of predicates is defined to be

$$P = [\Sigma \to \mathbb{B}_\perp]$$

where $\mathbb{B} = \{\texttt{tt}, \texttt{ff}\}$. The domain of expressions is

$$E = [\Sigma \to \mathbb{N}_\perp].$$

Slide 61

Remark As before, fixing the semantic domains tells us something about the language. For example, using ST for the commands acknowledges the possibility of non-termination, but makes clear that a command will yield at most one final state in any given starting state. Similarly, our choice of domain for the booleans automatically eliminates any possibility of side-effects being caused by booleans.

4.2.3 Semantic Functions

We shall now attempt to define three semantic functions, one for each category in the grammar of *While*. See Slide 62.

Semantics of Expressions We shall give the semantics only for the new kind of expression, namely the variable. The others will be left as an exercise. The semantics of variable lookup is given on Slide 63.

Semantic functions for *While*

$$\begin{array}{rrcl} \mathcal{C}\llbracket - \rrbracket : & \mathit{Com} & \rightarrow & \mathit{ST} \\ \mathcal{E}\llbracket - \rrbracket : & \mathit{Exp} & \rightarrow & E \\ \mathcal{B}\llbracket - \rrbracket : & \mathit{Bool} & \rightarrow & P \end{array}$$

Slide 62

Variable Lookup

The semantics of a variable is simple: the store is examined to see if there is a value in the variable. If so, the value is returned, and if not, the expression is stuck so we return $\perp$.

$$\mathcal{E}\llbracket x \rrbracket(s) = \begin{cases} s(x) & \text{if } s(x) \text{ defined} \\ \perp & \text{otherwise.} \end{cases}$$

Slide 63

Semantics of Commands To define the semantics of a command, we ask ourselves how the command transforms the state, and attempt to write down a function which captures our intuition. The next few slides give the definitions for assignment, the skip command, and sequential composition.

Assignment

This is easy: an assignment $x := E$ transforms store s by updating x to contain the value of E. We have to take account of the possibility of stuck expressions, however.

$$\mathcal{C}[\![x := E]\!](s) = \begin{cases} s[x \mapsto \mathcal{E}[\![E]\!](s)] & \text{if } \mathcal{E}[\![E]\!](s) \neq \bot \\ \bot & \text{otherwise.} \end{cases}$$

Note that in this definition, E is evaluated in store s.

Slide 64

Exercise Give definitions of the semantics of all the remaining constructs apart from `while`.

Example We have now given enough definitions to allow us to calculate the meanings of a few commands. Consider for example

$$C = x := \mathtt{0}\,; x := x + \mathtt{1}.$$

We shall show that $\mathcal{C}[\![C]\!](s) = s[x \mapsto 1]$.

$$\begin{aligned} \mathcal{C}[\![C]\!](s) &= (\mathcal{C}[\![x := \mathtt{0}]\!]\,; \mathcal{C}[\![x := x + \mathtt{1}]\!])(s) \\ &= \mathcal{C}[\![x := x + \mathtt{1}]\!](\mathcal{C}[\![x := \mathtt{0}]\!](s)) \\ &= \mathcal{C}[\![x := x + \mathtt{1}]\!](s[x \mapsto 0]) \\ &= s[x \mapsto \mathcal{E}[\![x + \mathtt{1}]\!](s[x \mapsto 0])]. \end{aligned}$$

Clearly,

$$\mathcal{E}[\![x + \mathtt{1}]\!](s[x \mapsto 0]) = 1$$

so

$$\mathcal{C}[\![C]\!](s) = s[x \mapsto 1]$$

as we claimed.

`skip`

The easiest of all: `skip` leaves the store alone.

$$\mathcal{C}[\![\texttt{skip}]\!](s) = s.$$

Slide 65

Sequential Composition

How does $C_1\,;C_2$ transform a store? Intuitively, first C_1 transforms the original state s to some s', then C_2 starts running in state s', leaving some s'', which is the outcome of the whole command.

If C_1 gets stuck or into an infinite loop, so does the whole command; similarly for C_2.

We shall define a state transformer for $C_1\,;C_2$ to reflect this intuition.

Slide 66

Semantics of Sequential Composition

The state transformer $\mathcal{C}[\![C_1 \,;C_2]\!]$ is defined by

$$\mathcal{C}[\![C_1 \,; C_2]\!](s) = \begin{cases} \bot & \text{if } \mathcal{C}[\![C_1]\!](s) = \bot \\ \mathcal{C}[\![C_2]\!](\mathcal{C}[\![C_1]\!](s)) & \text{otherwise.} \end{cases}$$

Notice that the second line is "well-typed", because if $\mathcal{C}[\![C_1]\!](s) \neq \bot$ then $\mathcal{C}[\![C_1]\!](s) \in \Sigma$, so we can indeed apply $\mathcal{C}[\![C_2]\!]$ to it.

Slide 67

Conditional

A command `if` B `then` C_1 `else` C_2 transforms a state s as follows:

- work out if B is true or false in state s
- if true, transform the state s by running C_1
- if false, transform the state s by running C_2.

Slide 68

Conditional, continued

We therefore define $\mathcal{C}[\![\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2]\!](s)$ to be

$$\begin{array}{cl} \mathcal{C}[\![C_1]\!](s) & \text{if } \mathcal{B}[\![B]\!](s) = \texttt{tt} \\ \mathcal{C}[\![C_2]\!](s) & \text{if } \mathcal{B}[\![B]\!](s) = \texttt{ff} \\ \bot & \text{otherwise} \end{array}$$

Slide 69

4.2.4 Compositionality

As for the language of expressions, the denotational semantics is *compositional*, so the meaning of a program is built up out of the meanings of its subprograms. This means that each of the term forming operations in the language *While* has a denotational meaning.

For example, the term forming operation ; which takes two commands and gives back their sequential composition, has as its meaning the function seq, defined on Slide 70.

It is now reasonable to say that $[\![;]\!] = \mathsf{seq}$.

This is really no more than rewriting the original definitions, but it makes the point that denotational semantics gives meaning to term-forming operations, not just individual programs.

4.2.5 Semantics of `while`

Okay, what about `while`? How can we write down a "looping" state transformer?

Recall the trick that we used to give a small-step semantics to `while`:

$$\langle \texttt{while } B \texttt{ do } C, s\rangle \rightarrow \langle \texttt{if } B \texttt{ then } (C \,;\, \texttt{while } B \texttt{ do } C) \texttt{ else skip}, s\rangle.$$

Semantics of the Sequential Composition Operator

We define the function

$$\mathsf{seq} : ST \times ST \rightarrow ST$$

by

$$\mathsf{seq}(f, g)(s) = \begin{cases} \perp & \text{if } f(s) = \perp \\ g(f(s)) & \text{otherwise.} \end{cases}$$

Slide 70

Semantics of the Conditional Operator

We define the function

$$\mathsf{cond} : P \times ST \times ST \rightarrow ST$$

by

$$\mathsf{cond}(p, f, g)(s) = \begin{cases} f(s) & \text{if } p(s) = \mathtt{tt} \\ g(s) & \text{if } p(s) = \mathtt{ff} \\ \perp & \text{otherwise} \end{cases}$$

Slide 71

This says that the way $\texttt{while}\ B\ \texttt{do}\ C$ transforms the state is the same as the transformation given by

$$\texttt{if}\ B\ \texttt{then}\ (C\,;\texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}.$$

In denotational terms, this statement looks like the equation on Slide 72.

An equation for `while`

$$\begin{aligned} & \mathcal{C}[\![\texttt{while}\ B\ \texttt{do}\ C]\!] \\ = \ & \mathcal{C}[\![\texttt{if}\ B\ \texttt{then}\ (C\,;\texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}]\!]. \end{aligned}$$

Slide 72

But can we use this equation as a definition?

We are trying to define the semantics of program phrases by induction on their structure. That means, as usual, that when we define the semantics of a compound phrase, we may assume that the semantics of each of its sub-phrases have already been defined. Bearing this in mind, it is clear that each of the definitions we have given so far is well-defined, i.e. the formula on the right-hand side denotes an element of the semantic domain.

The equation above is different. It contains, on the right, a reference to

$$\mathcal{C}[\![\texttt{while}\ B\ \texttt{do}\ C]\!]$$

which we have *not* yet defined: it is not a sub-phrase of itself! So we have a circular definition, or to put it another way, we don't have a definition at all.

The approach we've used before guided us to good definitions of semantic functions, so let's try to answer the following question: how does $\texttt{while}\ B\ \texttt{do}\ C$ transform a state s?

We need a fixed point

What we need to do is to find some $f \in ST$ such that

$$f = \mathsf{cond}(\mathcal{B}\llbracket B \rrbracket, (\mathcal{C}\llbracket C \rrbracket \,; f), \mathsf{id})$$

where id, the *identity function* is the semantics of `skip` we have already defined. We can then use f as the semantics of `while` B `do` C.

Slide 73

A helper function

To put this another way, define a function $F : ST \to ST$ by

$$F(f) = \mathsf{cond}(\mathcal{B}\llbracket B \rrbracket, (\mathcal{C}\llbracket C \rrbracket \,; f), \mathsf{id}).$$

Slide 74

A first approximation of `while`

If B is false in state s, it does nothing, that is, it returns the state s. In this particular case, the transformation is the same as that given by

$$\texttt{if } B \texttt{ then } \text{anything } \texttt{else skip}.$$

Slide 75

A sneaky step

Since the “anything” above could be anything (!!!), let us replace it with the phrase (C ; anything). That gives us

$$\texttt{if } B \texttt{ then } (C \,;\, \text{anything}) \texttt{ else skip}.$$

The semantics of this is now $F(\text{anything})$.

Slide 76

A second approximant

If B is true in state s but becomes false after running the loop body C once, then the loop transforms the state in the same way as

$$\begin{aligned}\texttt{if}\ B\quad &\texttt{then}\ C\,;(\texttt{if}\ B\ \texttt{then}\ C\,;\text{anything}\ \texttt{else skip})\\ &\texttt{else skip}\end{aligned}$$

The semantics of this is $F(F(\text{anything}))$.

Slide 77

- If B is false in state s, it does nothing, that is, it returns the state s. In this particular case, the transformation is the same as that given by

  $$\texttt{if}\ B\ \texttt{then}\ \text{anything}\ \texttt{else skip}.$$

  Since the "anything" above could be anything (!!!), let us replace it with the phrase $(C\,;\text{anything})$. That gives us

  $$\texttt{if}\ B\ \texttt{then}\ (C\,;\text{anything})\ \texttt{else skip}.$$

  We have done this for a sneaky reason: the semantics of the phrase above is now the same as $F(f)$, where F is the function we defined earlier and f is the semantics of "anything".

  This phrase acts the same way as $\texttt{while}\ B\ \texttt{do}\ C$ on those states which do not require entering the loop body at all. Of course, if the loop body is entered, it is very different.

  The key point is that this is a phrase for which we already have a semantics, and *it gives us the right answer some of the time.* We shall now improve on this by finding a phrase which gives us the right answer more often.

- If B is true in state s, $\texttt{while}\ B\ \texttt{do}\ C$ runs the command C, transforming the state to s'; if B is now false, that is the end of the computation. In this case, therefore, the transformation is the same as that given by

  $$\texttt{if}\ B\ \texttt{then}\ C\,;(\texttt{if}\ B\ \texttt{then}\ \text{anything}\ \texttt{else skip})\ \texttt{else skip}.$$

Again replacing anything with (C ; anything) gives

$$\texttt{if } B \texttt{ then } C\,;(\texttt{if } B \texttt{ then } (C\,; \text{anything}) \texttt{ else skip}) \texttt{ else skip}.$$

This semantics of this is $F(F(f))$, and gives the right state transformation in the case that the loop body is entered no times, or one time; but if the loop body needs to be entered more than once, it might not be correct. Still, we're getting closer.... Our new phrase works for all the states the previous one worked for, and some more.

- The phrase corresponding to $F(F(F(f)))$ gives the correct transformation for states which require going round the loop no times, once or twice. If we write $F^n(f)$ for the same phrase with n uses of F, we get the right state transformer for those states which require going round the loop $n-1$ or fewer times.

A sequence of approximants

Given a starting state s:

- if, starting in state s, the loop body would be executed less than n times, then for any state transformer f, $F^n(f)(s)$ gives the same final state that the loop would give;
- if more than n executions of the loop body would be required, $F^n(f)(s)$ may give the wrong answer.

Note that as n gets bigger, $F^n(f)$ is right on more and more starting states.

Slide 78

These ideas are enough to let us define a state transformer which gives the fixed point we require: see Slides 79 and 80.

Rather than going into the proof that the f we've just defined (on Slide 80) is really a state transformer and is really a fixed point of F, let's try to do the same tricks using the syntax of *While*.

Recall that `diverge` is a program which immediately goes into an infinite loop. Let's add `diverge` as a primitive to our language, just for now, and define

$$\mathcal{C}[\![\texttt{diverge}]\!](s) = \bot$$

Better Approximants

In the above, set f to be the state transformer which gives $\perp$ for any starting state s. Write this state transformer as $\perp$ too! Then

- if, starting in state s, the loop body would be executed less than n times, then $F^n(\perp)(s)$ gives the same final state that the loop would give;
- if more than n executions of the loop body would be required, $F^n(f)(\perp)$ gives $\perp$, which may be incorrect.

Note that if $F^n(\perp)(s) \neq \perp$, we know it must be the right answer.

Slide 79

We've got a fixed point

Define a state transformer f as follows.

$$f(s) = \begin{cases} F^n(s) & \text{if } F^n(s) \neq \perp \text{ for some } n \\ \perp & \text{otherwise.} \end{cases}$$

This is well-defined and is a fixed point of F.

Slide 80

for all states s.

Then we can define a sequence of *syntactic approximants* to the loop `while` B `do` C as shown on Slide 81.

Approximating a `while`-loop

We define the approximants of `while` B `do` C as follows.

$$\begin{array}{rcl} C_0 & = & \texttt{diverge} \\ C_1 & = & \texttt{if } B \texttt{ then } C\,;\texttt{diverge else skip} \\ & \vdots & \\ C_{n+1} & = & \texttt{if } B \texttt{ then } C\,;C_n \texttt{ else skip} \end{array}$$

This is an inductive definition (mathematical induction on the subscript i of the C_i).

Slide 81

We have argued before that the command C_n has the same effect as `while` B `do` C in those states which require going fewer than n times round the loop to terminate. Let us now prove that this sequence of approximations really does get better as you go on.

Theorem 14 For any natural number n and any state s, if $\mathcal{C}[\![C_n]\!](s) \neq \bot$ then $\mathcal{C}[\![C_{n+1}]\!](s) = \mathcal{C}[\![C_n]\!](s)$.

Proof By induction on n.

Base case: In the case $n = 0$, $C_n =$ `diverge` so it is never the case that $\mathcal{C}[\![C_n]\!](s) \neq \bot$. There is therefore nothing to prove.

Inductive step: Consider the case $n = k + 1$. By definition,

$$\begin{array}{rcl} \mathcal{C}[\![C_{k+1}]\!] & = & \mathcal{C}[\![\texttt{if } B \texttt{ then } (C\,;C_k) \texttt{ else skip}]\!] \\ & = & \mathsf{cond}(\mathcal{B}[\![B]\!], (\mathcal{C}[\![C]\!]\,;\mathcal{C}[\![C_k]\!]), \mathsf{id}). \end{array}$$

Since we are assuming that $\mathcal{C}[\![C_{k+1}]\!](s) \neq \bot$, it cannot be that $\mathcal{B}[\![B]\!](s) = \bot$. It There are therefore two subcases to consider.

- If $\mathcal{B}[\![B]\!](s) = \texttt{false}$, then clearly $\mathcal{C}[\![C_{k+1}]\!](s) = s$, and similarly $\mathcal{C}[\![C_{k+2}]\!](s) = s$, which gives the desired conclusion.
- If $\mathcal{B}[\![B]\!](s) = \texttt{true}$, then $\mathcal{C}[\![C_{k+1}]\!](s) = (\mathcal{C}[\![C]\!] \,;\, \mathcal{C}[\![C_k]\!])(s)$. Since we know this is not $\perp$, it must be the case that $\mathcal{C}[\![C]\!](s) \neq \perp$, so,

  $$\mathcal{C}[\![C_{k+1}]\!](s) = \mathcal{C}[\![C_k]\!](\mathcal{C}[\![C]\!](s)).$$

  By the inductive hypothesis,

  $$\mathcal{C}[\![C_{k+1}]\!](\mathcal{C}[\![C]\!](s)) = \mathcal{C}[\![C_k]\!](\mathcal{C}[\![C]\!](s)).$$

  Putting these two together, we get

  $$\mathcal{C}[\![C_{k+1}]\!](s) = \mathcal{C}[\![C_{k+1}]\!](\mathcal{C}[\![C]\!](s)) = \mathcal{C}[\![C_{k+2}]\!](s)$$

  which is what we needed to prove.

■

We therefore have an improving sequence of approximations to our `while`-loop. We can now define the semantics of `while` B `do` C as follows.

Semantics of `while`

$$\mathcal{C}[\![\texttt{while}\ B\ \texttt{do}\ C]\!](s) = \begin{cases} s', & \text{if any } \mathcal{C}[\![C_k]\!](s) = s' \\ \perp, & \text{otherwise} \end{cases}$$

Slide 82

It should be reasonably obvious that this is the same state transformer we previously claimed was a fixed point of F; see Slide 80.

The preceding theorem tells us that this does indeed define a function. Let us now make use of our semantics of `while` to prove a simple fact. Here we will prove that the state left after the execution of a loop always makes the boolean guard B is false.

We prove this by first showing an appropriate fact about the syntactic approximants to the loop.

Lemma 15 For any natural number n and any state s, if $\mathcal{C}[\![C_n]\!](s) = s'$ then $\mathcal{B}[\![B]\!](s') = \texttt{false}$.

Proof By induction on n.

Base case: In the case $n = 0$, $C_0 = \texttt{diverge}$ so it never holds that $\mathcal{C}[\![C_n]\!](s) = s'$. There is therefore nothing to prove.

Inductive step: Consider the case $n = k + 1$. By definition,

$$\begin{aligned} \mathcal{C}[\![C_{k+1}]\!] &= \mathcal{C}[\![\texttt{if } B \texttt{ then } C \,;\, C_k \texttt{ else skip}]\!] \\ &= \mathsf{cond}(\mathcal{B}[\![B]\!], \mathcal{C}[\![C]\!] \,;\, \mathcal{C}[\![C_k]\!], \mathsf{id}). \end{aligned}$$

So, if $\mathcal{C}[\![C_{k+1}]\!](s) = s'$, there are two cases:

- either $\mathcal{B}[\![B]\!](s) = \texttt{false}$ and $s = s'$, in which case $\mathcal{B}[\![B]\!](s') = \texttt{false}$ as required, or
- $\mathcal{B}[\![B]\!](s) = \texttt{true}$, and then

$$s' = (\mathcal{C}[\![C]\!] \,;\, \mathcal{C}[\![C_k]\!])(s)$$

In this case, it is clear that

$$s' = \mathcal{C}[\![C_k]\!](s'')$$

where $s'' = \mathcal{C}[\![C]\!](s)$. But the inductive hypothesis tells us that any state coming from $\mathcal{C}[\![C_k]\!]$ makes $\mathcal{B}[\![B]\!]$ false, that is

$$\mathcal{B}[\![B]\!](s') = \texttt{false}$$

as required.

■

Theorem 16 If $\mathcal{C}[\![\texttt{while } B \texttt{ do } C]\!](s) = s'$ then $\mathcal{B}[\![B]\!](s') = \texttt{false}$.

Proof By the definition of the semantics of `while`, if $\mathcal{C}[\![\texttt{while } B \texttt{ do } C]\!](s) =$

s' then $s' = \mathcal{C}[\![C_n]\!](s)$ for some n. By the previous lemma, $\mathcal{B}[\![B]\!](s') = \texttt{false}$ as required. ■

Exercise Prove that

- $\mathcal{C}[\![x := y \,;\, y := x]\!] = \mathcal{C}[\![x := y]\!]$.
- $\mathcal{C}[\![x := z \,;\, y := z]\!] = \mathcal{C}[\![y := z \,;\, x := z]\!]$.
- $\mathcal{C}[\![C_1 \,;\, (C_2 \,;\, C_3)]\!] = \mathcal{C}[\![(C_1 \,;\, C_2) \,;\, C_3]\!]$.

Is it the case that $\mathcal{C}[\![x := x]\!] = \mathcal{C}[\![\texttt{skip}]\!]$ in this language? If so, give a proof; if not, give a counterexample.

4.3 Axiomatic Semantics

We have so far seen two distinct approaches to semantics.

operational: explains how to evaluate programs.

denotational: gives meaning to a program by identifying an element of a mathematical structure which "is" the program's meaning.

Using either of these styles of semantics we can prove various facts about programs, verify correctness of programs with respect to their specification, and so on. We have seen a few examples of this.

The final form of semantics that we shall consider is aimed directly at supporting such reasoning and verification.

axiomatic: gives a proof system for verifying program properties.

The axiomatic semantics we shall study is called *Hoare logic*, after its inventor, Tony Hoare. It is sometimes called Floyd-Hoare logic in recognition of an earlier contribution by Floyd.

4.3.1 Partial Correctness

Hoare logic is a system for demonstrating that certain *partial correctness assertions* hold of programs. A partial correctness assertion takes the form shown on Slide 83.

This is called a *partial* correctness assertion because it says nothing about what happens if C fails to terminate, and it says nothing about whether C terminates or not.

Partial correctness assertions

> if we run command C in a starting state which satisfies property P, then *if* C *terminates*, the final state satisfies Q.

We use the notation

$$\{P\}C\{Q\}$$

for this situation. P is called the *precondition*, and Q is the *postcondition*.

Slide 83

Exercise What do you think *total correctness assertions* would be? Can you see why total correctness might be harder to work with than partial correctness? Think about `while` loops...

A question arises immediately: what are we allowed to put in the conditions P and Q? That is, what is the syntax of these predicates? We shall not be too precise about this on this course, but let us ask that at least every Boolean expression from our language *While* may be used. That gives us at the very least things like equality tests, logical "and", "or" and "not", and so on. Sophisticated versions of axiomatic semantics may include $\forall$, $\exists$ and many other kinds of logical operation; let's not worry about that.

Slide 84 gives some examples of simple partial correctness assertions which hold in our language.

The first of these says that assignment really does change the value in a variable; the second says that assignment to x leaves y unchanged. The third says that `skip` changes nothing. The last one is quite interesting. It says

> If you start the program `while true do skip` in any state (more accurately, any state which satisfies `true`, which is any state at all), if it terminates, the final state satisfies `false`.

Since no state satisfies `false`, this is tantamount to saying that the program `while true do skip` never terminates.

Some example assertions

- $\{x = 3\} x := 4 \{x = 4\}$.
- $\{y = 3\} x := 4 \{y = 3\}$.
- $\{P\}$`skip`$\{P\}$.
- $\{$`true`$\}$`while true do skip`$\{$`false`$\}$.

Slide 84

4.3.2 The formal system

We shall now give a collection of axioms and rules for deriving partial correctness assertions. Each construct of the programming language is associated with one axiom or rule; additionally there are some purely logical rules which allow us to manipulate the pre- and postconditions in certain ways.

The best we can say of `skip` is that it never changes anything. We would like to say that it always terminates, but partial correctness does not let us do so. Slide 85 gives the axiom for `skip`, which is the same "do nothing" assertion we saw before.

For sequential composition $C_1 ; C_2$, the obvious rule is that if C_1 takes a state satisfying P to one satisfying Q, and C_2 takes such an intermediate state to one satisfying R, then $C_1 ; C_2$ takes a state satisfying P to one satisfying R. Slide 86 makes this into a rule of our system.

For conditional statements, if we are to guarantee that Q holds after

$$\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2$$

terminates, then we need it to be the case that, if B is true, C_1 makes Q true, and if B is false, C_2 makes Q true. The rule for conditionals is therefore that on Slide 87.

Let us now consider assignment statements of the form $x := E$. Again, let us think backwards. Suppose we want to say that a predicate Q is satisfied

Hoare triple for `skip`

$$\textsc{(h-skip)}\quad \frac{}{\{P\}\texttt{skip}\{P\}}$$

Slide 85

Sequential composition

$$\textsc{(h-seq)}\quad \frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1\,;C_2\{R\}}$$

Slide 86

Conditionals

$$\textsc{(h-cond)}\quad \frac{\{P \wedge B\}C_1\{Q\} \quad \{P \wedge \neg B\}C_2\{Q\}}{\{P\}\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2\{Q\}}$$

Slide 87

after executing $x := E$. Q is a predicate which talks about the new value of x (among other things). What we need as our precondition is a predicate which says the same thing in terms of the old value of x. For example, if we have an assignment like

$$x := (x + x)$$

and we want to ensure that the predicate $x = 4$ holds afterwards, it is enough to ensure that $x + x = 4$ held beforehand. That is, if we replace x by $x + x$, we get the appropriate predicate.

Let us write $Q[E/x]$ for the predicate Q with each occurrence of x replaced by E. In the above example, $(x = 4)[x + x/x]$ is $(x + x) = 4$. Thus $Q[E/x]$ is the “same” predicate as Q but written “in terms of the old value of x”.

This idea gives us the axiom for assignment, which is shown in Slide 88.

Some simple examples of this axiom:

- $\{y + y = 4\}x := y + y\{x = 4\}$.
- $\{y = 3\}x := 4\{y = 3\}$.

Exercise Can we yet show that

$$\{x = 3\}x := 4\{x = 4\}$$

holds?

Assignment

$$(\textsc{h-ass})\quad \frac{}{\{Q[E/x]\}x := E\{Q\}}$$

Slide 88

Let us now consider `while` B `do` C. One thing that we know about this command is that if it terminates, the boolean guard B is false. So here's a valid assertion:

$$\{\texttt{true}\}\texttt{while}\ B\ \texttt{do}\ C\{\neg B\}$$

This doesn't tell us much of interest. To say more, we need a new idea. Roughly speaking, if we want some predicate P to be true after `while` B `do` C terminates, it must be true no matter how many times the loop body is run: since the loop body could be run no times, it must be true at the beginning; and since it could be run once, it must be true after going once round the loop; or twice; or three times; and so on.

This is the idea of a *loop invariant.* A loop invariant is a predicate P which is *preserved* by going round the loop. That is to say, if P holds at the start of a trip round the loop, it should hold afterwards too. As a first guess, we might say P is a loop invariant if

$$\{P\}C\{P\}$$

holds.

This is indeed enough to guarantee that P is a good loop invariant, but we can do better: because the loop body is only executed when B is true, the command C only needs to maintain the truth of P in the case when B holds initially: see Slide 89.

Loop invariant

The predicate P is a loop invariant for the loop $\texttt{while}\ B\ \texttt{do}\ C$ iff

$$\{P \wedge B\}C\{P\}$$

Slide 89

In this situation, we know that if we start $\texttt{while}\ B\ \texttt{do}\ C$ in a state satisfying P, then P will be true every time we finish the loop body. Therefore, if the loop terminates, we will know two things:

- P is true, since P is an invariant
- B is false, since that's the only way the loop can stop.

We therefore have the rule for `while`-loops shown on Slide

Exercise Why does the rule not have the conclusion

$$\{P \wedge B\}\texttt{while}\ B\ \texttt{do}\ C\{P \wedge \neg B\}?$$

Example: `while true do skip` Instantiating the above rule with `while true do skip`, and letting P be `true` too, gives us the following.

$$\frac{\{\texttt{true} \wedge \texttt{true}\}\texttt{skip}\{\texttt{true}\}}{\{\texttt{true}\}\texttt{while true do skip}\{\texttt{true} \wedge \neg\texttt{true}\}}$$

This allows us to deduce that

$$\{\texttt{true}\}\texttt{while true do skip}\{\texttt{false}\}$$

which says that, in any starting state, if the program `while true do skip` terminates, `false` holds. Since `false` never holds, we can conclude that this loop does not terminate, which is a good thing!

Axiomatic semantics of `while`

$$\textsc{(h-while)}\quad \frac{\{P \wedge B\}C\{P\}}{\{P\}\texttt{while}\ B\ \texttt{do}\ C\{P \wedge \neg B\}}$$

Slide 90

Some purely logical rules The rules that we have so far make good sense, but it turns out that as they stand they are not quite good enough to let us prove some of the assertions we expect to hold. By adding a few purely logical rules, we can obtain greater manipulative power.

The rule of consequence, on Slide 91, says that we can use logical implication to alter the precondition and postcondition of an assertion in the expected way.

The first rule on Slide 92 says that if the same postcondition can be established from a number of preconditions, then taking the disjunction of the preconditions (the "or" of them all) also establishes the postcondition. The second rule is a nullary case of this: the precondition `false` is never satisfied, so the assertion in this rule is rather meaningless.

The rules on Slide 93 are similar to those on the previous slide. These say that if several postconditions follow from the same precondition, then the conjunction or "and" of them all also follows. The nullary case of this is that the postcondition `true` may always be concluded: since we are interested in *partial* correctness only, this rule tells us nothing. However, it might sometimes come in handy when using the rules given above as a purely formal system.

Example: factorial again! Slide 94 shows yet another program for computing factorials.

The rule of consequence

$$\textsc{(h-cons)}\quad \frac{P \implies P' \quad \{P'\}C\{Q'\} \quad Q' \implies Q}{\{P\}C\{Q\}}$$

Slide 91

Precondition disjunction

$$\textsc{(h-pre.or)}\quad \frac{\{P_1\}C\{Q\} \cdots \{P_n\}C\{Q\}}{\{P_1 \vee \cdots \vee P_n\}C\{Q\}}$$

$$\textsc{(h-pre.false)}\quad \frac{}{\{\texttt{false}\}C\{Q\}}$$

Slide 92

Postcondition conjunction

$$\text{(H-POST.AND)}\quad \frac{\{P\}C\{Q_1\}\cdots\{P\}C\{Q_n\}}{\{P\}C\{Q_1 \wedge \cdots \wedge Q_n\}}$$

$$\text{(H-POST.TRUE)}\quad \frac{}{\{P\}C\{\mathtt{true}\}}$$

Slide 93

Another factorial program

```
C =  y := 1 ; a := 1;
     while y ≠ x do
       (y := y + 1;
        a := a × y)
```

Slide 94

We are going to show, using the rules given above (but leaving out some of the details), that this program computes the value of $x!$, leaving the result in the variable a.

The partial correctness assertion we want to establish is

$$\{\texttt{true}\}C\{a = x!\}.$$

The interesting part here is the loop invariant. The idea of the loop is that a holds a "running product" of all the numbers between 1 and y, so that when y reaches x, a holds $x!$.

We might therefore guess that the loop invariant should be

$$a = 1 \times \cdots \times y.$$

Therefore our first task is to show that

$$\{(a = 1 \times \cdots \times y) \wedge (y \neq x)\}y := y + \mathtt{1}\,;\, a := a \times y\{a = 1 \times \cdots \times y\}$$

is a valid assertion. Let us call the precondition above P, and the postcondition Q.

We shall show this assertion is valid by applying the rules for assignment and sequential composition. The assignment rule works right-to-left, so we work backwards through the assignments as shown on Slide 95.

Working backwards through assignments

The assertion

$$\{a \times y = 1 \times \cdots \times y\}a := a \times y\{a = 1 \times \cdots \times y\}$$

is an instance of the axiom for assignments: the precondition has been calculated by substituting $a \times y$ for a in the postcondition.

Similarly we can obtain

$$\{a\times(y+1) = 1\times\cdots\times(y+1)\}y := y+\mathtt{1}\{a\times y = 1\times\cdots\times y\}$$

Slide 95

Notice that in doing this calculation, we work from right to left: starting from the postcondition we need to establish, we calculate a precondition for the last command in the sequential composition. This precondition is then used as a postcondition for the previous command, and we calculate its own precondition.

We can now use the rule for sequential composition. First let us define the abbreviations

$$\begin{array}{lll} P_1 & \text{for} & (a \times (y+1) = 1 \times \cdots \times (y+1)) \\ P_2 & \text{for} & (a \times y = 1 \times \cdots \times y) \end{array}$$

Our application of the rule for sequential composition is that shown on Slide 96.

Sequentially composing

$$\frac{\{P_1\}y := y + \mathtt{1}\{P_2\} \quad \{P_2\}a := a \times y\{Q\}}{\{P_1\}y := y + \mathtt{1}\,;\, a := a \times y\{Q\}}$$

Slide 96

This is *almost* what we need. The postcondition, Q, is the right thing. But the precondition P_1 is not the same as the P we need. Our desired precondition P is

$$(a = 1 \times \cdots \times y) \wedge (x \neq y)$$

which certainly implies that

$$a = 1 \times \cdots \times y.$$

By multiplying both sides by $y + 1$, we see that this implies

$$a \times (y+1) = 1 \times \cdots \times (y+1)$$

We therefore use the rule of consequence as shown on Slide 97.

Using the rule of consequence

$$\frac{P \implies P_1 \quad \{P_1\}y := y + \mathtt{1}\,;\, a := a \times y\{Q\}}{\{P\}y := y + \mathtt{1}\,;\, a := a \times y\{Q\}}$$

Slide 97

The loop invariant has therefore been established, so we can use the rule for `while` as shown on Slide 98. Note that P is the same as $Q \wedge (x \neq y)$.

Exercise Show that

$$\{\mathtt{true}\}y := \mathtt{1}\,;\, a := \mathtt{1}\{Q\}$$

is valid.

Once we have also shown that

$$\{\mathtt{true}\}y := \mathtt{1}\,;\, a := \mathtt{1}\{Q\}$$

is valid, we can then use the rule of sequential composition again to conclude that

$$\{\mathtt{true}\}C\{Q \wedge (x = y)\}.$$

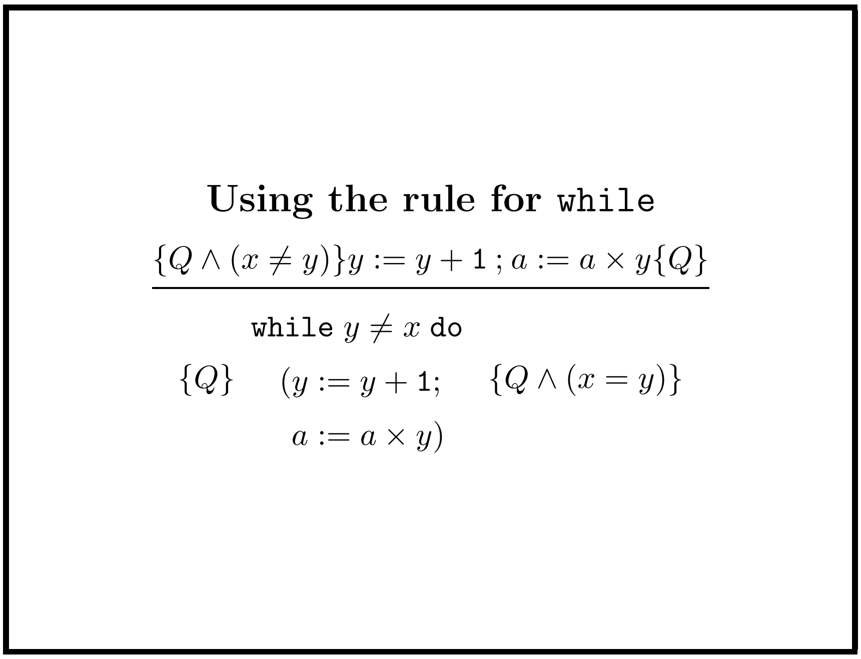
Finally, it is clear that

$$(Q \wedge (x = y)) \implies (a = x!)$$

so by the rule of consequence we conclude that

$$\{\mathtt{true}\}C\{a = x!\}$$

as required.



Using the rule for `while`

$$\frac{\{Q \wedge (x \neq y)\} y := y + \mathtt{1}\, ; a := a \times y \{Q\}}{\{Q\} \quad \begin{array}{l} \mathtt{while}\ y \neq x\ \mathtt{do} \\ \quad (y := y + \mathtt{1}; \\ \quad\ a := a \times y) \end{array} \quad \{Q \wedge (x = y)\}}$$

Slide 98

Exercise There are clearly many commands C which satisfy the assertion

$$\{\mathtt{true}\}C\{a = x!\}$$

since any program which computes the factorial of x and stores the result in a will do. But here is another program which also satisfies the above assertion.

$$x := \mathtt{0}\, ; a := \mathtt{1}$$

- Prove that this program also satisfies the assertion above using the rules of Hoare logic.
- What additional assertion do we need to say that a program really does compute the factorial of the value initially stored in x?

5 Relating the semantics

We have now given three ways of describing and reasoning about programs in the simple language *While*. Our intuition tells us that everything we have done corresponds to the same underlying ideas. It ought to be the case, therefore, that the three different semantics are related in some way.

For the purposes of this course, we will take the view that the denotational semantics is primary: the denotational semantics says what a program "really means". In this case, a program "really means" a partial function from states to states.

We shall prove two *soundness* results.

Soundness for small-step semantics

Given a command C and a state s, if

$$\langle C, s\rangle \rightarrow^* \langle \mathtt{skip}, s'\rangle$$

then

$$\mathcal{C}[\![C]\!](s) = s'.$$

Slide 99

We might also be interested in *completeness* statements like these:

- for the small-step operational semantics, if $\mathcal{C}[\![C]\!](s) = s'$ then $\langle C, s\rangle \rightarrow^* \langle \mathtt{skip}, s'\rangle$.
- for the axiomatic semantics, given predicates P and Q and a command C, if it is the case that for any state s satisfying P, either $\mathcal{C}[\![C]\!](s) = \perp$ or $\mathcal{C}[\![C]\!](s) = s'$ for some s' which satisfies Q, then $\{P\}C\{Q\}$ can be proved using Hoare logic.

The first of these certainly holds, although we will (probably) not prove that it does in this course. The second statement, and things like it, are rather trickier, and certainly beyond the scope of this course.

5.1 Soundness for small-step semantics

The soundness theorem for small-step semantics talks about what happens over many steps of evaluation. To prove it, we will work with single steps of evaluation, by induction on their derivation, as usual.

Soundness for axiomatic semantics

If

$$\{P\}C\{Q\}$$

can be proved, then

if s is any state satisfying P and $\mathcal{C}[\![C]\!](s) = s'$,

then s' is a state satisfying Q.

Slide 100

First of all, since our denotational semantics does not take account of any state-changes caused by evaluating expressions or booleans, we need to show that there are in fact no state changes in the small-step semantics.

Lemma 17 For any expression E and state s, if $\langle E, s\rangle \rightarrow \langle E', s'\rangle$ then $s = s'$. A similar statement holds for booleans.

Exercise Prove this lemma.

We now show that doing a step of the small-step semantics does not change the denotational "final answer". See Slide 101.

We shall prove this lemma by induction on the derivation of the step $\langle C, s\rangle \rightarrow \langle C', s'\rangle$. Our proof technique is the usual one: for each rule (or axiom), assume that the steps above the line satisfy this condition, and prove that the step below the line does too. We will not organise this proof into base cases and inductive steps, since it is better organised by considering the semantics of each operation of the language in turn.

Assignment For assignment, there is one axiom and one rule. The axiom is

$$\frac{}{\langle x := \mathtt{n}, s\rangle \rightarrow \langle \mathtt{skip}, s[x \mapsto n]\rangle}$$

so we must prove that

$$\mathcal{C}[\![x := \mathtt{n}]\!](s) = \mathcal{C}[\![\mathtt{skip}]\!](s[x \mapsto n]).$$

Single step soundness

Lemma 18 For any command C and state s, if $\langle C, s\rangle \rightarrow \langle C', s'\rangle$, then $\mathcal{C}[\![C]\!](s) = \mathcal{C}[\![C']\!](s')$. Similar statements hold for the expressions and booleans.

Slide 101

But by definition, $\mathcal{C}[\![\texttt{skip}]\!] = \mathsf{id}$, so

$$\mathcal{C}[\![\texttt{skip}]\!](s[x \mapsto n]) = s[x \mapsto n]$$

and by definition of the semantics of assignment,

$$\mathcal{C}[\![x := \texttt{n}]\!](s) = s[x \mapsto n]$$

as required.

The rule is

$$\frac{\langle E, s\rangle \rightarrow \langle E', s'\rangle}{\langle x := E, s\rangle \rightarrow \langle x := E', s'\rangle}$$

We must prove that

$$\mathcal{C}[\![x := E]\!](s) = \mathcal{C}[\![x := E']\!](s')$$

using the assumption that

$$\mathcal{E}[\![E]\!](s) = \mathcal{E}[\![E']\!](s').$$

By the previous lemma, $s = s'$. By definition of the semantics of assignment,

$$\begin{array}{rcl} \mathcal{C}[\![x := E]\!](s) & = & s[x \mapsto \mathcal{E}[\![E]\!](s)] \\ & = & s'[x \mapsto \mathcal{E}[\![E']\!](s')] \\ & = & \mathcal{C}[\![x := E']\!](s') \end{array}$$

as required.

Sequential Composition Again we have one axiom and one rule to consider. The axiom is

$$\frac{}{\langle \texttt{skip}\,;C, s\rangle \rightarrow \langle C, s\rangle}$$

so we must prove that

$$\mathcal{C}[\![\texttt{skip}\,;C]\!](s) = \mathcal{C}[\![C]\!](s).$$

By definition,

$$\begin{array}{rcl}
\mathcal{C}[\![\texttt{skip}\,;C]\!](s) & = & (\mathcal{C}[\![\texttt{skip}]\!]\,;\mathcal{C}[\![C]\!])(s) \\
& = & \mathcal{C}[\![C]\!](\mathsf{id}(s)) \\
& = & \mathcal{C}[\![C]\!](s)
\end{array}$$

as required.

The rule is

$$\frac{\langle C_1, s\rangle \rightarrow \langle C_1', s'\rangle}{\langle C_1\,;C_2, s\rangle \rightarrow \langle C_1'\,;C_2, s'\rangle}$$

so we must prove that

$$\mathcal{C}[\![C_1\,;C_2]\!](s) = \mathcal{C}[\![C_1'\,;C_2]\!](s')$$

using the assumption that

$$\mathcal{C}[\![C_1]\!](s) = \mathcal{C}[\![C_1']\!](s').$$

There are two cases: either $\mathcal{C}[\![C_1]\!](s) = \perp$, in which case $\mathcal{C}[\![C_1;C_2]\!](s) = \mathcal{C}[\![C_1'\,;C_2]\!](s') = \perp$, or $\mathcal{C}[\![C_1]\!](s) \neq \perp$. In this case,

$$\begin{array}{rcl}
\mathcal{C}[\![C_1\,;C_2]\!](s) & = & \mathcal{C}[\![C_2]\!](\mathcal{C}[\![C_1]\!](s)) \\
& = & \mathcal{C}[\![C_2]\!](\mathcal{C}[\![C_1']\!](s')) \\
& = & \mathcal{C}[\![C_1'\,;C_2]\!](s').
\end{array}$$

The result therefore follows.

Skip There is no rule for `skip` since it makes no steps, so there is nothing to prove.

Conditional The cases for the conditional follow in a similar way to those for sequential composition and assignment, so we omit them.

While The rule for `while` is

$$\frac{}{\begin{array}{l}\langle \texttt{while}\ B\ \texttt{do}\ C, s\rangle \rightarrow \\ \langle \texttt{if}\ B\ \texttt{then}\ (C\,;\texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}, s\rangle\end{array}}$$

so we must prove that

$$\mathcal{C}[\![\texttt{while}\ B\ \texttt{do}\ C]\!](s) = \mathcal{C}[\![\texttt{if}\ B\ \texttt{then}\ (C\ ;\texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}]\!](s)$$

This is exactly the equation that we originally set out to solve when defining the denotational semantics of `while`. We didn't quite give the proof that it is satisfied, but...

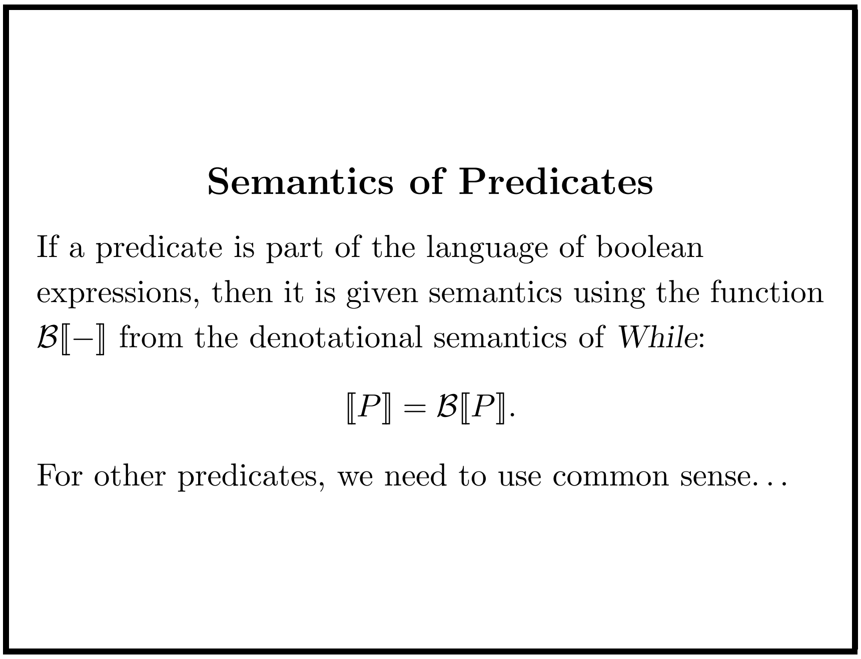
5.2 Soundness for Axiomatic Semantics

Our final task in this course is to prove that the axiomatic soundness only lets us derive partial correctness assertions that are *true* with respect to the operational semantics.

First of all, we need to say what that means.

5.2.1 Semantics of Predicates

We first need to say what the denotational semantics of a predicate is. Since we have not been specific about what predicates we allow, this is a bit tricky! Slide 102 shows a slightly informal approach.



Semantics of Predicates

If a predicate is part of the language of boolean expressions, then it is given semantics using the function $\mathcal{B}[\![-]\!]$ from the denotational semantics of *While*:

$$[\![P]\!] = \mathcal{B}[\![P]\!].$$

For other predicates, we need to use common sense...

Slide 102

There is a lot of room for extra precision.

Examples

- $[\![x = 3]\!](s) = \texttt{true}$ iff $s(x) = 3$.
- $[\![x = y]\!](s) = \texttt{true}$ iff $s(x) = s(y) \neq \bot$.

Slide 103

5.2.2 Semantics of assertions

We can now say what the semantics of a partial correctness assertion $\{P\}C\{Q\}$ is. Slide 104 contains the definition.

We can now state our soundness theorem, as follows.

Theorem 19 If a partial correctness assertion $\{P\}C\{Q\}$ can be derived using the axioms and rules of Hoare logic, then $[\![\{P\}C\{Q\}]\!] = \texttt{true}$.

To prove this, we again use the method of induction over the derivation of the assertion $\{P\}C\{Q\}$.

Skip The axiom for `skip` gives us the assertion $\{P\}\texttt{skip}\{P\}$. By definition, $\mathcal{C}[\![\texttt{skip}]\!](s) = s$ for any state s. So it is clear that if $[\![P]\!](s) = \texttt{true}$ then $[\![P]\!](s) = \texttt{true}$!

Sequential Composition The rule for sequential composition is

$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1\,;C_2\{R\}}$$

Suppose that $[\![P]\!](s) = \texttt{true}$ and $\mathcal{C}[\![C_1\,;C_2]\!](s) = s'$. By definition of the denotational semantics, there must be some s'' such that

$$\begin{array}{rcl} \mathcal{C}[\![C_1]\!](s) & = & s'' \\ \mathcal{C}[\![C_2]\!](s'') & = & s'. \end{array}$$

Semantics of Assertions

We define $[\![\{P\}C\{Q\}]\!] = \texttt{true}$ if and only if

> for all states s and s', if $[\![P]\!](s) = \texttt{true}$ and $\mathcal{C}[\![C]\!](s) = s'$, then $[\![Q]\!](s') = \texttt{true}$.

In any other case, $[\![\{P\}C\{Q\}]\!] = \texttt{false}$.

Slide 104

Soundness of Axiomatic Semantics

If a partial correctness assertion $\{P\}C\{Q\}$ can be derived using the axioms and rules of Hoare logic, then $[\![\{P\}C\{Q\}]\!] = \texttt{true}$.

Slide 105

Using the inductive hypothesis, we know that $[\![\{P\}C_1\{Q\}]\!] = \texttt{true}$ and $[\![\{Q\}C_2\{R\}]\!] = \texttt{true}$. This means that $[\![Q]\!](s'') = \texttt{true}$, and therefore that $[\![R]\!](s') = \texttt{true}$ as required.

Conditional The rule for the conditional is

$$\frac{\{P \wedge B\}C_1\{Q\} \quad \{P \wedge \neg B\}C_2\{Q\}}{\{P\}\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2\{Q\}.}$$

Suppose that $[\![P]\!](s) = \texttt{true}$ and $\mathcal{C}[\![\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2]\!](s) = s'$ for some states s and s'. There are two cases: either $\mathcal{B}[\![B]\!](s) = \texttt{true}$ or $\mathcal{B}[\![B]\!](s) = \texttt{false}$. (The only other possibility is that $\mathcal{B}[\![B]\!](s) = \bot$, but then $\mathcal{C}[\![\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2]\!](s) = \bot$ too, and we've assumed that is not the case.)

- If $\mathcal{B}[\![B]\!](s) = \texttt{true}$ then we know that $[\![P \wedge B]\!](s) = \texttt{true}$, so applying the inductive hypothesis to the first assertion above the line tells us that if $\mathcal{C}[\![C_1]\!](s) = s''$ then $[\![Q]\!](s'') = \texttt{true}$. But in this case, we also know that $\mathcal{C}[\![\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2]\!](s) = \mathcal{C}[\![C_1]\!](s) = s'$, so it follows that $[\![Q]\!](s') = \texttt{true}$ as required.
- The case of $\mathcal{B}[\![B]\!](s) = \texttt{false}$ is similar.

Assignment The axiom for assignment is

$$\frac{}{\{Q[E/x]\}x := E\{Q\}.}$$

To prove that this is sound, we must prove that if $[\![Q[E/x]]\!](s) = \texttt{true}$ then

$$[\![Q]\!](s[x \mapsto \mathcal{E}[\![E]\!](s)]) = \texttt{true},$$

for every predicate Q, variable x, expression E and state s. To do this properly we would use induction on the structure of the predicate. We have not made the syntax of predicates precise, so we can't do that formally. We therefore omit a proof of this case, leaving it to your intuition.

While The rule for `while` is

$$\frac{\{P \wedge B\}C\{P\}}{\{P\}\texttt{while } B \texttt{ do } C\{P \wedge \neg B\}}$$

Let us suppose that $[\![P]\!](s) = \texttt{true}$ and that $\mathcal{C}[\![\texttt{while } B \texttt{ do } C]\!](s) = s'$. We must show that $[\![P \wedge \neg B]\!](s) = \texttt{true}$, using the assumption that $[\![\{P \wedge B\}C\{P\}]\!] = \texttt{true}$.

By definition of the semantics of `while`, $s' = \mathcal{C}[\![C_i]\!](s)$ for one of the approximants C_n. It is therefore enough for us to show that the assertion

$$\{P\}C_n\{P \wedge \neg B\}$$

is semantically valid for each n. This we do by induction on n.

For the base case, since $\mathcal{C}[\![C_0]\!](s) = \perp$ always, there is nothing to prove.

For the inductive step, suppose that $\{P\}C_k\{P \wedge \neg B\}$ is valid. By definition, C_{k+1} is the command

$$\texttt{if } B \texttt{ then } C\,;C_k \texttt{ else skip}$$

and we need to show that $\{P\}C_{k+1}\{P \wedge \neg B\}$ is semantically valid. Since we have already argued that the rules for `skip`, sequential composition and conditional preserve semantic validity, we can use these rules to help us. The rule for conditional tells us that it will be enough to show that both

$$\{P \wedge B\}C\,;C_k\{P \wedge \neg B\}$$

and

$$\{P \wedge \neg B\}\texttt{skip}\{P \wedge \neg B\}$$

are valid. The second of these obviously holds: it is an instance of the rule for `skip`. For the first, the rule for sequential composition tells us that it will be enough to find some predicate Q such that both

$$\{P \wedge B\}C\{Q\}$$

and

$$\{Q\}C_k\{P \wedge \neg B\}$$

are valid.

But by the inductive hypothesis for this inner induction,

$$\{P\}C_k\{P \wedge \neg B\}$$

is valid, and by the inductive hypothesis for the outer induction, so is

$$\{P \wedge B\}C\{P\}.$$

So letting Q be P completes the inner induction, and hence the proof for this case.

Logical rules The purely logical rules are very straightforward to verify, so we omit their proofs.

So, both the operational and axiomatic semantics are correct with respect to the denotational semantics. We have therefore given an account of the meaning of simple programs in terms of state transformers which can be used to show that a simple kind of interpreter (the small-step semantics) works properly, and that certain reasoning principles (Hoare logic) are valid. Thus we have a reasonably complete semantic theory for these very simple programs.

6 The end

And, er, that's it. Just for fun, why not think about how you might give a small-step semantics to an extension of our language *While* with commands like

$$C_1 \| C_2$$

which runs the commands C_1 and C_2 "in parallel". The idea is to simulate *concurrent* execution by *interleaved* execution, which is to say that at any given time, a computation step of $C_1 \| C_2$ may be a step of C_1 or of C_2. What happens to the property of determinacy? And can you give a corresponding denotational semantics?

6.1 Why Semantics Matters

This course has been about a formal, mathematical approach to semantics. The greatest advantage of formality is that if you don't take anything for granted, you won't miss any subtle points. Of course, many of the really useful discoveries one can make by studying semantics formally can also be made by thinking hard about programs without the aid of a formalism.

Here are another couple of examples of places where real programming languages have semantic quirks which can trip you up if you're not careful.

We all know that

$$(x + (y + z)) = ((x + y) + z).$$

But in Java, the expressions

```
(x + (y + z))
```

and

```
((x + y) + z)
```

are *not* equivalent.

To see why, consider the following code:

```
int x = 3;
int y = 7;
String z = "";
System.out.println((x + y) + z); // prints out 10
System.out.println(x + (y + z)); // prints out 37
```

Is this cheating? Not really: the semantics of + in Java is complicated because it is an overloaded operator. As a programmer, you probably knew this already. But if we gave a formal semantics to Java, we would *have* to make the distinction between the numeric + and the string + explicit, so that there was no confusion.

The second example illustrates the need to understand evaluation order, strictness and so on when programming.

Java's `&&` operator, for logical "and", is a left-to-right, short-circuit operator. That is, in an expression

```
x && y
```

if `x` evaluates to `false`, Java does not evaluate `y`. However, method parameters are passed *by value*, which means they are always evaluated. Therefore, a method like

```
 boolean logicalAnd(boolean x, boolean y) {
  return (x && y);
 }
```

is *not* equivalent to `&&`. Can you come up with an example context to distinguish them?

References

[NBB$^+$63] Peter Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauer, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, January 1963.