P

Lecture Notes on

Semantics of Programming Languages

for Part IB of the Computer Science Tripos

Andrew M. Pitts
University of Cambridge
Computer Laboratory

First edition 1997.
Revised 1998,1999, 1999bis, 2000, 2002 .

Contents

Learning Guide

These notes are designed to accompany 12 lectures on programming language semantics for Part IB of the Cambridge University Computer Science Tripos. The aim of the course is to introduce the structural, operational approach to programming language semantics. (An alternative, more mathematical approach and its relation to operational semantics, is introduced in the Part II course on **Denotational Semantics**.) The course shows how this formalism is used to specify the meaning of some simple programming language constructs and to reason formally about semantic properties of programs. At the end of the course you should:

- be familiar with rule-based presentations of the operational semantics of some simple imperative, functional and interactive program constructs;
- be able to prove properties of an operational semantics using various forms of induction (mathematical, structural, and rule-based);
- and be familiar with some operationally-based notions of semantic equivalence of program phrases and their basic properties.

The dependency between the material in these notes and the lectures will be something like:

section	1	2	3	4	5	6
lectures	1	2	3–4	5–6	7–9	10–12.

Tripos questions

Of the many past Tripos questions on programming language semantics, here are those which are relevant to the current course.

Year	01	01	00	00	99	99	98	98	97	97	96
Paper	5	6	5	6	5	6	5	6	5	6	5
Question	9	9	9	9	9	9	12	12	12	12	12
Year	95	94	93	92	91	90	90	88	88	87	
Paper	6	7	7	7	7	7	8	2	4	2	
Question	12	13	10	9	5	4	11	1	$2^\dagger$	$1^\ddagger$	

† not part (c)
‡ not part (b)

In addition, some exercises are given at the end of most sections. The harder ones are indicated with a $*$.

Recommended books

- Winskel, G. (1993). *The Formal Semantics of Programming Languages*. MIT Press.

  This is an excellent introduction to both the operational and denotational semantics of programming languages. As far as this course is concerned, the relevant chapters are 2–4, 9 (sections 1,2, and 5), 11 (sections 1,2,5, and 6) and 14.

- Hennessy, M. (1990). *The Semantics of Programming Languages*. Wiley.

  The book is subtitled 'An Elementary Introduction using Structural Operational Semantics' and as such is a very good introduction to many of the key topics in this course, presented in a more leisurely and detailed way than Winskel's book. The book is out of print, but a version of it is availble on the web at ⟨`www.cogs.susx.ac.uk/users/matthewh/semnotes.ps.gz`⟩.

Further reading

- Gunter, C. A. (1992). *Semantics of Programming Languages. Structures and Techniques*. MIT Press.

  This is a graduate-level text containing much material not covered in this course. I mention it here because its first, introductory chapter is well worth reading.

- Plotkin, G. D.(1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

  These notes first popularised the 'structural' approach to operational semantics—the approach emphasised in this course—but couched solely in terms of transition relations ('small-step' semantics), rather than evaluation relations ('big-step', 'natural', or 'relational' semantics). Although somewhat dated and hard to get hold of (the Computer Laboratory Library has a copy), they are still a mine of interesting examples.

- The two essays:
Hoare, C. A. R.. Algebra and Models.
Milner, R. Semantic Ideas in Computing.
In: Wand, I. and R. Milner (Eds) (1996). *Computing Tomorrow*. CUP.

  Two accessible essays giving somewhat different perspectives on the semantics of computation and programming languages.

Note

The material in these notes has been drawn from several different sources, including the books mentioned above, previous versions of this course by the author and by others, and similar courses at some other universities. Any errors are of course all the author's own work. A list of corrections will be available from the course web page (follow links from ⟨`www.cl.cam.ac.uk/Teaching/`⟩). A lecture(r) appraisal form is included at the end of the notes. Please take time to fill it in and return it. Alternatively, fill out an electronic version of the form via the URL ⟨www.cl.cam.ac.uk/cgi − bin/lr/login⟩.

Andrew Pitts
⟨amp12@cl.cam.ac.uk⟩

1 Introduction

1.1 Operational semantics

Some aspects of the design and use of programming languages are shown on Slide 1. The mathematical tools for precisely specifying *syntax* (regular expressions, context free grammars, BNF, etc) are by now well understood and widely applied: you meet this theory in the Part IA course **Regular Languages and Finite Automata** and see how it is applied in the Part IB **Compiler Construction** course. By contrast, effective techniques for precisely specifying the run-time behaviour of programs have proved much harder to develop. It is for this reason that a programming language's documentation very often gives only informal definitions (in natural language) of the meaning of the various constructs, backed up by example code fragments. But there are good reasons for wanting to go further than this and give a fully formal, mathematical definition of a language's semantics; some of these reasons are summarised on Slide 2.

Constituents of programming language definition

Syntax The alphabet of symbols and a formal description of the well-formed expressions, phrases, programs, etc.

Pragmatics Description and examples of how the various features of the language are intended to be used. Implementation of the language (compilers and interpreters). Auxiliary tools (syntax checkers, debuggers, etc.).

Semantics The *meaning* of the language's features (e.g. their run-time behaviour)—all too often only specified informally, or via the previous heading.

Slide 1

Uses of formal, mathematical semantics

Implementation issues. Machine-independent specification of behaviour. Correctness of program analyses and optimisations.

Verification. Basis of methods for reasoning about program properties and program specifications.

Language design. Can bring to light ambiguities and unforeseen subtleties in programming language constructs. Mathematical tools used for semantics can suggest useful new programming styles. (E.g. influence of Church's lambda calculus (circa 1934) on functional programming).

Slide 2

Styles of semantics

Denotational Meanings for program phrases defined abstractly as elements of some suitable mathematical structure.

Axiomatic Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties.

Operational Meanings for program phrases defined in terms of the steps of computation they can take during program execution.

Slide 3

Some different approaches to programming language semantics are summarised on Slide 3. This course will be concerned with *Operational Semantics*. The denotational approach (and its relation to operational semantics) is introduced in the Part II course on **Denotational Semantics**. Examples of the axiomatic approach occur in the Part II course on **Specification and Verification I**. Each approach has its advantages and disadvantages and there are strong connections between them. However, it is good to start with operational semantics because it is easier to relate operational descriptions to practical concerns and the mathematical theory underlying such descriptions is often quite concrete. For example, some of the operational descriptions in this course will be phrased in terms of the simple notion of a *transition system*, defined on Slide 4.

Transition systems defined

A *transition system* is specified by

- a set $Config$, and
- a binary relation $\rightarrow \subseteq Config \times Config$.

The elements of $Config$ are often called *configurations* (or 'states'), and the binary relation is written infix, i.e. $\boxed{c \rightarrow c'}$ means c and c' are related by $\rightarrow$.

Slide 4

Definition 1.1.1. Here is some notation and terminology commonly used in connection with a transition system $(Config, \rightarrow)$.

(i) $\rightarrow^*$ denotes the binary relation on $Config$ which is the *reflexive-transitive* closure of $\rightarrow$. In other words $c \rightarrow^* c'$ holds just in case

$$c = c_0 \rightarrow c_1 \rightarrow \ldots \rightarrow c_{n-1} \rightarrow c_n = c'$$

holds for some $c_0, \ldots, c_n \in Config$ (where $n \geq 0$; the case $n = 0$ just means $c = c'$).

(ii) $c \not\rightarrow$ means that there is no $c' \in Config$ for which $c \rightarrow c'$ holds.

(iii) The transition system is said to be *deterministic* if for all $c, c_1, c_2 \in Config$

$$c \rightarrow c_1 \;\&\; c \rightarrow c_2 \Rightarrow c_1 = c_2.$$

(The term 'monogenic' is perhaps more appropriate, but less commonly used for this property.)

(iv) Very often the structure of a transition system is augmented with distinguished subsets I and T of Config whose elements are called *initial* and *terminal* configurations respectively. ('Final' is a commonly used synonym for 'terminal' in this context.) The idea is that a pair (i, t) of configurations with $i \in I, t \in T$ and $i \to^* t$ represents a 'run' of the transition system. It is usual to arrange that if $c \in T$ then $c \not\to$; configurations satisfying $c \notin T \; \& \; c \not\to$ are said to be *stuck*.

1.2 An abstract machine

Historically speaking, the first approach to giving mathematically rigorous operational semantics to programming languages was in terms of a suitable *abstract machine*—a transition system which specifies an interpreter for the programming language. We give an example of this for a simple Language of Commands, which we call LC.[1] The abstract machine we describe is often called the *SMC-machine* (e.g. in Plotkin 1981, 1.5.2). The name arises from the fact that its configurations can be defined as triples (S, M, C), where S is a *S*tack of (intermediate and final) results, M is a *M*emory, i.e. an assignment of integers to some finite set of locations, and C is a *C*ontrol stack of phrases to be evaluated. So the name is somewhat arbitrary. We prefer to call memories *states* and to order the components of a configuration differently, but nevertheless we stick with the traditional name 'SMC'.

LC **Syntax**

Phrases

$$P ::= C \mid E \mid B$$

Commands

$$\begin{array}{lll} C & ::= & \mathbf{skip} \mid \ell := E \mid C \,;\, C \\ & \mid & \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ B\ \mathbf{do}\ C \end{array}$$

Integer expressions

$$E ::= n \mid {!\ell} \mid E \; iop \; E$$

Boolean expressions

$$B ::= b \mid E \; bop \; E$$

Slide 5

[1] LC is essentially the same as $\mathbf{IMP}$ in Winskel 1993, 2.1 and WhileL in Hennessy 1990, 4.3.

LC is a very simple language for describing, in a structured way, computable algorithms on numbers via the manipulation of state. In this context we can take a 'state' to consist of a finite number of locations (registers) for storing integers. LC integer and boolean expressions are notations for state-dependent integer and boolean values; LC commands are notations for state-manipulating operations. The syntax of LC phrases is given on Slide 5, where

- $n \in \mathbb{Z} \stackrel{\text{def}}{=} \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the set of *integers*;
- $b \in \mathbb{B} \stackrel{\text{def}}{=} \{\mathbf{true}, \mathbf{false}\}$, the set of *booleans*;
- $\ell \in \mathbb{L} \stackrel{\text{def}}{=} \{\ell_0, \ell_1, \ell_2, \ell_3, \ldots\}$ a fixed, infinite set of symbols whose elements we will call *locations* (the term *program variable* is also commonly used), because they denote locations for storing integers—the integer expression $!\ell$ denotes the integer currently stored in ℓ;
- $iop \in \mathbb{I}op \stackrel{\text{def}}{=} \{+, -, *, \ldots\}$ a fixed, finite set of integer-valued binary operations;
- $bop \in \mathbb{B}op \stackrel{\text{def}}{=} \{=, <, >, \ldots\}$ a fixed, finite set of boolean-valued binary operations.

SMC-machine configurations

are triples $\langle c, r, s\rangle$ consisting of

- a *C*ontrol stack $c ::=$

$$\mathbf{nil} \mid P \cdot c \mid iop \cdot c \mid bop \cdot c \mid := \cdot\, c \mid \mathbf{if} \cdot c \mid \mathbf{while} \cdot c$$

- a *S*tack of (intermediate and final) results

$$r ::= \mathbf{nil} \mid P \cdot r \mid \ell \cdot r$$

- a *M*emory state, s, which by definition is a partial function mapping locations to integers, defined only at finitely many locations.

Slide 6

The set T of configurations of the SMC machine is defined on Slide 6 and its transition relation is defined in Figure 1. It is not hard to see that this is a deterministic transition system: the head of the control stack c uniquely determines which type of transition applies next (if any), unless the head is **if** or $\mathbf{while}$, in which case the head of the phrase stack p determines which transition applies.

The SMC-machine can be used to execute LC commands for their effects on state (in turn involving the evaluation of LC integer and boolean expressions). We define:

initial configurations to be of the form $\langle C \cdot \mathbf{nil}, \mathbf{nil}, s\rangle$ where C is an LC command and s is a state;

terminal configurations to be of the form $\langle \mathbf{nil}, \mathbf{nil}, s\rangle$ where s is a state.

Then existence of a run of the SMC-machine, $\langle C \cdot \mathbf{nil}, \mathbf{nil}, s\rangle \to^* \langle \mathbf{nil}, \mathbf{nil}, s'\rangle$, provides a precise definition of what it means to say that "C executed in state s terminates successfully producing state s'". Some of the transitions in an example run are shown on Slide 7.

$$
\begin{array}{lll}
 & \langle C \cdot \mathbf{nil}, \mathbf{nil}, s\rangle & (\textit{Iteration}) \\
\to & \langle B \cdot \mathbf{while} \cdot \mathbf{nil}, B \cdot C' \cdot \mathbf{nil}, s\rangle & (\textit{Compound}) \\
\to & \langle !\ell \cdot 0 \cdot > \cdot \mathbf{while} \cdot \mathbf{nil}, B \cdot C' \cdot \mathbf{nil}, s\rangle & (\textit{Location}) \\
\to & \langle 0 \cdot > \cdot \mathbf{while} \cdot \mathbf{nil}, 4 \cdot B \cdot C' \cdot \mathbf{nil}, s\rangle & (\textit{Constant}) \\
\to & \langle > \cdot \mathbf{while} \cdot \mathbf{nil}, 0 \cdot 4 \cdot B \cdot C' \cdot \mathbf{nil}, s\rangle & (\textit{Operator}) \\
\to & \langle \mathbf{while} \cdot \mathbf{nil}, \mathbf{true} \cdot B \cdot C' \cdot \mathbf{nil}, s\rangle & (\textit{While-True}) \\
\to & \langle C' \cdot C \cdot \mathbf{nil}, \mathbf{nil}, s\rangle & \ldots \\
\to^* & \langle \mathbf{nil}, \mathbf{nil}, s[\ell \mapsto 0, \ell' \mapsto 24]\rangle &
\end{array}
$$

$$
\text{where} \begin{cases}
C & \stackrel{\text{def}}{=} \mathbf{while}\ B\ \mathbf{do}\ C', \\
B & \stackrel{\text{def}}{=} !\ell > 0, \\
C' & \stackrel{\text{def}}{=} \ell' := !\ell * !\ell' \ ; \ell := !\ell - 1, \\
s & \stackrel{\text{def}}{=} \{\ell \mapsto 4, \ell' \mapsto 1\}.
\end{cases}
$$

Slide 7

	Integer expressions		
Constant	$\langle n \cdot c, r, s\rangle$	$\rightarrow$	$\langle c, n \cdot r, s\rangle$
Location (1)	$\langle !\ell \cdot c, r, s\rangle$	$\rightarrow$	$\langle c, n \cdot r, s\rangle$ if $s(\ell) = n$
Compound	$\langle (E_1\ iop\ E_2) \cdot c, r, s\rangle$	$\rightarrow$	$\langle E_1 \cdot E_2 \cdot iop \cdot c, r, s\rangle$
Operator (2)	$\langle iop \cdot c, n_2 \cdot n_1 \cdot r, s\rangle$	$\rightarrow$	$\langle c, n \cdot r, s\rangle$ if $n_1\ iop\ n_2 = n$
	Boolean expressions		
Constant	$\langle b \cdot c, r, s\rangle$	$\rightarrow$	$\langle c, b \cdot r, s\rangle$
Compound	$\langle (E_1\ bop\ E_2) \cdot c, r, s\rangle$	$\rightarrow$	$\langle E_1 \cdot E_2 \cdot bop \cdot c, r, s\rangle$
Operator (2)	$\langle bop \cdot c, n_2 \cdot n_1 \cdot r, s\rangle$	$\rightarrow$	$\langle c, b \cdot r, s\rangle$ if $n_1\ bop\ n_2 = b$
	Commands		
Skip	$\langle \mathbf{skip} \cdot c, r, s\rangle$	$\rightarrow$	$\langle c, r, s\rangle$
Assignment	$\langle (\ell := E) \cdot c, r, s\rangle$	$\rightarrow$	$\langle E \cdot := \cdot c, \ell \cdot r, s\rangle$
Assign (3)	$\langle := \cdot c, n \cdot \ell \cdot r, s\rangle$	$\rightarrow$	$\langle c, r, s[\ell \mapsto n]\rangle$
Conditional	$\langle (\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2) \cdot c, r, s\rangle$	$\rightarrow$	$\langle B \cdot \mathbf{if} \cdot c, C_1 \cdot C_2 \cdot r, s\rangle$
If-True	$\langle \mathbf{if} \cdot c, \mathbf{true} \cdot C_1 \cdot C_2 \cdot r, s\rangle$	$\rightarrow$	$\langle C_1 \cdot c, r, s\rangle$
If-False	$\langle \mathbf{if} \cdot c, \mathbf{false} \cdot C_1 \cdot C_2 \cdot r, s\rangle$	$\rightarrow$	$\langle C_2 \cdot c, r, s\rangle$
Sequencing	$\langle (C_1 ; C_2) \cdot c, r, s\rangle$	$\rightarrow$	$\langle C_1 \cdot C_2 \cdot c, r, s\rangle$
Iteration	$\langle (\mathbf{while}\ B\ \mathbf{do}\ C) \cdot c, r, s\rangle$	$\rightarrow$	$\langle B \cdot \mathbf{while} \cdot c, B \cdot C \cdot r, s\rangle$
While-True	$\langle \mathbf{while} \cdot c, \mathbf{true} \cdot B \cdot C \cdot r, s\rangle$	$\rightarrow$	$\langle C \cdot (\mathbf{while}\ B\ \mathbf{do}\ C) \cdot c, r, s\rangle$
While-False	$\langle \mathbf{while} \cdot c, \mathbf{false} \cdot B \cdot C \cdot r, s\rangle$	$\rightarrow$	$\langle c, r, s\rangle$

Notes

(1) The side condition means: the partial function s is defined at ℓ and has value n there.

(2) The side conditions mean that n and b are the (integer and boolean) values of the operations iop and bop at the integers n_1 and n_2. The SMC-machine abstracts away from the details of how these basic arithmetic operations are actually calculated. *Note the order of arguments* $(n_2 \cdot n_1)$ *on the left-hand side!*

(3) The state $s[\ell \mapsto n]$ is the finite partial function that maps ℓ to n and otherwise acts like s.

Figure 1: SMC-machine transitions

Informal Semantics

Here is the informal definition of

$$\mathbf{while}\ B\ \mathbf{do}\ C$$

adapted from B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (Prentice-Hall, 1978), p 202:

> The command C is executed repeatedly so long as the value of the expression B remains $\mathtt{true}$. The test takes place before each execution of the command.

Slide 8

Aims of Plotkin's Structural Operational Semantics

Transition systems should be structured in a way that reflects the structure of the language: the possible transitions for a compound phrase should be built up inductively from the transitions for its constituent subphrases.

At the same time one tries to increase the clarity of semantic definitions by minimising the role of ad hoc, phrase-analysis transitions and by making the configurations of the transition system as simple (abstract) as possible.

Slide 9

1.3 Structural operational semantics

The SMC-machine is quite representative of the notion of an abstract machine for executing programs step-by-step. It suffers from the following defects, which are typical of this approach to operational semantics based on the use of abstract machines.

- Only a few of the transitions really perform computation, the rest being concerned with phrase analysis.
- There are many stuck configurations which (we hope) are never reached starting from an initial configuration. (E.g. $\langle \mathbf{if} \cdot c, 2 \cdot C_1 \cdot C_2 \cdot r, s\rangle$.)
- The SMC-machine does not directly formalise our intuitive understanding of the LC control constructs (such as that for **while**-loops given on Slide 8). Rather, it is more or less clearly correct on the basis of this intuitive understanding.
- The machine has "a tendency to pull the syntax to pieces or at any rate to wander around the syntax creating various complex symbolic structures which do not seem particularly forced by the demands of the language itself" (to quote Plotkin 1981, page 21). For this reason, it is quite hard to use the machine as a basis for formal reasoning about properties of LC programs.

Plotkin (1981) develops a *structural approach to operational semantics* based on transition systems which successfully avoids many of these pitfalls. Its aims are summarised on Slide 9. It is this approach—coupled with related developments based on evaluation relations rather than transition relations (Kahn 1987; Milner, Tofte, and Harper 1990)—that we will illustrate in this course with respect to a number of small programming languages, of which LC is the simplest. The languages are chosen to be small and with 'idealised' syntax, in order to bring out more clearly the operational semantics of the various features, or combination of features they embody. For an example of the specification of a structural operational semantics for a full-scale language, see (Milner, Tofte, and Harper 1990).

2 Induction

Inductive definitions and proofs by induction are all-pervasive in the structural approach to operational semantics. The familiar (one hopes!) principle of Mathematical Induction and the equivalent Least Number Principle are recalled on Slide 10. Most of the induction techniques we will use can be justified by appealing to Mathematical Induction. Nevertheless, it is convenient to derive from it a number of induction principles more readily applicable to the structures with which we have to deal. This section briefly reviews some of the ideas and techniques; many examples of their use will occur throughout the rest of the course. Apart from the importance of these techniques for the subject, they should be important to you too, for **examination questions on this course assume an ability to give proofs using the various induction techniques**.

Mathematical Induction

For any property $\Phi(x)$ of natural numbers $x \in \mathbb{N} \stackrel{\text{def}}{=} \{0, 1, 2, \dots\}$, to prove

$$\forall x \in \mathbb{N}. \Phi(x)$$

it suffices to prove

$$\Phi(0) \quad \& \quad \forall x \in \mathbb{N}.\ \Phi(x) \Rightarrow \Phi(x+1).$$

Equivalently:

Least Number Principle: any non-empty subset of $\mathbb{N}$ possesses a least element.

Slide 10

2.1 A note on abstract syntax

When one gives a semantics for a programming language, one should only be concerned with the *abstract syntax* of the language, i.e. with the parse tree representation of phrases that results from lexical and syntax analysis of program texts. Accordingly, in this course when we look at various example languages we will *only* deal with abstract syntax trees.[1] Thus a

[1]In Section 5, when we consider binding constructs, we will be even more abstract and identify trees that only differ up to renaming of bound variables.

definition like that on Slide 5 is not really meant to specify LC phrases as strings of tokens, but rather as *finite labelled trees*. In this case the leaf nodes of the trees are labelled with elements from the set

$$\mathbb{Z} \cup \mathbb{B} \cup \mathbb{L} \cup \{\mathbf{skip}\}$$

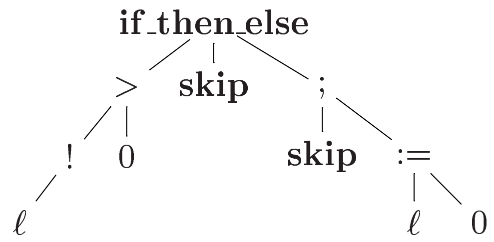
(using the notation introduced in Section 1.2), while the non-leaf nodes of the trees are labelled with elements of from the set

$$\mathbb{I}\,op \cup \mathbb{B}op \cup \{:=, ;, \mathbf{while\_do}, \mathbf{if\_then\_else}\}.$$

An example of such a tree is given on Slide 11, together with the informal textual representation which we will usually employ. The textual representation uses parentheses in order to indicate unambiguously which syntax tree is being referred to; and various infix and mixfix notations may be used for readability.

From this viewpoint of abstract syntax trees, the purpose of a grammar such as that on Slide 5 is to indicate which symbols are allowed as node labels, and the number and type of the children of each kind of node. Thus the grammar is analogous to the SML declaration of three mutually recursive datatypes given on Slide 12. Accordingly we will often refer to the labels at (non-leaf) nodes of syntax trees as *constructors* and the label at the root node of a tree as its *outermost constructor*.

Abstract syntax tree of an LC command



Textual representation:

$$\mathbf{if}\ !\ell > 0\ \mathbf{then\ skip\ else}\ (\mathbf{skip}\ ;\ \ell := 0)$$

Slide 11

An SML datatype of LC phrases

```
datatype iexp = Int of int | Loc of loc
              | Iop of iop*iexp*iexp
and      bexp = True | False
              | Bop of bop*iexp*iexp
and      cmd  = Skip | Asgn of loc*iexp
              | Seq of cmd*cmd
              | If of bexp*cmd*cmd
              | While of bexp*cmd
```

where `int`, `loc`, `iop`, and `bop` are suitable, predefined datatypes of numbers, locations, integer operations and boolean operations.

Slide 12

2.2 Structural induction

The principle of *Structural Induction* for some set of finite labelled trees says that to prove a property holds for all the trees it suffices to show that

base cases: the property holds for each type of leaf node (regarded as a one-element tree); and

induction step: for each tree constructor c (taking $n \geq 1$ arguments, say), if the property holds for any trees $t_1, \ldots, t_n$, then it also holds for the tree $c(t_1, \ldots, t_n)$.

For example, the principle for LC integer expressions is given on Slide 13. It should be clear how to formulate the principle for other collections of syntax trees, such as the set of all LC phrases.

Structural Induction for LC integer expressions

To prove that a property $\Phi(E)$ holds for all LC integer expressions E, it suffices to prove:

base cases: $\Phi(n)$ holds for all integers $n \in \mathbb{Z}$, and $\Phi(!\ell)$ holds for all locations $\ell \in \mathbb{L}$; and

induction step: for all integer expressions E, E' and operators $iop \in \mathbb{I}op$, if $\Phi(E)$ and $\Phi(E')$ hold, then so does $\Phi(E\ iop\ E')$.

Slide 13

Structural induction can be justified by an appeal to Mathematical Induction, relying upon the fact that the trees we are considering are *finite*, i.e. each tree has a finite number of nodes. For example, suppose we are trying to prove a property $\Phi(E)$ holds for all LC integer expressions E, given the statements labelled **base cases** and **induction step** on Slide 13. For each $n \in \mathbb{N}$, define

$$\Phi'(n) \quad \stackrel{\text{def}}{=} \quad \text{for all } E \text{ with at most } n \text{ nodes, } \Phi(E) \text{ holds.}$$

Since every E has only finitely many nodes, we have

$$\forall E.\Phi(E) \Leftrightarrow \forall n \in \mathbb{N}.\Phi'(n).$$

Then $\forall n \in \mathbb{N}.\Phi'(n)$ can be proved by Mathematical Induction using the **base cases** and **induction step** on Slide 13. Indeed $\Phi'(0)$ holds automatically (since there are no trees with 0 nodes); and if $\Phi'(n)$ holds and E has at most $n+1$ nodes, then

- either E is a leaf—so that $\Phi(E)$ holds by the **base cases** assumption,
- or it is of the form $E_1\ iop\ E_2$—in which case E_1 and E_2 have at most n nodes each, so by $P'(n)$ we have $\Phi(E_1)$ and $\Phi(E_2)$ and hence $\Phi(E)$ by the **induction step** assumption.

Thus $\Phi'(n+1)$ holds if $\Phi'(n)$ does, as required to complete the proof using Mathematical Induction.

Here is an example of the use of Structural Induction.

Example 2.2.1. Suppose E is an LC integer expression and s is a state whose domain of definition contains the locations $loc(E)$ occurring in E. (Recall that an LC state is a finite partial function from locations to integers.) Referring to the SMC-machine of Section 1.2, we claim that there is an integer n so that

$$\langle E \cdot c, r, s\rangle \to^* \langle c, n \cdot r, s\rangle$$

holds for any control stack c and intermediate results stack r. (In fact n is uniquely determined by E and s, because the SMC-machine is a deterministic transition system.)

Proof. We have to prove $\forall E.\Phi(E)$, where Φ is defined on Slide 14, and we do this by induction on the structure of E.

Base cases: $\Phi(n)$ holds by the *Constant* transition in Figure 1; and the *Location* transition implies that $\Phi(!\ell)$ holds with $n = s(\ell)$ (this is where we need the assumption $loc(E) \subseteq dom(s)$).

Induction step: Suppose $\Phi(E_1)$ and $\Phi(E_2)$ hold. Then we have

$$\begin{aligned}
\langle (E_1\ iop\ E_2) \cdot c, r, s\rangle &\to \langle E_1 \cdot E_2 \cdot iop \cdot c, r, s\rangle && \text{by (Compound) in Fig. 1}\\
&\to^* \langle E_2 \cdot iop \cdot c, n_1 \cdot r, s\rangle && \text{for some } n_1\text{, by } \Phi(E_1)\\
&\to^* \langle iop \cdot c, n_2 \cdot n_1 \cdot r, s\rangle && \text{for some } n_2\text{, by } \Phi(E_2)\\
&\to \langle c, n \cdot r, s\rangle && \text{by (Operator) in Fig. 1,}\\
& && \quad \text{where } n = n_1\ iop\ n_2.
\end{aligned}$$

(Note the way we chose the quantification in the definition of Φ: in the middle two $\to^*$ of the induction step we need to apply the 'induction hypothesis' for E_1 and E_2 with control and phrase stacks other than the ones c, r that we started with.) □

Termination of the SMC-machine on expressions

Define $\Phi(E)$ to be:

$$\forall s.loc(E) \subseteq dom(s) \Rightarrow$$
$$\exists n.\forall c, r.\langle E \cdot c, r, s\rangle \to^* \langle c, n \cdot r, s\rangle.$$

where $loc(E)$ denotes the finite set of locations occurring in E and $dom(s)$ denotes the domain of definition of s.

Then

$$\forall E.\Phi(E)\,.$$

Slide 14

2.3 Rule-based inductive definitions

As well as proving properties by induction, we will need to construct *inductively defined* subsets of some given set, T say. The method and terminology we use is adopted from mathematical logic, where the theorems of a particular formal system are built up inductively starting from some axioms, by repeatedly applying the rules of inference. In this case an *axiom*, a, just amounts to specifying an element $a \in T$ of the set. A *rule*, r, is a pair (H, c) where

- H is a finite, non-empty[1] subset of T (the elements of H are called the *hypotheses* of the rule r); and

- c is an element of T (called the *conclusion* of the rule r).

[1]A rule with an empty set of hypotheses plays the same rôle as an axiom.

Inductively defined subset of a set T

Given axioms A and rules R over T, a *proof* is a finite tree with nodes labelled by elements of T such that:

- each leaf-node is labelled with an axiom $a \in A$
- for any non-leaf node, if H is the set of labels of children of the node and c is the label of the node, then $(H, c) \in R$.

By definition, the subset of T *inductively defined* by the axioms and rules (A, R) consists of those $t \in T$ for which there is such a proof whose root node is labelled by t.

Slide 15

Slide 15 gives the definition of the subset of T inductively defined by such a collection of axioms and rules, in terms of the notion of a *proof*.[1] For example

$$\begin{array}{ccccccccc} & & t_0 & & & & & & \\ & / & & \backslash & & & & & \\ t_1 & & & & t_2 & & & & \\ & & & / & & \backslash & & & \\ & & t_3 & & & & t_4 & & \\ & / & & \backslash & & & & \backslash & \\ t_5 & & & & t_6 & & & & t_7 \end{array}$$

is a proof tree provided t_1, t_5, t_6, and t_7 are axioms and $(\{t_1, t_2\}, t_0)$, $(\{t_3, t_4\}, t_2)$, $(\{t_5, t_6\}, t_3)$, and $(\{t_7\}, t_4)$ are rules. In this context we write such trees in the following form

$$\cfrac{t_1 \qquad \cfrac{\cfrac{t_5 \quad t_6}{t_3} \quad \cfrac{t_7}{t_4}}{t_2}}{t_0}$$

The label of the root node of a proof tree is called the *conclusion* of the proof. If there is a proof whose conclusion is t, we say that *t has a proof from the axioms and rules (A, R)*, or

[1] If one allows rules with infinitely many hypotheses, one must consider proofs that are not necessarily finite trees, but are *well-founded* trees—meaning that any path from a node towards the tree's leaves must be finite.

that it has a *derivation*, or just that it *is valid*. The collection of all such t is by definition *the subset of T inductively defined by the axioms and rules* (A, R).

Example 2.3.1 (Evaluation relation for LC expressions). Example 2.2.1 shows that the SMC-machine evaluation of LC integer expressions depends only upon the expression to be evaluated and the current state. Here is an inductively defined relation that captures this evaluation directly (i.e. without the need for control and phrase stacks). We will extend this to all LC phrases in the next section.

The evaluation relation, $\Downarrow$, is a subset of the set of all triples (E, s, n), where E is an LC integer expression, s is a state, and n is an integer. It is inductively defined by the axioms and rules on Slide 16, where we use an infix notation $E, s \Downarrow n$ instead of writing $(E, s, n) \in \Downarrow$.

Here for example, is a proof that $(!\ell * 2) - 3, \{\ell \mapsto 4\} \Downarrow 5$ is a valid instance of the evaluation relation:

$$\frac{\dfrac{!\ell, \{\ell \mapsto 4\} \Downarrow 4 \quad 2, \{\ell \mapsto 4\} \Downarrow 2}{!\ell * 2, \{\ell \mapsto 4\} \Downarrow 8} \qquad 3, \{\ell \mapsto 4\} \Downarrow 3}{(!\ell * 2) - 3, \{\ell \mapsto 4\} \Downarrow 5}\ .$$

An evaluation relation for LC expressions

can be inductively defined by the axioms

$$n, s \Downarrow n$$
$$!\ell, s \Downarrow n \quad \text{if } \ell \in dom(s) \ \&\ s(\ell) = n$$

and the rules

$$\frac{E_1, s \Downarrow n_1 \quad E_2, s \Downarrow n_2}{E_1 \ iop\ E_2, s \Downarrow n} \ \text{if } n = n_1 \ iop\ n_2$$

where E_1, E_2 are LC integer expressions, s is a state, ℓ is a location, and n_1, n_2, n are integers.

Slide 16

This is our first (albeit very simple) example of a *structural* operational semantics. Structural, because the axioms and rules for proving instances

(1) $$E, s \Downarrow n$$

of the inductively defined evaluation relation follow the structure of the expression E. For if (1) has a proof, we can reconstruct it from the bottom up guided by the structure of E: if E is an integer or a location the proof must just be an axiom, whereas if E is compound the proof must end with an application of the corresponding rule.

Note. The axioms and rules appearing on Slide 16, and throughout the course, are meant to be 'schematic'—in the sense that, for example, there is one axiom of the form $n, s \Downarrow n$ for each possible choice of an integer n and a state s. The statements beginning 'if ...' which qualify the second axiom and the rule are often called *side-conditions*. They restrict how the schematic presentation of an axiom or rule may be instantiated to get an actual axiom or rule. For example, $!\ell, s \Downarrow n$ is only an axiom of this particular system for some particular choice of location ℓ, state s, and integer n, provided ℓ is in the domain of definition of s and the value of s at ℓ is n.

Rule Induction

Given axioms A and rules R over a set T, let I be the subset of T inductively defined by (A, R) (cf. Slide 15). Given a property $\Phi(t)$ of elements $t \in T$, to prove

$$\forall t \in I.\Phi(t)$$

it suffices to show

closure under axioms: $\Phi(a)$ holds for each $a \in A$; and

closure under rules: for each rule $\left(\frac{h_1 \cdots h_n}{c}\right) \in R$

$$\Phi(h_1) \ \& \ \ldots \ \& \ \Phi(h_n) \ \Rightarrow \ \Phi(c).$$

Slide 17

2.4 Rule induction

Suppose that (A, R) are some axioms and rules on a set T and that $I \subseteq T$ is the subset inductively defined by them. The principle of *Rule Induction* for I is given on Slide 17. It can be justified by an appeal to Mathematical Induction in much the same way that we justified Structural Induction in Section 2.2: the closure of I under the axioms and rules allows one to prove

$\forall n \in \mathbb{N}. \forall t \in I.$ if t is the conclusion of a proof with at most n nodes, then

$\Phi(t)$ holds

by induction on n. And since any proof is a finite[1] tree, this shows that $\forall t \in I.\Phi(t)$ holds.

Example 2.4.1. We show by Rule Induction that $E, s \Downarrow n$ implies that in the SMC-machine $\langle E \cdot c, p, s\rangle \rightarrow^* \langle c, n \cdot p, s\rangle$ holds for any control stack c and phrase stack p.

Proof. So let $\Phi(E, s, n)$ be the property

$$\forall c, p.\langle E \cdot c, p, s\rangle \rightarrow^* \langle c, n \cdot p, s\rangle.$$

According to Slide 17 we have to show that $\Phi(E, s, n)$ is closed under the axioms and rules on Slide 16.

Closure under axioms: $\Phi(n, s, n)$ holds by the *Constant* transition in Figure 1; and if $\ell \in dom(s)$ and $s(\ell) = n$, then the *Location* transition implies that $\Phi(\ell, s, n)$ holds.

Closure under rules: We have to show

$$\Phi(E_1, s, n_1) \ \& \ \Phi(E_2, s, n_2) \Rightarrow \Phi(E_1 \ iop \ E_2, s, n_1 \ iop \ n_2)$$

and this follows just as in the **Induction step** of the proof in Example 2.2.1.

$\square$

2.5 Exercises

Exercise 2.5.1. Give an example of an SMC-machine configuration from which there is an infinite sequence of transitions.

Exercise 2.5.2. Consider the subset D of the set $\mathbb{N}\times\mathbb{N}$ of pairs of natural numbers inductively defined by the following axioms and rules

$$(n, 0) \in D$$

$$\frac{(n, n') \in D}{(n, n + n') \in D}.$$

Use Rule Induction to prove

$$(n, n') \in D \ \Rightarrow \ \exists k \in \mathbb{N}.\ n' = k * n$$

(where $*$ denotes multiplication). Use Mathematical Induction on k to show conversely that if $n' = k * n$ then $(n, n') \in D$.

Exercise 2.5.3. Let $(\mathit{Config}, \rightarrow)$ be a transition system (cf. Slide 4). Give an inductive definition of the subset of $\mathit{Config} \times \mathit{Config}$ consisting of the reflexive-transitive closure of $\rightarrow$. Use Mathematical Induction and Rule Induction to prove that your definition gives the same relation as Definition 1.1.1(i).

[1] This relies upon the fact that we are only considering rules with finitely many hypotheses. Without this assumption, Rule Induction is still valid, but is not necessarily reducible to Mathematical Induction.

3 Structural Operational Semantics

In this section we will give structural operational semantics for the LC language introduced in Section 1.2. We do this first in terms of an inductively defined transition relation and then in terms of an inductively defined relation of evaluation. The induction principles of the previous section are used to relate the two approaches.

3.1 Transition semantics of LC

Recall the definition of LC *phrases*, P, on Slide 5. Recall also that a *state*, s, is by definition a finite partial function from locations to integers; we include the possibility that the set of locations at which s is defined, $dom(s)$, is empty—we simply write $\emptyset$ for this s.

We define a transition system (cf. Section 1.1) for LC whose configurations are pairs $\langle P, s\rangle$ consisting of an LC phrase P and a state s. The transition relation is inductively defined by the axioms and rules on Slides 18, 19, and 20. In rule ($\overrightarrow{\text{set2}}$), $s[\ell \mapsto n]$ denotes the state that maps ℓ to n and otherwise acts like s. Thus $dom(s[\ell \mapsto n]) = dom(s) \cup \{\ell\}$ and

$$s[\ell \mapsto n](\ell') = \begin{cases} n & \text{if } \ell' = \ell, \\ s(\ell') & \text{if } \ell' \neq \ell \;\&\; \ell' \in dom(s). \end{cases}$$

Note that the axioms and rules for $\langle P, s\rangle \to \langle P', s'\rangle$ follow the syntactic structure of the phrase P. There are no axioms or rules for transitions from $\langle P, s\rangle$ in case P is an integer, a boolean, $\mathbf{skip}$, or in case $P = !\ell$ where ℓ is a location not in the domain of definition of s. The first three of these alternatives are defined to be the LC *terminal configurations*; the fourth alternative is a basic example of a *stuck* configuration.[2] This is summarised on Slide 21.

[2] One can rule out stuck LC configurations by restricting the set of configurations to consist of all pairs $\langle P, s\rangle$ satisfying that the domain of definition of the state s contains all the locations that occur in the phrase P; see Exercise 3.4.4.

LC transition relation — expressions

$(\overrightarrow{\mathrm{loc}})$ $\quad \langle !\ell, s\rangle \to \langle n, s\rangle$ if $\ell \in dom(s)$ & $s(\ell) = n$

$$(\overrightarrow{\mathrm{op1}}) \quad \frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 \ op\ E_2, s\rangle \to \langle E_1' \ op\ E_2, s'\rangle}$$

$$(\overrightarrow{\mathrm{op2}}) \quad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle n_1 \ op\ E_2, s\rangle \to \langle n_1 \ op\ E_2', s'\rangle}$$

$(\overrightarrow{\mathrm{op3}})$ $\quad \langle n_1 \ op\ n_2, s\rangle \to \langle c, s\rangle$ if $c = n_1 \ op\ n_2$

Slide 18

LC transition relation — := and ;

$$(\overrightarrow{\mathrm{set1}}) \quad \frac{\langle E, s\rangle \to \langle E', s'\rangle}{\langle \ell := E, s\rangle \to \langle \ell := E', s'\rangle}$$

$$(\overrightarrow{\mathrm{set2}}) \quad \langle \ell := n, s\rangle \to \langle \mathbf{skip}, s[\ell \mapsto n]\rangle$$

$$(\overrightarrow{\mathrm{seq1}}) \quad \frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1 \,;\, C_2, s\rangle \to \langle C_1' \,;\, C_2, s'\rangle}$$

$$(\overrightarrow{\mathrm{seq2}}) \quad \langle \mathbf{skip} \,;\, C, s\rangle \to \langle C, s\rangle$$

Slide 19

LC transition relation — conditional & while

$$(\overrightarrow{\text{if1}}) \qquad \frac{\langle B, s\rangle \to \langle B', s'\rangle}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \to \langle \textbf{if } B' \textbf{ then } C_1 \textbf{ else } C_2, s'\rangle}$$

$$(\overrightarrow{\text{if2}}) \qquad \langle \textbf{if true then } C_1 \textbf{ else } C_2, s\rangle \to \langle C_1, s\rangle$$

$$(\overrightarrow{\text{if3}}) \qquad \langle \textbf{if false then } C_1 \textbf{ else } C_2, s\rangle \to \langle C_2, s\rangle$$

$$(\overrightarrow{\text{whl}}) \qquad \langle \textbf{while } B \textbf{ do } C, s\rangle \to \langle \textbf{if } B \textbf{ then } (C \,; \textbf{while } B \textbf{ do } C) \textbf{ else skip}, s\rangle$$

Slide 20

Terminal and stuck LC configurations

The *terminal* configurations are by definition

$$\langle n, s\rangle \quad \langle \textbf{true}, s\rangle \quad \langle \textbf{false}, s\rangle \quad \langle \textbf{skip}, s\rangle.$$

A configuration $\langle P, s\rangle$ is *stuck* if and only if it is not terminal, but $\langle P, s\rangle \not\to$.

(For example, $\langle (!\ell + 1), \{\ell' \mapsto 1\}\rangle$ is stuck if $\ell' \neq \ell$.)

Slide 21

An example of a sequence of valid transitions is given on Slide 22. Compared with the corresponding run of the SMC-machine given on Slide 7, each transition is doing some real computation rather than just symbol juggling. On the other hand, the validity of each transition on Slide 7 is immediate from the definition of the SMC-machine, whereas each transition on Slide 22 has to be justified with a proof from the axioms and rules in Slides 18–20. For example, the proof of the second transition on Slide 22 is:

$$\dfrac{\dfrac{\dfrac{}{\langle !\ell, s\rangle \to \langle 4, s\rangle}\;(\overrightarrow{\mathrm{loc}})}{\langle !\ell > 0, s\rangle \to \langle 4 > 0, s\rangle}\;(\overrightarrow{\mathrm{op1}})}{\begin{array}{l}\langle \mathbf{if}\ !\ell > 0\ \mathbf{then}\ (C' \,;C)\ \mathbf{else\ skip}, s\rangle \to \\ \qquad \langle \mathbf{if}\ 4 > 0\ \mathbf{then}\ (C' \,;C)\ \mathbf{else\ skip}, s\rangle\end{array}}\;(\overrightarrow{\mathrm{if1}}).$$

Luckily the structural nature of the axioms and rules makes it quite easy to check whether a particular transition $\langle P, s\rangle \to \langle P', s'\rangle$ is valid or not: one tries to construct a proof from the bottom up, and at each stage the syntactic structure of P dictates which axiom or rule must have been used to conclude that transition.

$$\begin{aligned}\langle C, s\rangle &\to \langle \mathbf{if}\ B\ \mathbf{then}\ (C' \,;C)\ \mathbf{else\ skip}, s\rangle\\ &\to \langle \mathbf{if}\ 4 > 0\ \mathbf{then}\ (C' \,;C)\ \mathbf{else\ skip}, s\rangle\\ &\to \langle \mathbf{if\ true\ then}\ (C' \,;C)\ \mathbf{else\ skip}, s\rangle\\ &\to \langle C' \,;C, s\rangle\\ &\to^* \langle \mathbf{skip}, s[\ell \mapsto 0, \ell' \mapsto 24]\rangle\end{aligned}$$

$$\text{where}\begin{cases} C & \stackrel{\text{def}}{=} \mathbf{while}\ B\ \mathbf{do}\ C',\\ B & \stackrel{\text{def}}{=} !\ell > 0,\\ C' & \stackrel{\text{def}}{=} \ell' := !\ell * !\ell' \,;\ell := !\ell - 1,\\ s & \stackrel{\text{def}}{=} \{\ell \mapsto 4, \ell' \mapsto 1\}.\end{cases}$$

Slide 22

Some properties of LC transitions

Determinacy. If $\langle P, s\rangle \rightarrow \langle P', s'\rangle$ and $\langle P, s\rangle \rightarrow \langle P'', s''\rangle$, then $P' = P''$ and $s' = s''$.

Subject reduction. If $\langle P, s\rangle \rightarrow \langle P', s'\rangle$, then P' is of the same type (command/integer expression/boolean expression) as P.

Expressions are side-effect free. If $\langle P, s\rangle \rightarrow \langle P', s'\rangle$ and P is an integer or boolean expression, then $s = s'$.

Slide 23

Some properties of the LC transition relation are stated on Slide 23. They can all be proved by Rule Induction (see Slide 17). For example, to prove the transition system is deterministic define the property $\Phi(P, s, P', s')$ to be

$$\langle P, s\rangle \rightarrow \langle P', s'\rangle \quad \& \quad \forall P'', s''.\langle P, s\rangle \rightarrow \langle P'', s''\rangle \Rightarrow (P' = P'' \;\&\; s' = s'').$$

We wish to prove that every valid transition $\langle P, s\rangle \rightarrow \langle P', s'\rangle$ satisfies $\Phi(P, s, P', s')$, and by the principle of Rule Induction it suffices to check that $\Phi(P, s, P', s')$ is closed under the axioms and rules that inductively define $\rightarrow$. We give the argument for closure under rule $(\overrightarrow{\text{set1}})$ and leave the other cases as exercises.

Proof of closure under rule $(\overrightarrow{\text{set1}})$. Suppose $\Phi(E, s, E', s')$ holds. We have to prove that $\Phi(\ell := E, s, \ell := E', s')$ holds, i.e. that $\langle \ell := E, s\rangle \rightarrow \langle \ell := E', s'\rangle$ (which follows from $\Phi(E, s, E', s')$ by $(\overrightarrow{\text{set1}})$), and that if

$$\langle \ell := E, s\rangle \rightarrow \langle P'', s''\rangle \tag{2}$$

then $P'' = (\ell := E')$ and $s'' = s'$.

Now the last step in the proof of (2) can only be by $(\overrightarrow{\text{set1}})$ or $(\overrightarrow{\text{set2}})$ (because of the structure of $\ell := E$). But in fact it cannot be by $(\overrightarrow{\text{set2}})$ since then E would have to be some integer n; so $\langle E, s\rangle \not\rightarrow$ (cf. Slide 21), which contradicts $\Phi(E, s, E', s')$.* Therefore the

*See Remark 3.1.1.

last step of the proof of (2) uses ($\overrightarrow{\text{set1}}$) and hence

(3) $$P'' = \ell := E''$$

for some E'' satisfying

(4) $$\langle E, s\rangle \to \langle E'', s''\rangle.$$

Then by definition of $\Phi(E, s, E', s')$, (4) implies that $E' = E''$ and $s' = s''$, and hence also by (3) that $P'' = \ell := E'' = \ell := E'$. Thus we do indeed have $\Phi(\ell := E, s, \ell := E', s')$, as required. $\square$

Remark 3.1.1. Note that some care is needed in choosing the property when applying Rule Induction. For example, if we had defined $\Phi(P, s, P', s')$ to just be

$$\forall P'', s''.\langle P, s\rangle \to \langle P'', s''\rangle \Rightarrow (P' = P'' \ \&\ s' = s'')$$

what would go wrong with the above proof of closure under rule ($\overrightarrow{\text{set1}}$)? [Hint: look at the point in the proof marked with a $*$.]

3.2 Evaluation semantics of LC

Given an LC phrase P and a state s, since the LC transition system is deterministic, there is a unique sequence of transitions starting from $\langle P, s\rangle$ and of maximal length:

$$\langle P, s\rangle \to \langle P_1, s_1\rangle \to \langle P_2, s_2\rangle \to \langle P_3, s_3\rangle \to \dots.$$

We call this the *evaluation sequence* for $\langle P, s\rangle$. In general, for deterministic languages there are three possible types of evaluation sequence, shown on Slide 24. For LC, the stuck evaluation sequences can be avoided by restricting attention to 'sensible' configurations satisfying $loc(P) \subseteq dom(s)$: see Exercise 3.4.4. LC certainly possesses divergent evaluation sequences—the simplest possible example is given on Slide 25. In this section we give a direct inductive definition of the terminating evaluation sequences, i.e. of the relation

$$\langle P, s\rangle \to^* \langle V, s'\rangle \qquad \langle V, s'\rangle \text{ terminal.}$$

Types of evaluation sequence

$$\langle P, s\rangle \to \langle P_1, s_1\rangle \to \langle P_2, s_2\rangle \to \ldots$$

***Terminating*:** the sequence eventually reaches a terminal configuration (cf. Slide 21).

***Stuck*:** the sequence eventually reaches a stuck configuration.

***Divergent*:** the sequence is infinite.

Slide 24

A divergent command

For $C \stackrel{\text{def}}{=} \mathbf{while\ true\ do\ skip}$ we have

$$\begin{array}{ll} & \langle C, s\rangle \\ \to & \langle \mathbf{if\ true\ then}\ (\mathbf{skip}\ ; C)\ \mathbf{else\ skip}, s\rangle \\ \to & \langle \mathbf{skip}\ ; C, s\rangle \\ \to & \langle C, s\rangle \\ \to & \ldots \end{array}$$

Slide 25

The LC *evaluation relation*, will be given as an inductively defined subset of $(Phrases \times States) \times (Phrases \times States)$, written with infix notation

(5) $$\langle P, s\rangle \Downarrow \langle V, s'\rangle.$$

The axioms and rules inductively defining (5) are given in Figure 2 and on Slide 26. Note that if (5) is derivable, then $\langle V, s'\rangle$ is a terminal configuration (this is easily proved by Rule Induction).

Evaluation rules for while

$$(\Downarrow_{\mathrm{wh1}}) \quad \frac{\langle B, s\rangle \Downarrow \langle \mathbf{true}, s'\rangle \quad \langle C, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle \quad \langle \mathbf{while}\ B\ \mathbf{do}\ C, s''\rangle \Downarrow \langle \mathbf{skip}, s'''\rangle}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \Downarrow \langle \mathbf{skip}, s'''\rangle}$$

$$(\Downarrow_{\mathrm{wh2}}) \quad \frac{\langle B, s\rangle \Downarrow \langle \mathbf{false}, s'\rangle}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle}$$

Slide 26

As for the transition relation, the axioms and rules defining (5) follow the structure of P and this helps us to construct proofs of evaluation from the bottom up. Given a configuration $\langle P, s\rangle$, since $\Downarrow$ collapses whole sequences of computation steps into one relation (this is made precise below by the Theorem on Slide 27) it may be difficult to decide for which terminal configuration $\langle V, s'\rangle$ we should try to reconstruct a proof of (5). It is sometimes possible to deduce this information at the same time as building up the proof tree from the bottom—an example of this process is illustrated in Figure 3. However, the fact (which we will not pursue here) that LC is capable of coding any partial recursive function means that there is no algorithm which, given a configuration $\langle P, s\rangle$, decides whether or not there exists $\langle V, s'\rangle$ for which (5) holds.

We have seen how to exploit the structural nature of the LC evaluation relation to construct proofs of evaluation. The following example illustrates how to prove that a configuration *does not* evaluate to anything.

$(\Downarrow_{\mathrm{con}})$ $\langle c, s\rangle \Downarrow \langle c, s\rangle$ ($c \in \mathbb{Z} \cup \mathbb{B}$)

$(\Downarrow_{\mathrm{loc}})$ $\langle !\ell, s\rangle \Downarrow \langle n, s\rangle$ if $\ell \in dom(s)$ & $s(\ell) = n$

$(\Downarrow_{\mathrm{op}})$ $\dfrac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1 \ op \ E_2, s\rangle \Downarrow \langle c, s''\rangle}$ where c is the value of $n_1 \ op \ n_2$ (for op an integer or boolean operation)

$(\Downarrow_{\mathrm{skip}})$ $\langle \mathbf{skip}, s\rangle \Downarrow \langle \mathbf{skip}, s\rangle$

$(\Downarrow_{\mathrm{set}})$ $\dfrac{\langle E, s\rangle \Downarrow \langle n, s'\rangle}{\langle \ell := E, s\rangle \Downarrow \langle \mathbf{skip}, s'[\ell \mapsto n]\rangle}$

$(\Downarrow_{\mathrm{seq}})$ $\dfrac{\langle C_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}{\langle C_1 \,;\, C_2, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}$

$(\Downarrow_{\mathrm{if1}})$ $\dfrac{\langle B, s\rangle \Downarrow \langle \mathbf{true}, s'\rangle \quad \langle C_1, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}{\langle \mathbf{if} \ B \ \mathbf{then} \ C_1 \ \mathbf{else} \ C_2, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}$

$(\Downarrow_{\mathrm{if2}})$ $\dfrac{\langle B, s\rangle \Downarrow \langle \mathbf{false}, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}{\langle \mathbf{if} \ B \ \mathbf{then} \ C_1 \ \mathbf{else} \ C_2, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}$

plus rules $(\Downarrow_{\mathrm{wh1}})$ and $(\Downarrow_{\mathrm{wh2}})$ on Slide 26.

Figure 2: Axioms and rules for LC evaluation

For $\begin{cases} C & \stackrel{\text{def}}{=} \mathbf{while}\ !\ell > 0\ \mathbf{do}\ \ell := 0 \\ s & \stackrel{\text{def}}{=} \{\ell \mapsto 1\} \end{cases}$

we try to find s'' such that $\langle C, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle$ is provable. Since $\langle !\ell > 0, s\rangle \Downarrow \langle \mathbf{true}, s\rangle$ (proof shown below), the last rule used in the proof must be ($\Downarrow_{\mathrm{wh1}}$):

$$\dfrac{\dfrac{\dfrac{}{\langle !\ell, s\rangle \Downarrow \langle 1, s\rangle}\,(\Downarrow_{\mathrm{loc}}) \quad \dfrac{}{\langle 0, s\rangle \Downarrow \langle 0, s\rangle}\,(\Downarrow_{\mathrm{con}})}{\langle !\ell > 0, s\rangle \Downarrow \langle \mathbf{true}, s\rangle}\,(\Downarrow_{\mathrm{op}}) \quad \begin{array}{c} ? \\ \vdots \\ \langle \ell := 0, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle \end{array} \quad \begin{array}{c} ? \\ \vdots \\ \langle C, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle \end{array}}{\langle C, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}\,(\Downarrow_{\mathrm{wh1}})$$

for some s' and s''. The middle hypothesis of ($\Downarrow_{\mathrm{wh1}}$) must have been deduced using ($\Downarrow_{\mathrm{set}}$). So $s' = \{\ell \mapsto 0\}$ and we have:

$$\dfrac{\dfrac{\dfrac{}{\langle !\ell, s\rangle \Downarrow \langle 1, s\rangle}\,(\Downarrow_{\mathrm{loc}}) \quad \dfrac{}{\langle 0, s\rangle \Downarrow \langle 0, s\rangle}\,(\Downarrow_{\mathrm{con}})}{\langle !\ell > 0, s\rangle \Downarrow \langle \mathbf{true}, s\rangle}\,(\Downarrow_{\mathrm{op}}) \quad \dfrac{\dfrac{}{\langle 0, s\rangle \Downarrow \langle 0, s\rangle}\,(\Downarrow_{\mathrm{con}})}{\langle \ell := 0, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle}\,(\Downarrow_{\mathrm{set}}) \quad \begin{array}{c} ? \\ \vdots \\ \langle C, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle \end{array}}{\langle C, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}\,(\Downarrow_{\mathrm{wh1}}).$$

Finally, since $\langle !\ell > 0, s'\rangle \Downarrow \langle \mathbf{false}, s'\rangle$ (proof shown below), the last rule used in the proof of the right-hand branch must be ($\Downarrow_{\mathrm{wh2}}$). So $s'' = s' = \{\ell \mapsto 0\}$ and the complete proof is:

$$\dfrac{\dfrac{\dfrac{}{\langle !\ell, s\rangle \Downarrow \langle 1, s\rangle}\,(\Downarrow_{\mathrm{loc}}) \quad \dfrac{}{\langle 0, s\rangle \Downarrow \langle 0, s\rangle}\,(\Downarrow_{\mathrm{con}})}{\langle !\ell > 0, s\rangle \Downarrow \langle \mathbf{true}, s\rangle}\,(\Downarrow_{\mathrm{op}}) \quad \dfrac{\dfrac{}{\langle 0, s\rangle \Downarrow \langle 0, s\rangle}\,(\Downarrow_{\mathrm{con}})}{\langle \ell := 0, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle}\,(\Downarrow_{\mathrm{set}}) \quad \dfrac{\dfrac{\dfrac{}{\langle !\ell, s'\rangle \Downarrow \langle 0, s'\rangle}\,(\Downarrow_{\mathrm{loc}}) \quad \dfrac{}{\langle 0, s'\rangle \Downarrow \langle 0, s'\rangle}\,(\Downarrow_{\mathrm{con}})}{\langle !\ell > 0, s'\rangle \Downarrow \langle \mathbf{false}, s'\rangle}\,(\Downarrow_{\mathrm{op}})}{\langle C, s'\rangle \Downarrow \langle \mathbf{skip}, s'\rangle}\,(\Downarrow_{\mathrm{wh2}})}{\langle C, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle}\,(\Downarrow_{\mathrm{wh1}}).$$

Figure 3: Reconstructing a proof of evaluation

Example 3.2.1. Consider $C \stackrel{\text{def}}{=} \mathbf{while}\ \mathbf{true}\ \mathbf{do}\ \mathbf{skip}$ and any state s. We claim that there is no s' such that

$$\langle C, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle \tag{6}$$

is valid. We argue by contradiction. Suppose (6) has a proof. Then by the Least Number Principle (see Slide 10), amongst all the proof trees with (6) as their conclusion, there is one with a minimal number of nodes—call it $\mathcal{P}$. Because of the structure of C, the last part of $\mathcal{P}$ can only be

$$\dfrac{\dfrac{}{\langle \mathbf{true}, s\rangle \Downarrow \langle \mathbf{true}, s\rangle}\,(\Downarrow_{\text{con}}) \quad \dfrac{}{\langle \mathbf{skip}, s\rangle \Downarrow \langle \mathbf{skip}, s\rangle}\,(\Downarrow_{\text{skip}}) \quad \begin{array}{c}\mathcal{P}'\\ \langle C, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle\end{array}}{\langle C, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle}\,(\Downarrow_{\text{wh1}})$$

where $\mathcal{P}'$ is also a proof of (6). But $\mathcal{P}'$ is a proper subtree of $\mathcal{P}$ and so has strictly fewer nodes than it—contradicting the minimality property of $\mathcal{P}$. So there cannot exist any s' for which $\langle C, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$ holds. □

Equivalence of
LC transition and evaluation semantics

Theorem. *For all configurations $\langle P, s\rangle$ and all terminal configurations $\langle V, s'\rangle$*

$$\langle P, s\rangle \Downarrow \langle V, s'\rangle \quad \Leftrightarrow \quad \langle P, s\rangle \to^* \langle V, s'\rangle.$$

Three part proof:

(a) $\langle P, s\rangle \Downarrow \langle V, s'\rangle \Rightarrow \langle P, s\rangle \to^* \langle V, s'\rangle$

(b) $\langle P, s\rangle \to \langle P', s'\rangle \;\&\; \langle P', s'\rangle \Downarrow \langle V, s''\rangle \Rightarrow \langle P, s\rangle \Downarrow \langle V, s''\rangle$

(c) $\langle P, s\rangle \to^* \langle V, s'\rangle \Rightarrow \langle P, s\rangle \Downarrow \langle V, s'\rangle$

Slide 27

3.3 Equivalence of LC transition and evaluation semantics

The close relationship between the LC evaluation and transition relations is stated in the Theorem on Slide 27. (Recall from 1.1.1(i) that $\to^*$ denotes the reflexive-transitive closure of $\to$.) As indicated on the slide, we break the proof of the Theorem into three parts.

Proof of (a) on Slide 27. Let $\Phi(P, s, V, s')$ be the property

$$\langle P, s\rangle \to^* \langle V, s'\rangle \;\&\; \langle V, s'\rangle \text{ is terminal.}$$

By Rule Induction, to prove (a) it suffices to show that $\Phi(P, s, V, s')$ is closed under the axioms and rules inductively defining $\Downarrow$. We give the proof of closure under rule $(\Downarrow_{\mathrm{wh1}})$ and leave the other cases as exercises.

So suppose

$$\begin{array}{ll} (7) & \langle B, s\rangle \to^* \langle \mathbf{true}, s'\rangle \\ (8) & \langle C, s'\rangle \to^* \langle \mathbf{skip}, s''\rangle \\ (9) & \langle \mathbf{while}\ B\ \mathbf{do}\ C, s''\rangle \to^* \langle \mathbf{skip}, s'''\rangle. \end{array}$$

We have to show that

$$\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \to^* \langle \mathbf{skip}, s'''\rangle.$$

Writing C' for $\mathbf{while}\ B\ \mathbf{do}\ C$, using the axioms and rules inductively defining $\to$ we have:

$$\begin{array}{lll} & \langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle & \\ \to & \langle \mathbf{if}\ B\ \mathbf{then}\ (C\,; C')\ \mathbf{else\ skip}, s\rangle & \text{by } (\overrightarrow{\mathrm{whl}}) \\ \to^* & \langle \mathbf{if\ true\ then}\ (C\,; C')\ \mathbf{else\ skip}, s'\rangle & \text{by } (\overrightarrow{\mathrm{if1}})^* \text{ on (7)} \\ \to & \langle C\,; C', s'\rangle & \text{by } (\overrightarrow{\mathrm{if2}}) \\ \to^* & \langle \mathbf{skip}\,; C', s''\rangle & \text{by } (\overrightarrow{\mathrm{seq1}})^* \text{ on (8)} \\ \to & \langle C', s''\rangle & \text{by } (\overrightarrow{\mathrm{seq2}}) \\ \to^* & \langle \mathbf{skip}, s'''\rangle & \text{by (9)} \end{array}$$

as required. □

Proof of (b) on Slide 27. Let $\Psi(P, s, P', s')$ be the property

$$\forall \langle V, s''\rangle.\ \langle P', s'\rangle \Downarrow \langle V, s''\rangle \Rightarrow \langle P, s\rangle \Downarrow \langle V, s''\rangle.$$

By Rule Induction, to prove (b) it suffices to show that $\Psi(P, s, P', s')$ is closed under the axioms and rules inductively defining $\to$. We give the proof of closure under rule $(\overrightarrow{\mathrm{whl}})$ and leave the other cases as exercises.

So writing C' for $\mathbf{while}\ B\ \mathbf{do}\ C$ we have to prove $\Psi(C', s, \mathbf{if}\ B\ \mathbf{then}\ (C\,; C')\ \mathbf{else\ skip}, s)$ holds for any s, i.e. that for all terminal $\langle V, s''\rangle$

$$(10) \qquad \langle \mathbf{if}\ B\ \mathbf{then}\ (C\,; C')\ \mathbf{else\ skip}, s\rangle \Downarrow \langle V, s''\rangle$$

implies

$$(11) \qquad \langle C', s\rangle \Downarrow \langle V, s''\rangle.$$

But if (10) holds it can only have been deduced by a proof ending with either $(\Downarrow_{\mathrm{if1}})$ or $(\Downarrow_{\mathrm{if2}})$. So there are two cases to consider:

Case (10) was deduced by ($\Downarrow_{\mathrm{if1}}$) from

$$(12) \qquad \langle B, s\rangle \Downarrow \langle \mathbf{true}, s'\rangle$$

for some state s' such that

$$\langle C\,; C', s'\rangle \Downarrow \langle V, s''\rangle$$

which in turn must have been deduced by ($\Downarrow_{\mathrm{seq}}$) from

$$(13) \qquad \langle C, s'\rangle \Downarrow \langle \mathbf{skip}, s'''\rangle$$

$$(14) \qquad \langle C', s'''\rangle \Downarrow \langle \mathbf{skip}, s''\rangle$$

for some state s'''. Hence $V = \mathbf{skip}$ and applying ($\Downarrow_{\mathrm{wh1}}$) to (12), (13), and (14) yields (11), as required.

Case (10) was deduced by ($\Downarrow_{\mathrm{if2}}$) from

$$(15) \qquad \langle B, s\rangle \Downarrow \langle \mathbf{false}, s'\rangle$$

$$(16) \qquad \langle \mathbf{skip}, s'\rangle \Downarrow \langle V, s''\rangle$$

for some state s'. Now (16) can only have been deduced using ($\Downarrow_{\mathrm{skip}}$), so $V = \mathbf{skip}$ and $s'' = s'$. Then ($\Downarrow_{\mathrm{wh2}}$) applied to (15) yields (11), as required. $\square$

Proof of (c) on Slide 27. Applying property (b) repeatedly, for any finite chain of transitions $\langle P, s\rangle \to \ldots \to \langle P', s'\rangle$ we have that $\langle P', s'\rangle \Downarrow \langle V, s''\rangle$ implies $\langle P, s\rangle \Downarrow \langle V, s''\rangle$. Now since $\langle V, s''\rangle$ is terminal it is the case that $\langle V, s''\rangle \Downarrow \langle V, s''\rangle$. Therefore taking $\langle P', s'\rangle = \langle V, s''\rangle$ we obtain property (c). $\square$

In view of this equivalence theorem, we can deduce the properties of $\Downarrow$ given on Slide 28 from the corresponding properties of $\to$ given on Slide 23. Alternatively these properties can be proved directly using Rule Induction for $\Downarrow$.

We conclude this section by stating, without proof, the relationship between LC evaluation and runs of the SMC-machine (cf. Section 1.2).

Theorem 3.3.1. *For all configurations $\langle P, s\rangle$ and all terminal configurations $\langle V, s'\rangle$, $\langle P, s\rangle \Downarrow \langle V, s'\rangle$ holds if and only if*

either *P is an integer or boolean expression and there is a run of the SMC-machine of the form $\langle P \cdot \mathbf{nil}, \mathbf{nil}, s\rangle \to^* \langle \mathbf{nil}, V \cdot \mathbf{nil}, s'\rangle$;*

or *P is a command, $V = \mathbf{skip}$ and there is a run of the SMC-machine of the form $\langle P \cdot \mathbf{nil}, \mathbf{nil}, s\rangle \to^* \langle \mathbf{nil}, \mathbf{nil}, s'\rangle$.*

Some properties of LC evaluation

Determinacy. If $\langle P, s\rangle \Downarrow \langle V, s'\rangle$ and $\langle P, s\rangle \Downarrow \langle V', s''\rangle$, then $V = V'$ and $s' = s''$.

Subject reduction. If $\langle P, s\rangle \Downarrow \langle V, s'\rangle$, then V is of the same type (command/integer expression/boolean expression) as P.

Expressions are side-effect free. If $\langle P, s\rangle \Downarrow \langle V, s'\rangle$ and P is an integer or boolean expression, then $s = s'$.

Slide 28

3.4 Exercises

Exercise 3.4.1. By analogy with rules ($\overrightarrow{\text{op1}}$), ($\overrightarrow{\text{set1}}$), ($\overrightarrow{\text{seq1}}$), and ($\overrightarrow{\text{if1}}$) on Slides 18–20, why is there not a rule

$$\frac{\langle B, s\rangle \to \langle B', s'\rangle}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \to \langle \mathbf{while}\ B'\ \mathbf{do}\ C, s'\rangle}\ ?$$

Use this rule (together with the other ones) to derive some transitions that look incorrect compared with the intuitive meaning of while loops (Slide 8) or with the behaviour of the SMC-machine.

Exercise 3.4.2. Let LC' be the language obtained from LC by adding a new command **exit** whose intended behaviour is to immediately abort execution of the smallest enclosing **while**-loop (if any) and return control to the following commands (if any). For example, if

$$\begin{aligned} C \stackrel{\text{def}}{=} {} & \mathbf{while}\ !\ell > 0\ \mathbf{do} \\ & \quad (\mathbf{while\ true\ do} \\ & \quad\quad (\ell := !\ell - 1\ ;\ \mathbf{if}\ !\ell < 0\ \mathbf{then\ exit\ else\ skip})) \end{aligned}$$

then the configuration $\langle C, \{\ell \mapsto 1\}\rangle$ should evaluate to the terminal configuration $\langle \mathbf{skip}, \{\ell \mapsto -1\}\rangle$, whereas $\langle \mathbf{exit}\ ;\ \ell := 0, \{\ell \mapsto 1\}\rangle$ should evaluate to the configuration $\langle \mathbf{exit}, \{\ell \mapsto 1\}\rangle$.

Give an inductively defined evaluation relation for LC', $\Downarrow'$, that captures this intended behaviour. It should be of the form

$$\langle P, s\rangle \Downarrow' \langle V, s'\rangle$$

where P is an LC' phrase, s, s' are states, and V ranges over $\mathbb{B} \cup \mathbb{Z} \cup \{\mathbf{skip}, \mathbf{exit}\}$. It should extend the evaluation relation for LC in the sense that if P does not involve any occurrences of $\mathbf{exit}$ then

$$\langle P, s\rangle \Downarrow' \langle V, s'\rangle \;\Leftrightarrow\; \langle P, s\rangle \Downarrow \langle V, s'\rangle.$$

Check that your rules do give the two evaluations mentioned above.

Exercise 3.4.3. Try to devise a transition semantics for $\mathsf{LC}^{\mathrm{loc}}$ extending the one for LC given in Section 3.1.

Exercise 3.4.4. Call an LC configuration $\langle P, s\rangle$ *sensible* if the set of locations on which s is defined, $dom(s)$, contains all the locations that occur in the phrase P. Prove by induction on the structure of P that if $\langle P, s\rangle$ is sensible, then it is not stuck. Prove by Rule Induction for $\rightarrow$ that if $\langle P, s\rangle \rightarrow \langle P', s'\rangle$ and $\langle P, s\rangle$ is sensible, then so is $\langle P', s'\rangle$ and $dom(s') = dom(s)$. Deduce that a stuck configuration can never be reached by a series of transitions from a sensible configuration.

Exercise 3.4.5. Use Rule Induction to prove each of the statements on Slide 23; in each case define suitable properties $\Phi(P, s, P', s')$ and then check carefully that the properties are closed under the axioms and rules defining $\rightarrow$. Do the same for the corresponding statements on Slide 28, using Rule Induction for the axioms and rules defining $\Downarrow$.

Exercise 3.4.6. Complete the details of the proofs of properties (a) and (b) from Slide 27.

Exercise* **3.4.7.** Prove Theorem 3.3.1.

4 Semantic Equivalence

One of the reasons for wanting to have a formal definition of the semantics of a programming language is that it can serve as the basis of methods for reasoning about program properties and program specifications. In particular, a precise mathematical semantics is necessary for settling questions of *semantic equivalence* of program phrases, in other words for saying precisely when two phrases *have the same meaning*. The different styles of semantics mentioned on Slide 3 have different strengths and weaknesses when it comes to this task.

In an *axiomatic* approach to semantic equivalence, one just postulates axioms and rules for semantic equivalence which will include the general properties of equality shown on Slide 29, together with specific axioms and rules for the various phrase constructions. The importance of the Congruence rule cannot be over emphasised: it lies at the heart of the familiar process of *equational reasoning* whereby an equality is deduced in a number of steps, each step consisting of replacing a subphrase by another phrase already known to be equal to it. (Of course stringing the steps together relies upon the Transitivity rule.) For example, if we already know that $(C\,;C')\,;C''$ and $C\,;(C'\,;C'')$ are equivalent, then we can deduce that

$$\mathbf{while}\ B\ \mathbf{do}\ ((C\,;C')\,;C'') \quad \text{and} \quad \mathbf{while}\ B\ \mathbf{do}\ (C\,;(C'\,;C''))$$

are too, by applying the congruence rule with $\mathcal{C}[-] = \mathbf{while}\ B\ \mathbf{do}\ -$. Note that while Reflexivity, Symmetry and Transitivity are properties that can apply to any binary relation on a set, the Congruence property only makes sense once we have fixed which language we are talking about, and hence which 'contexts' $\mathcal{C}[-]$ are applicable.

How does one know which language-dependent axioms and rules to postulate in an axiomatisation of semantic equivalence? The approach we take here is to regard an operational semantics as part of a language's definition, develop a notion of semantic equivalence based on it, and then validate axioms and rules against this operational equivalence. We will illustrate this approach with respect to the language LC.

Basic properties of equality

Reflexivity $P = P$

Symmetry $\dfrac{P' = P}{P = P'}$

Transitivity $\dfrac{P = P' \quad P' = P''}{P = P''}$

Congruence $\dfrac{P = P'}{\mathcal{C}[P] = \mathcal{C}[P']}$

where $\mathcal{C}[P]$ is a phrase containing an occurrence of P and $\mathcal{C}[P']$ is the same phrase with that occurrence replaced by P'.

Slide 29

Definition of semantic equivalence of LC phrases

Two phrases of the same type are semantically equivalent

$$\boxed{P_1 \cong P_2}$$

if and only if for all states s and all terminal configurations $\langle V, s'\rangle$

$$\langle P_1, s\rangle \Downarrow \langle V, s'\rangle \quad \Leftrightarrow \quad \langle P_2, s\rangle \Downarrow \langle V, s'\rangle .$$

Slide 30

4.1 Semantic equivalence of LC phrases

It is natural to say that two LC phrases of the same type (i.e. both integer expressions, boolean expressions, or commands) are *semantically equivalent* if evaluating them in any given starting state produces exactly the same final state and value (if any). This is formalised on Slide 30. Using the properties of LC evaluation stated on Slide 28, one can reformulate the definition of $\cong$ according to the type of phrase:

- Two LC commands are semantically equivalent, $C_1 \cong C_2$, if and only if they determine the same partial function from states to states: for all s, either $\langle C_1, s\rangle \not\Downarrow$ & $\langle C_2, s\rangle \not\Downarrow$, or $\langle C_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$ & $\langle C_2, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$ for some s'.

- Two LC integer expressions are semantically equivalent, $E_1 \cong E_2$, if and only if they determine the same partial function from states to integers: for all s, either $\langle E_1, s\rangle \not\Downarrow$ & $\langle E_2, s\rangle \not\Downarrow$, or $\langle E_1, s\rangle \Downarrow \langle n, s\rangle$ & $\langle E_2, s\rangle \Downarrow \langle n, s\rangle$ for some $n \in \mathbb{Z}$.

- Two LC boolean expressions are semantically equivalent, $B_1 \cong B_2$, if and only if they determine the same partial function from states to booleans: for all s, either $\langle B_1, s\rangle \not\Downarrow$ & $\langle B_2, s\rangle \not\Downarrow$, or $\langle B_1, s\rangle \Downarrow \langle b, s\rangle$ & $\langle B_2, s\rangle \Downarrow \langle b, s\rangle$ for some $b \in \mathbb{B}$.

Slide 31 spells out what is required to show that two LC commands are *not* semantically equivalent; we write $C_1 \not\cong C_2$ in this case. There are similar characterisations of semantic inequivalence for LC integer and boolean expressions.

Semantic inequivalence of LC commands

To show $C_1 \not\cong C_2$, it suffices to find states s, s' such that

either $\langle C_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$ and $\langle C_2, s\rangle \not\Downarrow \langle \mathbf{skip}, s'\rangle$,

or $\langle C_1, s\rangle \not\Downarrow \langle \mathbf{skip}, s'\rangle$ and $\langle C_2, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$

E.g. (Exercise 4.3.2) when $C = \mathbf{skip}$, $C' = \mathbf{while\ true\ do\ skip}$, $C'' = (\ell := 1)$ and $B = (!\ell = 0)$, then

$$C'';(\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C') \not\cong \mathbf{if}\ B\ \mathbf{then}\ (C'';C)\ \mathbf{else}\ C'';C')$$

Slide 31

Example 4.1.1.

$$(\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C')\ ;C'' \cong \mathbf{if}\ B\ \mathbf{then}\ (C\ ;C'')\ \mathbf{else}\ (C'\ ;C'')$$

Proof. Write

$$\begin{aligned} C_1 &\stackrel{\text{def}}{=} (\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C')\ ;C'' \\ C_2 &\stackrel{\text{def}}{=} \mathbf{if}\ B\ \mathbf{then}\ (C\ ;C'')\ \mathbf{else}\ (C'\ ;C''). \end{aligned}$$

We exploit the structural nature of the rules in Figure 2 that inductively define the LC evaluation relation (and also the properties listed on Slide 28, in order to mildly simplify the case analysis). If it is the case that $\langle C_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$, because of the structure of C_1 this must have been deduced using $(\Downarrow_{\text{seq}})$. So for some s'' we have

$$\langle \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C', s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle \tag{17}$$

$$\langle C'', s''\rangle \Downarrow \langle \mathbf{skip}, s'\rangle. \tag{18}$$

The rule used to deduce (17) must be either $(\Downarrow_{\text{if1}})$ or $(\Downarrow_{\text{if2}})$. So

$$\begin{cases} \text{either} & \langle B, s\rangle \Downarrow \langle \mathbf{true}, s\rangle \ \&\ \langle C, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle, \\ \text{or} & \langle B, s\rangle \Downarrow \langle \mathbf{false}, s\rangle \ \&\ \langle C', s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle \end{cases} \tag{19}$$

(where we have made use of the fact that evaluation of expressions is side-effect free—cf. Slide 28). In either case, combining (19) with (18) and applying $(\Downarrow_{\text{seq}})$ we get

$$\begin{cases} \text{either} & \langle B, s\rangle \Downarrow \langle \mathbf{true}, s\rangle \ \&\ \langle C\ ;C'', s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle, \\ \text{or} & \langle B, s\rangle \Downarrow \langle \mathbf{false}, s\rangle \ \&\ \langle C'\ ;C'', s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle. \end{cases} \tag{20}$$

But then $(\Downarrow_{\text{if1}})$ or $(\Downarrow_{\text{if2}})$ applied to (20) yields $\langle C_2, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$ in either case.

Similarly, starting from $\langle C_2, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$ we can deduce $\langle C_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle$. Since this holds for any s, s', we have $C_1 \cong C_2$, as required. □

Slide 32 lists some other examples of semantically equivalent commands whose proofs we leave as exercises.

Examples of semantically equivalent LC commands

$$C \mathbin{;} \mathbf{skip} \cong C \cong \mathbf{skip} \mathbin{;} C$$

$$(C \mathbin{;} C') \mathbin{;} C'' \cong C \mathbin{;} (C' \mathbin{;} C'')$$

$$\mathbf{if\ true\ then}\ C\ \mathbf{else}\ C' \cong C$$

$$\mathbf{if\ false\ then}\ C\ \mathbf{else}\ C' \cong C'$$

$$\begin{aligned} \mathbf{while}\ B\ \mathbf{do}\ C \cong\ & \mathbf{if}\ B\ \mathbf{then}\ C \mathbin{;} (\mathbf{while}\ B\ \mathbf{do}\ C) \\ & \mathbf{else\ skip} \end{aligned}$$

$$\ell := n \mathbin{;} \ell' := n' \cong \begin{cases} \ell' := n' \mathbin{;} \ell := n & \text{if } \ell \neq \ell' \\ \ell := n' & \text{if } \ell = \ell'. \end{cases}$$

Slide 32

Theorem 4.1.2. LC *semantic equivalence satisfies the properties of Reflexivity, Symmetry, Transitivity and Congruence given on Slide* 29.

Proof. The only one of the properties that does not follow immediately from the definition of $\cong$ is Congruence:

$$P_1 \cong P_2 \quad \Rightarrow \quad \mathcal{C}[P_1] \cong \mathcal{C}[P_2].$$

Analysing the structure of of LC contexts, this amounts to proving each of the properties listed in Figure 4. Most of these follow by routine case analysis of proofs of evaluation, along the lines of Example 4.1.1. The only non-trivial one is

$$C_1 \cong C_2 \quad \Rightarrow \quad \mathbf{while}\ B\ \mathbf{do}\ C_1 \cong \mathbf{while}\ B\ \mathbf{do}\ C_2$$

the proof of which we give.

For commands: if $C_1 \cong C_2$ then for all C and B

$$\begin{aligned}
C_1 ; C &\cong C_2 ; C \\
C ; C_1 &\cong C ; C_2 \\
\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C &\cong \mathbf{if}\ B\ \mathbf{then}\ C_2\ \mathbf{else}\ C \\
\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C_1 &\cong \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C_2 \\
\mathbf{while}\ B\ \mathbf{do}\ C_1 &\cong \mathbf{while}\ B\ \mathbf{do}\ C_2.
\end{aligned}$$

For integer expressions: if $E_1 \cong E_2$ then for all ℓ and E

$$\begin{aligned}
\ell := E_1 &\cong \ell := E_2 \\
E_1\ op\ E &\cong E_2\ op\ E \\
E\ op\ E_1 &\cong E\ op\ E_2.
\end{aligned}$$

For boolean expressions: if $B_1 \cong B_2$ then for all C and C'

$$\begin{aligned}
\mathbf{if}\ B_1\ \mathbf{then}\ C\ \mathbf{else}\ C' &\cong \mathbf{if}\ B_2\ \mathbf{then}\ C\ \mathbf{else}\ C' \\
\mathbf{while}\ B_1\ \mathbf{do}\ C &\cong \mathbf{while}\ B_2\ \mathbf{do}\ C.
\end{aligned}$$

Figure 4: Congruence properties of LC semantic equivalence

Proof of

$$\boxed{C_1 \cong C_2 \quad \Rightarrow \quad \mathbf{while}\ B\ \mathbf{do}\ C_1 \cong \mathbf{while}\ B\ \mathbf{do}\ C_2}$$

is via:

Lemma. *If $C_1 \cong C_2$, then for all $n \geq 0$*

$$(\dagger_n) \begin{cases} \forall m \leq n. \forall s, s'. \\ \quad \langle \mathbf{while}\ B\ \mathbf{do}\ C_1, s\rangle \to^m \langle \mathbf{skip}, s'\rangle \\ \qquad \Rightarrow \langle \mathbf{while}\ B\ \mathbf{do}\ C_2, s\rangle \to^* \langle \mathbf{skip}, s'\rangle \end{cases}$$

(where $\to^m$ means the composition of m transitions and $\to^*$ means $\to^m$ holds for some $m \geq 0$).

Slide 33

If suffices to show that if $C_1 \cong C_2$ then

(21) $$\forall s, s' \,.\, \langle \mathbf{while}\ B\ \mathbf{do}\ C_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle \Rightarrow \langle \mathbf{while}\ B\ \mathbf{do}\ C_2, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle.$$

The recursive nature of the **while** construct (rule ($\Downarrow_{\mathrm{wh1}}$) in particular) makes it difficult to give a direct proof of this (try it and see). Instead we resort to the theorem given on Slide 27 which characterises evaluation in terms of the transition relation $\to$. Using this theorem, to prove (21), it suffices to prove the Lemma on Slide 33. We do this by Mathematical Induction on n. The base case $n = 0$ is vacuously true, since '$\to^0$' means '$=$' and $\langle \mathbf{while}\ B\ \mathbf{do}\ C_1, s\rangle \neq \langle \mathbf{skip}, s'\rangle$. For the induction step, suppose that $C_1 \cong C_2$, that ($\dagger_n$) holds, and that we have

(22) $$\langle \mathbf{while}\ B\ \mathbf{do}\ C_1, s\rangle \to^{n+1} \langle \mathbf{skip}, s'\rangle.$$

We have to prove that $\langle \mathbf{while}\ B\ \mathbf{do}\ C_2, s\rangle \to^* \langle \mathbf{skip}, s'\rangle$.

The structural nature of the rules inductively generating $\to$ (given on Slides 18–20) mean that the transition sequence (22) starts off with an instance of axiom ($\overrightarrow{\mathrm{wh1}}$):

$$\langle \mathbf{while}\ B\ \mathbf{do}\ C_1, s\rangle \to \langle \mathbf{if}\ B\ \mathbf{then}\ C_1 \,;W_1\ \mathbf{else\ skip}, s\rangle \to^n \langle \mathbf{skip}, s'\rangle$$

where we write W_i for $\mathbf{while}\ B\ \mathbf{do}\ C_i$ $(i = 1, 2)$. Now there are two cases according to how B evaluates.

Case $\langle B, s\rangle \to^* \langle \mathbf{true}, s\rangle$. Then (22) looks like

$$\begin{aligned}
\langle W_1, s\rangle &\to \langle \mathbf{if}\ B\ \mathbf{then}\ C_1 \,; W_1\ \mathbf{else\ skip}, s\rangle \\
&\to^* \langle \mathbf{if\ true\ then}\ C_1 \,; W_1\ \mathbf{else\ skip}, s\rangle \\
&\to \langle C_1 \,; W_1, s\rangle \\
&\to^* \langle \mathbf{skip} \,; W_1, s''\rangle \\
&\to \langle W_1, s''\rangle \\
&\to^m \langle \mathbf{skip}, s'\rangle
\end{aligned}$$

for some s'' and some $m \leq n$ (less than or equal to $n-2$, in fact). Since ($\dagger_n$) holds by assumption, we have $\langle W_2, s''\rangle \to^* \langle \mathbf{skip}, s'\rangle$. Furthermore, the transitions $\langle C_1 \,; W_1, s\rangle \to^* \langle \mathbf{skip} \,; W_1, s''\rangle$ in the middle of the above sequence must have been deduced by applying rule ($\overrightarrow{\mathrm{seq1}}$) to $\langle C_1, s\rangle \to^* \langle \mathbf{skip}, s''\rangle$. Since $C_1 \cong C_2$, it follows that $\langle C_2, s\rangle \to^* \langle \mathbf{skip}, s''\rangle$ and hence by rule ($\overrightarrow{\mathrm{seq1}}$) also that $\langle C_2 \,; W_2, s\rangle \to^* \langle \mathbf{skip} \,; W_2, s''\rangle$. Therefore we can construct a transition sequence structured like the one displayed above which shows that $\langle W_2, s\rangle \to^* \langle \mathbf{skip}, s'\rangle$, as required.

Case $\langle B, s\rangle \to^* \langle \mathbf{false}, s\rangle$. Then (22) looks like

$$\begin{aligned}
\langle W_1, s\rangle &\to \langle \mathbf{if}\ B\ \mathbf{then}\ C_1 \,; W_1\ \mathbf{else\ skip}, s\rangle \\
&\to^* \langle \mathbf{if\ false\ then}\ C_1 \,; W_1\ \mathbf{else\ skip}, s\rangle \\
&\to \langle \mathbf{skip}, s\rangle
\end{aligned}$$

(and in particular $s' = s$). But this sequence does not depend upon the evaluation behaviour of C_1 and equally we have $\langle W_2, s\rangle \to^* \langle \mathbf{skip}, s\rangle$, as required. $\square$

$\mathsf{LC}^{\mathrm{loc}}$: LC+ block structured local state

Phrases: $P ::= C \mid E \mid B$

Commands:

$$\begin{array}{rcl} C & ::= & \mathbf{skip} \mid \ell := E \mid C\,;C \mid \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C \\ & \mid & \mathbf{while}\ B\ \mathbf{do}\ C \mid \boxed{\mathbf{begin\ loc}\,\ell := E;\ C\ \mathbf{end}} \end{array}$$

Integer expressions: $E ::= n \mid !\ell \mid E\ iop\ E$

Boolean expressions: $B ::= b \mid E\ bop\ E$

Slide 34

4.2 Block structured local state

Because of the need to control interference between the state in different program parts, most procedural languages include the facility for declarations of locally scoped locations (program variables) whose evaluation involves the dynamic creation of fresh storage locations. In this section we consider semantic equivalence of commands involving a particularly simple form of such *local state*, $\mathbf{begin\ loc}\,\ell := E;\ C\ \mathbf{end}$, in which the life time of the freshly created location correlates precisely with the textual scope of the declaration: the location ℓ is created and initialised with the value of E at the beginning of the program 'block' C and deallocated at the end of the block. We call the language obtained from LC by adding this construct $\mathsf{LC}^{\mathrm{loc}}$: see Slide 34. Taking configurations to be as before (i.e. (command,state)-pairs), we can specify the operational semantics of $\mathsf{LC}^{\mathrm{loc}}$by an evaluation relation inductively defined by the rules in Figure 2 and Slide 26, together with the rule for blocks on Slide 35.

Evaluation rule for blocks

$$(\Downarrow_{\mathrm{bs}})\qquad \frac{\begin{array}{c}\langle E, s\rangle \Downarrow \langle n, s'\rangle \\ \langle C[\ell'/\ell], s'[\ell' \mapsto n]\rangle \Downarrow \langle \mathbf{skip}, s''[\ell' \mapsto n']\rangle\end{array}}{\langle \mathbf{begin\ loc}\,\ell := E;\ C\ \mathbf{end}, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}\,*$$

$*$ if $\ell' \notin dom(s') \cup dom(s'')$ and ℓ' does not occur in C.

$C[\ell'/\ell]$ indicates the $\mathrm{LC}^{\mathrm{loc}}$ command obtained from C by replacing all occurrences of ℓ with ℓ'.

Slide 35

Example 4.2.1. To see how rule ($\Downarrow_{\mathrm{bs}}$) works in practice, consider a command to swap the contents of two locations using a temporary location that happens to have the same name as a global one.

$$C \stackrel{\mathrm{def}}{=} \mathbf{begin\ loc}\,\ell := !\ell_1;\ \ell_1 := !\ell_2\ ;\ \ell_2 := !\ell\ \mathbf{end}.$$

Here we assume ℓ, ℓ_1, ℓ_2 are three distinct locations. Then for all states s with $\ell_1, \ell_2 \in dom(s)$ we have

$$\langle C, s\rangle \Downarrow \langle \mathbf{skip}, s[\ell_1 \mapsto s(\ell_2), \ell_2 \mapsto s(\ell_1)]\rangle.$$

and in particular the value stored at ℓ in the final state $s[\ell_1 \mapsto s(\ell_2), \ell_2 \mapsto s(\ell_1)]$ (if any) is the same as it is in the initial state s.

Proof. Let $n_i = s(\ell_i)$ $(i = 1, 2)$ and

$$s_1 \stackrel{\mathrm{def}}{=} s[\ell_1 \mapsto n_2], \quad s_2 \stackrel{\mathrm{def}}{=} s[\ell_1 \mapsto n_2, \ell_2 \mapsto n_1]$$

and choose any $\ell' \notin \{\ell, \ell_1, \ell_2\}$. Then

$$\frac{\dfrac{}{\langle !\ell_1, s\rangle \Downarrow \langle n_1, s\rangle}\,(\Downarrow_{\mathrm{loc}}) \quad \dfrac{\dfrac{\dfrac{}{\langle !\ell_2, s[\ell' \mapsto n_1]\rangle \Downarrow \langle n_2, s[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{loc}})}{\langle \ell_1 := !\ell_2, s[\ell' \mapsto n_1]\rangle \Downarrow \langle \mathbf{skip}, s_1[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{set}}) \quad \dfrac{\dfrac{}{\langle !\ell', s_1[\ell' \mapsto n_1]\rangle \Downarrow \langle n_1, s_1[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{loc}})}{\langle \ell_2 := !\ell', s_1[\ell' \mapsto n_1]\rangle \Downarrow \langle \mathbf{skip}, s_2[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{set}})}{\langle \ell_1 := !\ell_2\ ;\ \ell_2 := !\ell', s[\ell' \mapsto n_1]\rangle \Downarrow \langle \mathbf{skip}, s_2[\ell' \mapsto n_1]\rangle}\,(\Downarrow_{\mathrm{seq}})}{\langle \mathbf{begin\ loc}\,\ell := !\ell_1;\ \ell_1 := !\ell_2\ ;\ \ell_2 := !\ell\ \mathbf{end}, s\rangle \Downarrow \langle \mathbf{skip}, s_2\rangle}\,(\Downarrow_{\mathrm{bs}}).$$

is a proof for the claimed evaluation. $\square$

The definition of semantic equivalence for $\mathsf{LC}^{\mathrm{loc}}$ phrases is exactly the same as for LC (see Slide 30). Slide 36 gives an example of semantically equivalent $\mathsf{LC}^{\mathrm{loc}}$ commands.

Example of semantically equivalent $\mathsf{LC}^{\mathrm{loc}}$ commands

[Cf. Tripos question 1999.5.9]

If $\ell \neq \ell'$, then

$$(\mathbf{begin\ loc}\,\ell := E;\ \ell' := !\ell\ \mathbf{end})\ \cong\ (\ell' := E)$$

What happens if $\ell = \ell'$?

Slide 36

Proof of the equivalence on Slide 36. Given any states s and s', suppose

(23) $$\langle(\mathbf{begin\ loc}\,\ell := E;\ \ell' := !\ell\ \mathbf{end}), s\rangle \Downarrow \langle\mathbf{skip}, s'\rangle.$$

This can only have been deduced by applying rule ($\Downarrow_{\mathrm{bs}}$) to

(24) $$\langle E, s\rangle \Downarrow \langle n, s''\rangle$$

(25) $$\langle \ell' := !\ell'', s''[\ell'' \mapsto n]\rangle \Downarrow \langle\mathbf{skip}, s'[\ell'' \mapsto n']\rangle$$

for some n, n' and s'' with $\ell'' \notin dom(s') \cup dom(s'') \cup \{\ell, \ell'\}$. Note that for this to be a correct application of ($\Downarrow_{\mathrm{bs}}$), we need to know that $\ell \neq \ell'$. (What happens in case $\ell = \ell'$? See Exercise 4.3.4.)

Now (25) can only hold because

(26) $$s' = s''[\ell' \mapsto n] \quad \text{and} \quad n' = n.$$

Applying ($\Downarrow_{\mathrm{set}}$) from Figure 2 to (24) yields $\langle \ell' := E, s\rangle \Downarrow \langle\mathbf{skip}, s''[\ell' \mapsto n]\rangle$ and hence by (26) that

(27) $$\langle \ell' := E, s\rangle \Downarrow \langle\mathbf{skip}, s'\rangle.$$

Thus (23) implies (27) for any s, s' and so we have proved half of the bi-implication needed for the semantic equivalence on Slide 36. Conversely, if (27) holds, it must have been deduced by applying $(\Downarrow_{\mathrm{set}})$ to (24) with $s' = s''[\ell' \mapsto n]$ for some n and s''; in which case (25) holds and hence by $(\Downarrow_{\mathrm{bs}})$ (once again using the fact that $\ell \neq \ell'$) so does (23). $\square$

4.3 Exercises

Exercise 4.3.1. Prove the LC semantic equivalences listed on Slide 32.

Exercise 4.3.2. Show by example that the LC command $C\,; (\mathbf{if}\ B\ \mathbf{then}\ C'\ \mathbf{else}\ C'')$ is not semantically equivalent to $\mathbf{if}\ B\ \mathbf{then}\ (C\,; C')\ \mathbf{else}\ (C\,; C'')$ in general. What happens if the locations assigned to in C are disjoint from the locations occurring in $loc(B)$?

Exercise 4.3.3. Prove the properties listed in Figure 4.

Exercise 4.3.4. Show by example that $\mathbf{begin}\ \mathbf{loc}\,\ell := E;\ \ \ell' := !\ell\ \mathbf{end}$ is not necessarily semantically equivalent to $\ell' := E$ in the case that ℓ and ℓ' are equal.

5 Functions

In this section we consider functional and procedural abstractions and the structural operational semantics of two associated calling mechanisms—*call-by-name* and *call-by-value*. To do this we use a Language of (higher order) Functions and Procedures, called LFP, that combines LC with the *simply typed lambda calculus* (cf. Winskel 1993, Chapter 11 and Gunter 1992, Chapter 2). LC phrases were divided into three syntactic categories—integer expressions, boolean expressions, and commands. By contrast, the grammar on Slide 37 specifies the LFP syntax in terms of a single syntactic category of *expressions* which later we will classify into different types using an inductively defined typing relation.

The major difference between LFP and LC lies in the last three items in the grammar on Slide 37. LFP has *variables*, x, standing for unknown LFP expressions and used as parameters in function abstractions. The expression $\lambda x. M$ is a *function abstraction*—a way of referring to the function mapping x to M without having to explicitly assign it a name; it is also a procedure abstraction, because we will identify procedures with functions whose bodies M are expressions of command type. Finally $M\, M'$ is an expression denoting the application of a function M to an argument M'.

LFP also generalises LC in a number of more minor ways. First, LFP has a branching construct for all types of expression, rather than just for commands. Secondly, locations (ℓ) are now first class expressions whereas in LC they only appeared indirectly, via assignment commands ($\ell := E$) and look-up expressions ($!\ell$); furthermore, compound expressions are allowed in look-ups and on the left-hand side of assignment.

Note. What we here call 'variables' are variables in the logical sense—placeholders standing for unknown quantities and for which substitutions can be made. In the context of programming languages they are often called 'identifiers', because what we here refer to as locations are very often called 'variables' (because their contents may vary during execution and because it is common to use the name of a storage location without qualification to denote its contents). What we here call 'function abstractions' are also called *lambda abstractions* because of the notation $\lambda x. M$ introduced by Church in his lambda calculus—see the Part IB course on **Foundations of Functional Programming**.

Expressions of the LFP language

$$
\begin{aligned}
M ::= {} & n \mid b \mid \ell \mid M \; iop \; M \mid M \; bop \; M \\
& \mid \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \mid {!M} \mid M := M \\
& \mid \mathbf{skip} \mid M \mathbin{;} M \mid \mathbf{while}\ M\ \mathbf{do}\ M \\
& \mid x \mid \lambda x.\, M \mid M\, M
\end{aligned}
$$

where
$x \in \mathbb{V}$, an infinite set of *variables*,
$n \in \mathbb{Z}$ (integers), $b \in \mathbb{B}$ (booleans), $\ell \in \mathbb{L}$ (locations),
$iop \in \mathbb{I}op$ (integer-valued binary operations), and $bop \in \mathbb{B}op$ (boolean-valued binary operations).

Slide 37

5.1 Substitution and α-conversion

When it comes to function application, the operational semantics of LFP will involve the syntactic operation $M[M'/x]$ of *substituting an expression M' for all free occurrences of the variable x in the expression M*. This operation involves several subtleties, illustrated on Slide 38, which arise from the fact that $M \mapsto \lambda x.\, M$ is a variable-binding operation. The occurrence of x next to λ in $\lambda x.\, M$ is a *binding occurrence* of the variable x whose *scope* is the whole syntax tree M; and in $\lambda x.\, M$ no occurrences of x in M are free for substitution by another expression (see example (ii) on Slide 38). The finite set of *free variables* of an expression is defined on Slide 39. The key clause is the last one—x is not a free variable of $\lambda x.\, M$.

In fact we need the operation of *simultaneously* substituting expressions for a number of different free variables in an expression. Given a *substitution* σ, i.e. a finite partial function mapping variables to LFP expressions, $M[\sigma]$ will denote the LFP expression resulting from simultaneous substitution of each $x \in dom(\sigma)$ by the corresponding expression $\sigma(x)$. It is defined by induction on the structure of M (simultaneously for all substitutions σ) in Figure 5 (cf. Stoughton 1988). Then we can take $M[M'/x]$ to be $M[\sigma]$ with $\sigma = \{x \mapsto M'\}$.

Substitution examples

$M[M'/x]$ — substitute M' for all free occurrences of the variable x in the expression M.

(i) $(\lambda x.\, x+y)[4/y]$ is $\lambda x.\, x+4$.

(ii) $(\lambda x.\, x+y)[4/x]$ is $\lambda x.\, x+y$, not $\lambda x.\, 4+y$, because $\lambda x.\, x+y$ contains no *free* occurrence of x.

(iii) $\lambda x.\, x+y$ is the same as (is α-*convertible* with) $\lambda z.\, z+y$; and $(\lambda x.\, x+y)[x/y]$ is $\lambda z.\, z+x$, not $\lambda x.\, x+x$.

Slide 38

$fv(M)$ **— set of *free variables* of** M

$$fv(n)=fv(b)=fv(\ell)=fv(\mathbf{skip}) \stackrel{\text{def}}{=} \emptyset$$

$$fv(!M) \stackrel{\text{def}}{=} fv(M)$$

$$fv(M\ op\ M')=fv(M := M')=fv(M\ ;\ M')=$$
$$=fv(\mathbf{while}\ M\ \mathbf{do}\ M')=fv(M\ M') \stackrel{\text{def}}{=} fv(M) \cup fv(M')$$

$$fv(\mathbf{if}\ M\ \mathbf{then}\ M'\ \mathbf{else}\ M'') \stackrel{\text{def}}{=} fv(M) \cup fv(M') \cup fv(M'')$$

$$fv(x) \stackrel{\text{def}}{=} \{x\}$$

$$fv(\lambda x.\, M) \stackrel{\text{def}}{=} \{x' \in fv(M) \mid x' \neq x\}.$$

Slide 39

- $x[\sigma] \stackrel{\text{def}}{=} \begin{cases} \sigma(x) & \text{if } x \in dom(\sigma) \\ x & \text{otherwise.} \end{cases}$
- $n[\sigma] \stackrel{\text{def}}{=} n$. Similarly for b, ℓ, and $\mathbf{skip}$.
- $(M\,M')[\sigma] \stackrel{\text{def}}{=} (M[\sigma])\,(M'[\sigma])$. Similarly for $M\ op\ M'$, $M := M'$, $M\ ;\ M'$, $\mathbf{while}\ M\ \mathbf{do}\ M'$, $!M$, and $\mathbf{if}\ M\ \mathbf{then}\ M'\ \mathbf{else}\ M''$.
- $(\lambda x.\, M)[\sigma] \stackrel{\text{def}}{=} \lambda x'.(M[\sigma[x \mapsto x']])$, where x' is the first variable not in $fv(\sigma) \cup fv(M)$.

Notes

In the last clause of the definition:

– $\sigma[x \mapsto x']$ is the substitution mapping x to x' and otherwise acting like σ.

– x' is first with respect to some fixed ordering of the set $\mathbb{V}$ of variables that we assume is given.

– $fv(\sigma) \stackrel{\text{def}}{=} \{y \mid \exists x \in dom(\sigma).\, y \in fv(\sigma(x))\}$ is the set of all free variables in the expressions being substituted by σ.

– Since $x' \notin fv(\sigma) \cup fv(M)$, the only occurrences of x' in $M[\sigma[x \mapsto x']]$ that are 'captured' by $\lambda x'.\,(-)$ correspond to occurrences of x in M that were bound in $\lambda x.\, M$.

Figure 5: Definition of substitution

α-Conversion relation

is inductively defined by the following axioms and rules:

$$\lambda x.\, M \equiv_\alpha \lambda x'.(M[x'/x]) \qquad M \equiv_\alpha M$$

$$\frac{M \equiv_\alpha M'}{M' \equiv_\alpha M} \qquad \frac{M \equiv_\alpha M' \quad M' \equiv_\alpha M''}{M \equiv_\alpha M''}$$

$$\frac{M \equiv_\alpha M'}{\lambda x.\, M \equiv_\alpha \lambda x.\, M'} \qquad \frac{M_1 \equiv_\alpha M_1' \quad M_2 \equiv_\alpha M_2'}{M_1\, M_2 \equiv_\alpha M_1'\, M_2'}$$

plus rules like the last one for each of the other LFP expression-forming constructs.

Slide 40

LFP **terms**

We identify LFP expressions up to α-conversion:

> An LFP *term* is by definition an $\equiv_\alpha$-equivalence class of LFP expressions.

But we will not make a notational distinction between an expression M and the LFP term it determines.

In using an expression to represent a term, we usually choose one whose bound variables are all distinct from each other and from any variables in the context of use.

Slide 41

Note how the last clause in Figure 5 avoids the problem of unwanted ‘capture’ of free variables in an expression being substituted, illustrated by example (iii) on Slide 38. It does so by ‘α-converting’ the bound variable. There is no problem with this from a semantical point of view, since in general we expect the meaning of a function abstraction to be independent of the name of the bound variable—$\lambda x.\, M$ and $\lambda x'.\, M[x'/x]$ should always mean the same thing. The equivalence relation $\equiv_\alpha$ of *α-conversion* between LFP expressions is defined on Slide 40. In Section 2.1 we noted that the representation of syntax as parse trees rather than as strings of symbols is the proper level of abstraction when discussing programming language semantics. In fact when the language involves binding constructs one should take this a step further and use a representation of syntax that identifies α-convertible expressions. It is possible to do this in a clever way that still allows expressions to be tree-like data structures through the use of de Bruijn’s ‘nameless terms’, but at the expense of complicating the definition of substitution: the interested reader is referred to (Barendregt 1984, Appendix C). Here we will use brute force and quotient the set of expressions by the equivalence relation $\equiv_\alpha$: see Slide 41. The convention mentioned on that slide—not making any notational distinction between an expression M and the LFP term it determines—is possible because the operations on syntax that we will employ all respect α-conversion. For example, and as one might expect, it is the case that the operation of substitution respects $\equiv_\alpha$:

$$(M_1 \equiv_\alpha M_1' \;\&\; M_2 \equiv_\alpha M_2') \Rightarrow M_1[M_2/x] \equiv_\alpha M_1'[M_2'/x].$$

Similarly, the set of free variables of an expression is invariant with respect to $\equiv_\alpha$:

$$M \equiv_\alpha M' \quad \Rightarrow \quad fv(M) = fv(M').$$

5.2 Call-by-name and call-by-value

We will give the structural operational semantics of LFP in terms of an inductively defined relation of evaluation whose form is shown on Slide 42. Compared with LC, the main novelty lies in the rules for evaluating function abstractions and function application. For function abstractions, we take configurations of the form $\langle \lambda x.\, M, s\rangle$ to be terminal. For function application, there are (at least) two natural strategies, depending upon whether or not an argument is evaluated before it is passed to the body of a function abstraction. These strategies are shown on Slide 43. Many pragmatic considerations to do with implementation influence which one to choose. The different strategies also radically alter the properties of evaluation and the ease with which one can reason about program properties—we shall see something of this below.

LFP evaluation relation

takes the form:

$$\langle M, s\rangle \Downarrow \langle V, s'\rangle$$

where

- M and V are *closed* terms, i.e. $fv(M) = fv(V) = \emptyset$.
- s and s' are *states*, i.e. finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$.
- V is a *value*, $V ::= n \mid b \mid \ell \mid \mathbf{skip} \mid \lambda x.\, M$.

Slide 42

Call-by-name and call-by-value evaluation

$$(\Downarrow_{\mathrm{cbn}}) \qquad \frac{\begin{array}{c}\langle M_1, s\rangle \Downarrow \langle \lambda x.\, M_1', s'\rangle \\ \langle M_1'[M_2/x], s'\rangle \Downarrow \langle V, s''\rangle\end{array}}{\langle M_1\, M_2, s\rangle \Downarrow \langle V, s''\rangle}$$

$$(\Downarrow_{\mathrm{cbv}}) \qquad \frac{\begin{array}{c}\langle M_1, s\rangle \Downarrow \langle \lambda x.\, M_1', s'\rangle \\ \langle M_2, s'\rangle \Downarrow \langle V_2, s''\rangle \\ \langle M_1'[V_2/x], s''\rangle \Downarrow \langle V, s'''\rangle\end{array}}{\langle M_1\, M_2, s\rangle \Downarrow \langle V, s'''\rangle}$$

Slide 43

$(\Downarrow_{\mathrm{val}})$ $\quad \langle V, s\rangle \Downarrow \langle V, s\rangle$ (V a value)

$(\Downarrow_{\mathrm{op}})$ $\quad \dfrac{\langle M_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle M_1 \; op \; M_2, s\rangle \Downarrow \langle c, s''\rangle}$ where c is the value of $n_1 \; op \; n_2$ (for op an integer or boolean operation)

$(\Downarrow_{\mathrm{if1}})$ $\quad \dfrac{\langle M_1, s\rangle \Downarrow \langle \mathbf{true}, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle V, s''\rangle}{\langle \mathbf{if}\; M_1 \;\mathbf{then}\; M_2 \;\mathbf{else}\; M_3, s\rangle \Downarrow \langle V, s''\rangle}$

$(\Downarrow_{\mathrm{if2}})$ $\quad \dfrac{\langle M_1, s\rangle \Downarrow \langle \mathbf{false}, s'\rangle \quad \langle M_3, s'\rangle \Downarrow \langle V, s''\rangle}{\langle \mathbf{if}\; M_1 \;\mathbf{then}\; M_2 \;\mathbf{else}\; M_3, s\rangle \Downarrow \langle V, s''\rangle}$

$(\Downarrow_{!})$ $\quad \dfrac{\langle M_1, s\rangle \Downarrow \langle \ell, s'\rangle}{\langle !M_1, s\rangle \Downarrow \langle n, s'\rangle}$ if $\ell \in dom(s') \;\&\; s'(\ell) = n$

$(\Downarrow_{:=})$ $\quad \dfrac{\langle M_1, s\rangle \Downarrow \langle \ell, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle n, s''\rangle}{\langle M_1 := M_2, s\rangle \Downarrow \langle \mathbf{skip}, s''[\ell \mapsto n]\rangle}$

$(\Downarrow_{\mathrm{seq}})$ $\quad \dfrac{\langle M_1, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}{\langle M_1 \,;\, M_2, s\rangle \Downarrow \langle \mathbf{skip}, s''\rangle}$

$(\Downarrow_{\mathrm{wh1}})$ $\quad \dfrac{\begin{array}{c}\langle M_1, s\rangle \Downarrow \langle \mathbf{true}, s'\rangle \quad \langle M_2, s'\rangle \Downarrow \langle \mathbf{skip}, s''\rangle \\ \langle \mathbf{while}\; M_1 \;\mathbf{do}\; M_2, s''\rangle \Downarrow \langle \mathbf{skip}, s'''\rangle\end{array}}{\langle \mathbf{while}\; M_1 \;\mathbf{do}\; M_2, s\rangle \Downarrow \langle \mathbf{skip}, s'''\rangle}$

$(\Downarrow_{\mathrm{wh2}})$ $\quad \dfrac{\langle M_1, s\rangle \Downarrow \langle \mathbf{false}, s'\rangle}{\langle \mathbf{while}\; M_1 \;\mathbf{do}\; M_2, s\rangle \Downarrow \langle \mathbf{skip}, s'\rangle}$

plus either rule $(\Downarrow_{\mathrm{cbn}})$ or rule $(\Downarrow_{\mathrm{cbv}})$ on Slide 43.

Figure 6: Axioms and rules for LFP evaluation

For LFP, $\Downarrow_{\mathrm{n}}$ and $\Downarrow_{\mathrm{v}}$ are incomparable

Let

$$C_0 \stackrel{\text{def}}{=} \mathbf{while}\ \mathbf{true}\ \mathbf{do}\ \mathbf{skip}$$

$$C_1 \stackrel{\text{def}}{=} (\lambda x.\, \mathbf{skip})\, C_0$$

$$C_2 \stackrel{\text{def}}{=} (\lambda x.\, \mathbf{if}\ !\ell = 0\ \mathbf{then}\ \mathbf{skip}\ \mathbf{else}\ C_0)\,(\ell := 0).$$

Then

$$\langle C_1, s\rangle \Downarrow_{\mathrm{n}} \langle \mathbf{skip}, s\rangle \quad \text{(any } s\text{)}$$
$$\langle C_1, s\rangle \not\Downarrow_{\mathrm{v}}$$

$$\langle C_2, \{\ell \mapsto 1\}\rangle \not\Downarrow_{\mathrm{n}}$$
$$\langle C_2, \{\ell \mapsto 1\}\rangle \Downarrow_{\mathrm{v}} \langle \mathbf{skip}, \{\ell \mapsto 0\}\rangle.$$

Slide 44

Definition 5.2.1. The *call-by-name* (respectively *call-by-value*) *evaluation relation* for LFP terms is denoted $\Downarrow_{\mathrm{n}}$ (respectively $\Downarrow_{\mathrm{v}}$) and is inductively generated by the rule ($\Downarrow_{\mathrm{cbn}}$) (respectively ($\Downarrow_{\mathrm{cbv}}$)) on Slide 43 together with the axioms and rules in Figure 6.

The examples on Slide 44 exploit the fact that evaluation of LFP terms can cause state change to show that there is no implication either way between call-by-value convergence and call-by-name convergence. The following notation is used on the slide:

$$\langle M, s\rangle \not\Downarrow \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \text{there is no } \langle V, s'\rangle \text{ for which } \langle M, s\rangle \Downarrow \langle V, s'\rangle \text{ holds.}$$

We leave the verification of these examples as simple exercises. (Prove the examples of $\not\Downarrow$ as in Example 3.2.1.)

5.3 Static semantics

As things stand, there are many LFP terms that do not evaluate to anything because of *type* mis-matches. For example, although the application of an integer to a function, such as $2\,(\lambda x.\, x)$, is a legal expression, it is not really a meaningful one. We can weed out such things by assigning types to LFP terms using a relation of the kind shown on Slide 45. The intended meaning of $\Gamma \vdash M : \tau$ is:

> "If the variable x has type $\Gamma(x)$ for each $x \in dom(\Gamma)$, then the term M has type τ."

We capture this intention through an inductive definition of the relation that follows the structure of the term M. The rules for function abstraction and application are shown on Slide 46 and the other axioms and rules in Figure 7. Note that these rules apply to *terms*, i.e. to expressions up to α-conversion. Thus

$$\dfrac{\dfrac{}{\{x' \mapsto \tau', x \mapsto \tau\} \vdash x : \tau}\ (:_{\mathrm{var}})}{\{x' \mapsto \tau'\} \vdash \lambda x'.\, x' : \tau \to \tau}\ (:_{\mathrm{fn}})$$

is a valid application of the rules, because $\lambda x'.\, x'$is the same term as $\lambda x.\, x$. In using the rules from the bottom up to deduce a type for a term M, it is as well to use a representative expression for M that has all its bound variables distinct from each other and from the variables in the domain of definition of the type environment. So for example $\emptyset \vdash \lambda x.\, \lambda x.\, x : \tau \to (\tau' \to \tau')$ holds, but it is probably easier to deduce this using the α-equivalent expression $\lambda x.\, \lambda x'.\, x'$.

Typing relation

takes the form $\boxed{\Gamma \vdash M : \tau}$ where:

- τ is a *type* $::=$ int integers
  - $\mid$ $bool$ booleans
  - $\mid$ loc location
  - $\mid$ cmd commands
  - $\mid$ $\tau \to \tau$ functions.
- Γ is a *type environment*, i.e. a finite partial function mapping variables to types.
- M is an LFP term.

Slide 45

$(:_{\mathrm{var}})$ $\quad \Gamma \vdash x : \tau \quad$ if $x \in dom(\Gamma)$ & $\Gamma(x) = \tau$

$(:_{\mathrm{int}})$ $\quad \Gamma \vdash n : int \quad (n \in \mathbb{Z})$

$(:_{\mathrm{bool}})$ $\quad \Gamma \vdash b : bool \quad (b \in \mathbb{B})$

$(:_{\mathrm{loc}})$ $\quad \Gamma \vdash \ell : loc \quad (\ell \in \mathbb{L})$

$(:_{\mathrm{iop}})$ $$\frac{\Gamma \vdash M_1 : int \quad \Gamma \vdash M_2 : int}{\Gamma \vdash M_1 \; iop \; M_2 : int}$$

$(:_{\mathrm{bop}})$ $$\frac{\Gamma \vdash M_1 : int \quad \Gamma \vdash M_2 : int}{\Gamma \vdash M_1 \; bop \; M_2 : bool}$$

$(:_{\mathrm{if}})$ $$\frac{\Gamma \vdash M_1 : bool \quad \Gamma \vdash M_2 : \tau \quad \Gamma \vdash M_3 : \tau}{\Gamma \vdash \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 : \tau}$$

$(:_{\mathrm{get}})$ $$\frac{\Gamma \vdash M : loc}{\Gamma \vdash {!M} : int}$$

$(:_{\mathrm{set}})$ $$\frac{\Gamma \vdash M_1 : loc \quad \Gamma \vdash M_2 : int}{\Gamma \vdash M_1 := M_2 : cmd}$$

$(:_{\mathrm{skip}})$ $\quad \Gamma \vdash \mathbf{skip} : cmd$

$(:_{\mathrm{seq}})$ $$\frac{\Gamma \vdash M_1 : cmd \quad \Gamma \vdash M_2 : cmd}{\Gamma \vdash M_1 \,;\, M_2 : cmd}$$

$(:_{\mathrm{whl}})$ $$\frac{\Gamma \vdash M_1 : bool \quad \Gamma \vdash M_2 : cmd}{\Gamma \vdash \mathbf{while}\ M_1\ \mathbf{do}\ M_2 : cmd}$$

plus rules $(:_{\mathrm{fn}})$ and $(:_{\mathrm{app}})$ on Slide 46.

Figure 7: Axioms and rules for LFP typing

Typing rules for function abstraction and application

$$(:_{\mathrm{fn}}) \qquad \frac{\Gamma[x \mapsto \tau] \vdash M : \tau'}{\Gamma \vdash \lambda x.\, M : \tau \to \tau'} \quad \text{if } x \notin dom(\Gamma)$$

$$(:_{\mathrm{app}}) \qquad \frac{\Gamma \vdash M_1 : \tau \to \tau' \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1\, M_2 : \tau'}$$

In rule $(:_{\mathrm{fn}})$, $\Gamma[x \mapsto \tau]$ denotes the type environment mapping x to τ and otherwise acting like Γ.

Slide 46

Definition 5.3.1 (Typeable closed terms). Given a closed LFP term M (i.e. one with no free variables), we say M *has type* τ and write

$$M : \tau$$

if $\emptyset \vdash M : \tau$ is derivable from the axioms and rules in Figure 7 (and Slide 46).

Note that an LFP term may have several different types—for example $\lambda x.\, x$ has type $\tau \to \tau$ for any τ. This is because we have not built any explicit type information into the syntax of expressions—an explicitly typed function abstraction would tag its bound variable with a type: $(\lambda x : \tau.\, M)$. For LFP, there is an algorithm which, given Γ and M, decides whether or not there exists a type τ satisfying $\Gamma \vdash M : \tau$. This is why this section is entitled *static* semantics: type checking is decidable and hence can be carried out at compile-time rather than at run-time. However, we will not pursue this topic of *type checking* here—see the Part II course on **Types**. Rather, we wish to indicate how types can be used to predict some aspects of the dynamic behaviour of terms. Slide 47 gives two examples of this. Both properties rely on the following substitution property of the typing relation.

Lemma 5.3.2. *If* $\Gamma \vdash M : \tau$ *and* $\Gamma[x \mapsto \tau] \vdash M' : \tau'$ *with* $x \notin dom(\Gamma)$, *then* $\Gamma \vdash M'[M/x] : \tau'$.

This can be proved by induction on the structure of M'; we omit the details.

Some type-related properties of evaluation in LFP

(i) **Subject reduction**. If $M : \tau$ and $\langle M, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle$, then $V : \tau$.

(ii) **Cbn-evaluation at non-command types is side-effect free**. If $\tau \neq cmd$, $M : \tau$, and $\langle M, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle$, then $s = s'$.

For $\Downarrow_{\mathrm{v}}$, property (i) holds, but property (ii) fails.

Slide 47

Property (i) on Slide 47 can be proved by Rule Induction for $\Downarrow_{\mathrm{n}}$ (and similarly for $\Downarrow_{\mathrm{v}}$). We leave the details as an exercise and concentrate on

Proof of (ii) on Slide 47. Let $\Phi(M, s, V, s')$ be the property

$$\langle M, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle \ \ \& \ \ \forall \tau \neq cmd\,(M : \tau \Rightarrow s = s').$$

By Rule Induction, it suffices to show that $\Phi(M, s, V, s')$ is closed under the axioms and rules inductively defining $\Downarrow_{\mathrm{n}}$. This is all very straightforward except for the case of the rule for call-by-name application, ($\Downarrow_{\mathrm{cbn}}$) on Slide 43, which we examine in detail.

So we have to prove $\Phi(M_1\, M_2, s, V, s'')$ given

$$\Phi(M_1, s, \lambda x.\, M_1', s') \tag{28}$$

$$\Phi(M_1'[M_2/x], s', V, s''). \tag{29}$$

Certainly ($\Downarrow_{\mathrm{cbn}}$), (28) and (29) imply that $\langle M_1\, M_2, s\rangle \Downarrow_{\mathrm{n}} \langle V, s''\rangle$ holds. So we just have to show that if

$$\emptyset \vdash M_1\, M_2 : \tau \tag{30}$$

holds for some $\tau \neq cmd$, then $s = s''$. But (30) must have been deduced using typing rule ($:_{\mathrm{app}}$) and hence

$$\emptyset \vdash M_1 : \tau' \to \tau \tag{31}$$

$$\emptyset \vdash M_2 : \tau' \tag{32}$$

hold for some type τ'. Since $\tau' \to \tau$ is not equal to cmd, (28) and (31) imply that $s = s'$ by definition of Φ. Furthermore, by the Subject Reduction property (i) on Slide 47, (28) and (31) also imply that $\lambda x.\, M_1' : \tau' \to \tau$. This typing can only have been deduced by $(:_{\mathrm{fn}})$ from

$$\{x \mapsto \tau'\} \vdash M_1' : \tau. \tag{33}$$

Applying Lemma 5.3.2 to (32) and (33) yields $M_1'[M_2/x] : \tau$; and by assumption $\tau \neq cmd$. Hence by (29) $s' = s''$. Therefore $s = s' = s''$, as required. □

Remark 5.3.3. Property (ii) on Slide 47 fails for *call-by-value* LFP because in the call-by-value setting, sequential composition cannot be limited just to commands, as the following example shows. Consider

$$M_1 \,\mathbf{andthen}\, M_2 \quad \stackrel{\text{def}}{=} \quad (\lambda x.\, M_2)\, M_1 \qquad (\text{where } x \notin fv(M_2)).$$

We have

$$M_1 : \tau \;\&\; M_2 : \tau' \Rightarrow M_1 \,\mathbf{andthen}\, M_2 : \tau'$$
$$\langle M_1, s\rangle \Downarrow_{\mathrm{v}} \langle V_1, s'\rangle \;\&\; \langle M_2, s'\rangle \Downarrow_{\mathrm{v}} \langle V_2, s''\rangle \Rightarrow \langle M_1 \,\mathbf{andthen}\, M_2, s\rangle \Downarrow_{\mathrm{v}} \langle V_2, s''\rangle.$$

Thus for example $(\ell := !\ell + 1)\,\mathbf{andthen}\,1$ is an 'active' term of type int:

$$\langle (\ell := !\ell + 1)\,\mathbf{andthen}\,1, \{\ell \mapsto 0\}\rangle \Downarrow_{\mathrm{v}} \langle 1, \{\ell \mapsto 1\}\rangle.$$

5.4 Local recursive definitions

In this section we consider the operational semantics of various kinds of *local declaration*, concentrating on *lexically scoped* constructs, i.e. ones whose scopes can be decided purely from the syntax of the language, at compile time. The designers of Algol 60 (Naur and Woodger (editors) 1963) defined the concept of locality for program blocks in their language as follows (quoted from Tennent 1991, page 84).

> "Any identifier occurring within a block may through a suitable declaration be specified to be local to the block in question. This means (a) that that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block."

The modern view (initiated by Landin 1966) is that for lexically scoped constructs, such matters can be made mathematically precise via the notions of *bound variable*, *substitution* and *α-conversion* from the lambda calculus (see Section 5.1). For example, function abstraction and application in LFP can be combined to give *local definitions*, as shown on Slide 48.

Local definitions in LFP

$$\boxed{\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \ \stackrel{\text{def}}{=}\ (\lambda x.\, M_2)\, M_1}$$

Derived typing rule:

$$\frac{\Gamma \vdash M_1 : \tau \qquad \Gamma[x \mapsto \tau] \vdash M_2 : \tau'}{\Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau'}$$

Derived evaluation rule (call-by-name):

$$\frac{\langle M_2[M_1/x], s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle}{\langle \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle}$$

Slide 48

Note that $fv(\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2) = fv(M_1) \cup \{x' \in fv(M_2) \mid x' \neq x\}$ and that free occurrences of x in M_2 become bound in $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$. Slide 49 illustrates how locality is enforced via α-conversion.

Remark 5.4.1. (i) Given the definition of $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ on Slide 48, the typing and evaluation rules given on the slide are *derivable* from the rules for call-by-name LFP in the sense that

- if $\Gamma \vdash M_1 : \tau$ and $\Gamma[x \mapsto \tau] \vdash M_2 : \tau'$ are derivable from Figure 7, then so is $\Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau'$;
- if $\langle M_2[M_1/x], s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle$, then $\langle \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle$.

Remember that we only defined evaluation for *closed* LFP terms. So in the evaluation rule M_1 is a closed term and M_2 contains at most x free.

(ii) For *call-by-value* evaluation of a local definition, rule ($\Downarrow_{\mathrm{cbv}}$) on Slide 43 means that we first compute the value of M_1 (if any) and use that as the local definition of x in evaluating M_2. So

- if $\langle M_1, s\rangle \Downarrow_{\mathrm{v}} \langle V_1, s'\rangle$ and $\langle M_2[V_1/x], s'\rangle \Downarrow_{\mathrm{v}} \langle V_2, s''\rangle$, then $\langle \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2, s\rangle \Downarrow_{\mathrm{v}} \langle V_2, s''\rangle$.

Locality via α-conversion

Because we identify LFP expressions up to α-conversion, the particular name of a bound variable is immaterial:

> $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ and $\mathbf{let}\ x' = M_1\ \mathbf{in}\ M_2[x'/x]$
> represent the *same* LFP term.

Moreover, up to α-conversion, a bound variable is always distinct from any variable in the surrounding context. For example:

$$(\mathbf{let}\ x = 1\ \mathbf{in}\ x + x) * x \equiv_\alpha (\mathbf{let}\ x' = 1\ \mathbf{in}\ x' + x') * x.$$

Slide 49

Note that the definition $x = M_1$ that occurs in $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ is a 'direct' definition—x is being declared as a local abbreviation for M_1 in M_2. By contrast, a *recursive* definition such as

$$f = \lambda x.\ \begin{array}{l}\mathbf{if}\ x < 0\ \mathbf{then}\ f\,x \\ \mathbf{else\ if}\ x = 0\ \mathbf{then}\ 1 \\ \mathbf{else}\ x * f(x-1)\end{array} \tag{34}$$

in which the variable f occurs (freely) in the right-hand side, has an altogether more complicated intended meaning: f is supposed to be some data (a function in this case) that satisfies the equation (34). What does this really mean? To give a denotational semantics (cf. Slide 3) requires one to model data as suitable mathematical structures for which 'fixed point equations' such as (34) always have solutions; and to do this in full generality requires some fascinating mathematics that, alas, is not part of this course. The operational reading of (34) is the *unfolding rule*:

> "During evaluation of an expression in the scope of the definition (34), whenever a use of f is encountered, use the right-hand side of the equation (thereby possibly introducing further uses of f) and continue evaluating."

In order to formulate this precisely, let us introduce an extension LFP with local recursive definitions, called LFP^+. The expressions of LFP^+ are given by the grammar for LFP (Slide 37) augmented by the $\mathbf{letrec}$ construct shown on Slide 50. Free occurrences of x

in M_1 and in M_2 become bound in $\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2$ and the extension to LFP^+ of the definition of substitution given in Figure 5 is:

$$(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2)[\sigma] \quad \stackrel{\text{def}}{=} \quad \mathbf{letrec}\ x' = M_1[\sigma[x \mapsto x']]\ \mathbf{in}\ M_2[\sigma[x \mapsto x']]$$

where x' is the first variable not in $fv(\sigma) \cup fv(M_1) \cup fv(M_2)$. We continue with the convention on Slide 41 and refer to $\equiv_\alpha$-equivalence classes as LFP^+ *terms*. Of course the α-conversion relation has to be suitably extended to cope with **letrec** expressions, by adding the axiom

$$(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2) \equiv_\alpha (\mathbf{letrec}\ x' = M_1[x'/x]\ \mathbf{in}\ M_2[x'/x])$$

and the rule

$$\frac{M_1 \equiv_\alpha M_1' \quad M_2 \equiv_\alpha M_2'}{(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2) \equiv_\alpha (\mathbf{letrec}\ x = M_1'\ \mathbf{in}\ M_2')}\ .$$

LFP^+ = LFP + local recursive definitions

Expressions:

$$M ::= \cdots \mid \boxed{\mathbf{letrec}\ x = M\ \mathbf{in}\ M}$$

Free variables:

$$fv(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2) \stackrel{\text{def}}{=} \{x' \in fv(M_1) \cup fv(M_2) \mid x' \neq x\}$$

Typing:

$$(:_{\text{letrec}}) \qquad \frac{\Gamma[x \mapsto \tau] \vdash M_1 : \tau \quad \Gamma[x \mapsto \tau] \vdash M_2 : \tau'}{\Gamma \vdash \mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2 : \tau'}$$

Slide 50

LFP^+ evaluation relation

is given by the evaluation rules for call-by-name LFP plus:

$$(\Downarrow_{\mathrm{letrec}}) \quad \frac{\langle M_2[(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_1)/x], s\rangle \Downarrow \langle V, s'\rangle}{\langle \mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2, s\rangle \Downarrow \langle V, s'\rangle}$$

Slide 51

The static semantics of LFP^+ is given by the typing axioms and rules for LFP (Figure 7) together with the rule $(:_{\mathrm{letrec}})$ on Slide 50. The LFP^+ evaluation relation is inductively defined by the axioms and rules for call-by-name LFP augmented by the rule $(\Downarrow_{\mathrm{letrec}})$ on Slide 51; we will continue to denote it by $\Downarrow_{\mathrm{n}}$. Note the similarity with the call-by-name evaluation of non-recursive $\mathbf{let}$-expressions (Slide 48). The difference is that when M_1 is substituted for x in M_2, it is surrounded by the recursive definition of x.

Fixpoint terms

$$\boxed{\mathbf{fix}\, x \,.\, M \quad \stackrel{\text{def}}{=} \quad \mathbf{letrec}\ x = M\ \mathbf{in}\ M}$$

Derived typing rule:

$$\frac{\Gamma[x \mapsto \tau] \vdash M : \tau}{\Gamma \vdash \mathbf{fix}\, x \,.\, M : \tau}$$

Derived evaluation rule (call-by-name):

$$\frac{\langle M[\mathbf{fix}\, x \,.\, M/x], s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle}{\langle \mathbf{fix}\, x \,.\, M, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle}$$

Slide 52

Slide 52 shows the specialisation of the **letrec** construct to yield *fixpoint* terms. The typing and evaluation properties stated on the slide are direct consequences of the rules ($:_{\mathrm{letrec}}$) and ($\Downarrow_{\mathrm{letrec}}$). We make use of such terms in the following example.

Example 5.4.2.

(35) $\quad \langle \mathbf{letrec}\ f = (\lambda x.\, \mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x * f(x-1))\ \mathbf{in}\ f\, 1, s\rangle \Downarrow_{\mathrm{n}} \langle 1, s\rangle.$

Proof. Define

$$F \quad \stackrel{\text{def}}{=} \quad \mathbf{fix}\, f \,.\, \lambda x.\, M \quad \stackrel{\text{def}}{=} \quad \mathbf{letrec}\ f = \lambda x.\, M\ \mathbf{in}\ \lambda x.\, M$$

where

$$M \quad \stackrel{\text{def}}{=} \quad \mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x * f(x-1).$$

For any closed term N and value V we have:

$$\dfrac{\dfrac{\dfrac{}{(\lambda x.\, M)[F/f] \Downarrow \lambda x.\, M[F/f]}\,(\Downarrow_{\mathrm{val}})}{F \Downarrow \lambda x.\, M[F/f]}\,(\Downarrow_{\mathrm{letrec}}) \qquad \begin{array}{c}\vdots\\ M[F/f][N/x] \Downarrow V\end{array}}{\dfrac{F\, N \Downarrow V}{\mathbf{letrec}\ f = \lambda x.\, M\ \mathbf{in}\ f\, N \Downarrow V}\,(\Downarrow_{\mathrm{letrec}})}\,(\Downarrow_{\mathrm{cbn}})$$

where we have suppressed mention of the state part of configurations because it plays no part in this proof. Taking $N = V = 1$, we see that to prove (35), it suffices to prove $M[F/f][1/x] \Downarrow_{\mathrm{n}} 1$. But since $M[F/f][1/x] = \mathbf{if}\ 1 = 0\ \mathbf{then}\ 1\ \mathbf{else}\ 1 * F(1-1)$, for this it clearly suffices to prove that $F(1-1) \Downarrow_{\mathrm{n}} 1$. Taking $N = 1 - 1$ and $V = 1$ in the proof fragment shown above, we have that $F(1-1) \Downarrow_{\mathrm{n}} 1$ if $M[F/f][1-1/x] \Downarrow_{\mathrm{n}} 1$. But $M[F/f][1-1/x] = \mathbf{if}\ (1-1) = 0\ \mathbf{then}\ 1\ \mathbf{else}\ (1-1) * F((1-1)-1)$ and:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{1 \Downarrow 1}\,(\Downarrow_{\mathrm{val}}) \quad \dfrac{}{1 \Downarrow 1}\,(\Downarrow_{\mathrm{val}})}{1 - 1 \Downarrow 0}\,(\Downarrow_{\mathrm{op}}) \quad \dfrac{}{0 \Downarrow 0}\,(\Downarrow_{\mathrm{val}})}{(1-1) = 0 \Downarrow \mathbf{true}}\,(\Downarrow_{\mathrm{op}}) \quad \dfrac{}{1 \Downarrow 1}\,(\Downarrow_{\mathrm{val}})}{\mathbf{if}\ (1-1) = 0\ \mathbf{then}\ 1\ \mathbf{else}\ (1-1) * F((1-1)-1) \Downarrow 1}\,(\Downarrow_{\mathrm{if1}}).$$

□

5.5 Exercises

Exercise 5.5.1. Consider the following LFP term for testing equality of location names in call-by-value LFP, where '$\mathbf{let}\ x = -\ \mathbf{in}\ -$' is as on Slide 48 and '$\mathbf{andthen}$' is as in Remark 5.3.3.

$$\begin{aligned} eq \stackrel{\text{def}}{=}\ & \lambda x_1.\,\lambda x_2.\,\mathbf{let}\ x = {!x_1}\ \mathbf{in} \\ & \quad x_1 := {!x_2} + 1\ \mathbf{andthen} \\ & \quad \mathbf{if}\ {!x_1} = {!x_2}\ \mathbf{then}\ (x_1 := x\ \mathbf{andthen}\ \mathbf{true}) \\ & \quad\quad \mathbf{else}\ (x_1 := x\ \mathbf{andthen}\ \mathbf{false}) \end{aligned}$$

Show that

$$\emptyset \vdash eq : loc \to (loc \to bool)$$

and that for all states s and all $\ell, \ell' \in dom(s)$

$$\langle eq\,\ell\,\ell', s\rangle \Downarrow_{\mathrm{v}} \langle b, s\rangle \quad \text{where } b = \begin{cases} \mathbf{true} & \text{if } \ell = \ell' \\ \mathbf{false} & \text{if } \ell \neq \ell'. \end{cases}$$

Exercise 5.5.2. What is wrong with the following suggestion?

"The rule $(\Downarrow_{\mathrm{letrec}})$ on Slide 51 can be simplified to

$$\frac{\langle M_2[(\mathbf{letrec}\ x = M_1\ \mathbf{in}\ x)/x], s\rangle \Downarrow \langle V, s'\rangle}{\langle \mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_2, s\rangle \Downarrow \langle V, s'\rangle}$$

because in the body of the $\mathbf{letrec}$-expression, x is defined to be M_1 so we can use $\mathbf{letrec}\ x = M_1\ \mathbf{in}\ x$ instead of $\mathbf{letrec}\ x = M_1\ \mathbf{in}\ M_1$."

[Hint: consider $\mathbf{letrec}\ x = 0\ \mathbf{in}\ x$.]

Exercise 5.5.3. Prove (i) on Slide 47 by Rule Induction: show that the property $\Phi(M, s, V, s')$ defined by

$$\langle M, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle \ \&\ \forall \tau.\, M : \tau \Rightarrow V : \tau$$

is closed under the axioms and rules inductively defining $\Downarrow_{\mathrm{n}}$. (For closure under rule ($\Downarrow_{\mathrm{cbn}}$) you will need Lemma 5.3.2. If you are really keen, try proving that, by induction on the structure of M'.)

Exercise* 5.5.4. *This exercise shows that* simultaneous *recursive definitions*

$$\begin{cases} x_1 = M_1(x_1, x_2) \\ x_2 = M_2(x_1, x_2) \end{cases} \tag{36}$$

can be encoded using $\mathbf{fix}$*-expressions.*

Let M_1, M_2 be LFP^+ terms containing at most variables $x_1 \neq x_2$ free. We say that a pair of closed terms X_1, X_2 is a solution of (36) if for $i = 1, 2$ we have

$$\langle M_i[X_1/x_1, X_2/x_2], s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle \Leftrightarrow \langle X_i, s\rangle \Downarrow_{\mathrm{n}} \langle V, s'\rangle$$

for all values V and states s, s'. Show how to construct such closed terms X_1, X_2 using the fixpoint construct of Slide 52.

6 Interaction

So far in this course we have looked at programming language constructs that are oriented towards computation of final results from some initial data. In this section we consider some that involve *interaction* between a program and its environment. We will look at a simple form of interactive input/output, and at inter-process communication via synchronised message passing.

Labelled transition systems defined

A *labelled transition system* is specified by

- a set Config and a set Act,
- a distinguished element $\tau \in \mathit{Act}$
- a ternary relation $\rightarrow \subseteq \mathit{Config} \times \mathit{Act} \times \mathit{Config}$.

The elements of Config are often called *configurations* (or 'states') and the elements of Act called *actions*. The ternary relation is written infix, i.e.

$$c \xrightarrow{\alpha} c'$$

means c, α, and c' are related by $\rightarrow$.

Slide 53

To specify the operational semantics of such constructs we have to be concerned with what happens along the way to termination as well as with final results; indeed, proper termination may not even enter into the semantic description of some constructs. So it is no surprise that transition relations between intermediate configurations (rather than evaluation relations between configurations and terminal configurations) will figure prominently. In order to describe the interactions that can happen at each transition step, we extend the notion of transition system (cf. Slide 4) by labelling the transitions with *actions* describing the nature of the interaction. The abstract notion of *labelled transition system* is given on Slide 53. What sets of configurations and actions to take is dictated by the particular programming language feature(s) being described. However, we will always include a distinguished action, τ, to label transition steps in which no external interaction takes place.[1] Thus the ordinary

[1]The insistence on the presence of a τ-action is slightly non-standard: in the literature a 'labelled transition system' is often taken to mean just a set of configurations, a set of actions, and a relation on (configuration, action, configuration)-triples

transition systems of Slide 4 can be regarded as instances of labelled transition systems by taking $Act = \{\tau\}$ and identifying transitions, $c \to c'$, with τ-labelled transitions, $c \xrightarrow{\tau} c'$.

6.1 Input/output

As a first example, we consider the language $\mathsf{LC}^{\mathrm{io}}$, obtained by adding to LC (cf. Section 3.1) facilities for reading integers from a single input stream and writing them to a single output stream. Its syntax is shown on Slide 54, where as usual ℓ ranges over some fixed set $\mathbb{L}$ of locations, n ranges over the integers, $\mathbb{Z}$, and b over the booleans, $\mathbb{B}$. We specify the operational semantics of $\mathsf{LC}^{\mathrm{io}}$ as a labelled transition system, where

- configurations are pairs $\langle P, s\rangle$ consisting of an $\mathsf{LC}^{\mathrm{io}}$ phrase P and a state s; as before, states are finite partial functions from $\mathbb{L}$ to $\mathbb{Z}$;
- actions are generated by the grammar on Slide 54;
- labelled transitions are inductively generated by the axioms and rules in Figure 8 and on Slide 55.

$\mathsf{LC}^{\mathrm{io}}$ — LC + input/output

Phrases: $P ::= C \mid E \mid B$

Commands:

$$\begin{array}{rcl} C & ::= & \mathbf{skip} \mid \ell := E \mid C \mathbin{;} C \mid \boxed{\mathbf{put}(E)} \\ & \mid & \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ B\ \mathbf{do}\ C \end{array}$$

Integer expressions: $E ::= n \mid {!\ell} \mid E\ iop\ E \mid \boxed{\mathbf{get}}$

Boolean expressions: $B ::= b \mid E\ bop\ E$

Actions: $\alpha ::= \tau \mid \mathrm{get}(n) \mid \mathrm{put}(n)$

Slide 54

Labelled transitions for get and put

$$(\xrightarrow{\text{get}}) \qquad \langle \mathbf{get}, s\rangle \xrightarrow{\text{get}(n)} \langle n, s\rangle$$

$$(\xrightarrow{\text{put1}}) \qquad \frac{\langle E, s\rangle \xrightarrow{\alpha} \langle E', s'\rangle}{\langle \mathbf{put}(E), s\rangle \xrightarrow{\alpha} \langle \mathbf{put}(E'), s'\rangle}$$

$$(\xrightarrow{\text{put2}}) \qquad \langle \mathbf{put}(n), s\rangle \xrightarrow{\text{put}(n)} \langle \mathbf{skip}, s\rangle$$

Slide 55

Note how the axioms for τ-transitions in LC^{io} correspond to the basic steps of computation in LC, but that the rules concerned with evaluating a subphrase of a phrase must deal with any type of action. As for LC, only transitions between commands can affect the state, but now transitions between any type of phrase can have input/output effects. An example sequence of labelled transitions is shown on Slide 56. We leave proving the validity of each transition as an exercise.

Remark 6.1.1. The simple notion of *determinacy* we used for transition systems (cf. Definition 1.1.1(iv)) has to be elaborated a bit for labelled transition systems. Clearly transitions from a given LC^{io} configuration are not uniquely determined: for example, $\langle \mathbf{get}, s\rangle$ can do a $\text{get}(n)$-action for any n. However, LC^{io} is deterministic in the sense that for each action α one has

$$\langle P, s\rangle \xrightarrow{\alpha} \langle P', s'\rangle \; \& \; \langle P, s\rangle \xrightarrow{\alpha} \langle P'', s''\rangle \quad \Rightarrow \quad P' = P'' \; \& \; s' = s''.$$

This can be proved using Rule Induction, along the same lines as the proof of determinacy for LC given in Section 3.1.

$(\xrightarrow{\text{loc}})$ $\langle !\ell, s\rangle \xrightarrow{\tau} \langle n, s\rangle$ if $\ell \in dom(s)$ & $s(\ell) = n$

$(\xrightarrow{\text{op1}})$ $$\frac{\langle E_1, s\rangle \xrightarrow{\alpha} \langle E_1', s'\rangle}{\langle E_1\ op\ E_2, s\rangle \xrightarrow{\alpha} \langle E_1'\ op\ E_2, s'\rangle}$$

$(\xrightarrow{\text{op2}})$ $$\frac{\langle E_2, s\rangle \xrightarrow{\alpha} \langle E_2', s'\rangle}{\langle n_1\ op\ E_2, s\rangle \xrightarrow{\alpha} \langle n_1\ op\ E_2', s'\rangle}$$

$(\xrightarrow{\text{op3}})$ $\langle n_1\ op\ n_2, s\rangle \xrightarrow{\tau} \langle c, s\rangle$ if $c = n_1\ iop\ n_2$

$(\xrightarrow{\text{set1}})$ $$\frac{\langle E, s\rangle \xrightarrow{\alpha} \langle E', s'\rangle}{\langle \ell := E, s\rangle \xrightarrow{\alpha} \langle \ell := E', s'\rangle}$$

$(\xrightarrow{\text{set2}})$ $\langle \ell := n, s\rangle \xrightarrow{\tau} \langle \mathbf{skip}, s[\ell \mapsto n]\rangle$

$(\xrightarrow{\text{seq1}})$ $$\frac{\langle C_1, s\rangle \xrightarrow{\alpha} \langle C_1', s'\rangle}{\langle C_1\ ;\ C_2, s\rangle \xrightarrow{\alpha} \langle C_1'\ ;\ C_2, s'\rangle}$$

$(\xrightarrow{\text{seq2}})$ $\langle \mathbf{skip}\ ;\ C, s\rangle \xrightarrow{\tau} \langle C, s\rangle$

$(\xrightarrow{\text{if1}})$ $$\frac{\langle B, s\rangle \xrightarrow{\alpha} \langle B', s'\rangle}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \xrightarrow{\alpha} \langle \mathbf{if}\ B'\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s'\rangle}$$

$(\xrightarrow{\text{if2}})$ $\langle \mathbf{if\ true\ then}\ C_1\ \mathbf{else}\ C_2, s\rangle \xrightarrow{\tau} \langle C_1, s\rangle$

$(\xrightarrow{\text{if3}})$ $\langle \mathbf{if\ false\ then}\ C_1\ \mathbf{else}\ C_2, s\rangle \xrightarrow{\tau} \langle C_2, s\rangle$

$(\xrightarrow{\text{whl}})$ $\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \xrightarrow{\tau} \langle \mathbf{if}\ B\ \mathbf{then}\ (C\ ;\ \mathbf{while}\ B\ \mathbf{do}\ C)\ \mathbf{else\ skip}, s\rangle$

plus $(\xrightarrow{\text{get}})$, $(\xrightarrow{\text{put1}})$, and $(\xrightarrow{\text{put2}})$ on Slide 55.

Figure 8: Axioms and rules for $\mathrm{LC}^{\mathrm{io}}$ labelled transitions

$C \stackrel{\text{def}}{=} \mathbf{while\ true\ do}\ C'$ where $C' \stackrel{\text{def}}{=} \ell := \mathbf{get}\ ; \mathbf{put}(\mathbf{get} - !\ell)$

$$
\begin{aligned}
\langle C, s\rangle &\xrightarrow{\tau} \cdot \xrightarrow{\tau} \langle C' ; C, s\rangle \\
&\xrightarrow{\text{get}(2)} \langle (\ell := 2 ; \mathbf{put}(\mathbf{get} - !\ell)) ; C, s\rangle \\
&\xrightarrow{\tau} \cdot \xrightarrow{\tau} \langle \mathbf{put}(\mathbf{get} - !\ell) ; C, s[\ell \mapsto 2]\rangle \\
&\xrightarrow{\text{get}(3)} \langle \mathbf{put}(3 - !\ell) ; C, s[\ell \mapsto 2]\rangle \\
&\xrightarrow{\tau} \cdot \xrightarrow{\tau} \langle \mathbf{put}(1) ; C, s[\ell \mapsto 2]\rangle \\
&\xrightarrow{\text{put}(1)} \langle \mathbf{skip} ; C, s[\ell \mapsto 2]\rangle \\
&\xrightarrow{\tau} \langle C, s[\ell \mapsto 2]\rangle \\
&\dots
\end{aligned}
$$

Slide 56

6.2 Bisimilarity

The notion of *semantic equivalence* considered in Section 4 is phrased in terms of observing final results of evaluating programs. For languages with interactive features this is not so appropriate, since interactive programs may never produce a final result. In this case it is more appropriate to formulate a notion of program equivalence based upon the (sequences of) actions programs can perform as they evolve. We present one such notion in this section.

Recall from Slide 53 the notion of labelled transition system that we used when specifying the operational semantics of languages involving interaction. We can associate with each labelled transition system the binary relation of *bisimilarity* on its set of configurations, as defined on Slide 57. The notation $\xrightarrow{\hat{\alpha}}$ used there is defined as follows:

$$
\xrightarrow{\hat{\alpha}} \;\stackrel{\text{def}}{=}\; \begin{cases} \xrightarrow{\tau^*} & \text{if } \alpha = \tau \\ \xrightarrow{\tau^*} \cdot \xrightarrow{\alpha} \cdot \xrightarrow{\tau^*} & \text{if } \alpha \neq \tau \end{cases}
$$

where $\xrightarrow{\tau^*}$ denotes the reflexive-transitive closure of the $\xrightarrow{\tau}$ relation (cf. Definition 1.1.1(i)).

Bisimulations and bisimilarity

Let $(\mathit{Config}, \mathit{Act}, \tau, \rightarrow)$ be a labelled transition system (cf. Slide 53). A *bisimulation* is a binary relation $\mathcal{R}$ on the set Config such that whenever $c_1 \;\mathcal{R}\; c_2$, then for all $\alpha \in \mathit{Act}$

- whenever $c_1 \xrightarrow{\alpha} c_1'$, then $c_2 \xrightarrow{\hat{\alpha}} c_2'$ holds for some c_2' with $c_1' \;\mathcal{R}\; c_2'$; and
- whenever $c_2 \xrightarrow{\alpha} c_2'$, then $c_1 \xrightarrow{\hat{\alpha}} c_1'$ holds for some c_1' with $c_1' \;\mathcal{R}\; c_2'$.

By definition, two configurations are *bisimilar*, $c_1 \approx c_2$, if and only if $c_1 \;\mathcal{R}\; c_2$ holds for some bisimulation relation $\mathcal{R}$.

Slide 57

So $\mathcal{R}$ is a bisimulation on Config if whenever $c_1 \;\mathcal{R}\; c_2$, then the following four conditions hold:

- If $c_1 \xrightarrow{\tau} c_1'$, then $c_2 \xrightarrow{\tau^*} c_2'$ holds for some c_2' with $c_1' \;\mathcal{R}\; c_2'$.
- If $c_2 \xrightarrow{\tau} c_2'$, then $c_1 \xrightarrow{\tau^*} c_1'$ holds for some c_1' with $c_1' \;\mathcal{R}\; c_2'$.
- For any action $\alpha \neq \tau$, if $c_1 \xrightarrow{\alpha} c_1'$, then $c_2 \xrightarrow{\tau^* \alpha \tau^*} c_2'$ holds for some c_2' with $c_1' \;\mathcal{R}\; c_2'$.
- For any action $\alpha \neq \tau$, if $c_2 \xrightarrow{\alpha} c_2'$, then $c_1 \xrightarrow{\tau^* \alpha \tau^*} c_1'$ holds for some c_1' with $c_1' \;\mathcal{R}\; c_2'$.

Here is an example of a bisimilarity for the language $\mathsf{LC}^{\mathrm{io}}$ of Section 6.1.

Example

Consider the following $\mathsf{LC}^{\mathrm{io}}$ commands:

$$C_1 \stackrel{\text{def}}{=} \textbf{while true do } (\ell := \textbf{get} \,;\, \textbf{put}(!\ell - \textbf{get}))$$

$$C_2 \stackrel{\text{def}}{=} \textbf{while true do put}(\textbf{get} - \textbf{get})$$

$$C_3 \stackrel{\text{def}}{=} \textbf{while true do } (\ell := \textbf{get} \,;\, \textbf{put}(\textbf{get} - !\ell)).$$

Then for any state s it is the case that

$$\langle C_1, s\rangle \approx \langle C_2, s\rangle \not\approx \langle C_3, s\rangle.$$

Slide 58

Example 6.2.1. Consider the three $\mathsf{LC}^{\mathrm{io}}$ commands C_1, C_2, C_3 on Slide 58. The essentially deterministic nature of the $\mathsf{LC}^{\mathrm{io}}$ labelled transition system (cf. Remark 6.1.1) makes it quite simple to establish the bisimilarity and non-bisimilarity given on the slide. For any state s, the only transitions from $\langle C_1, s\rangle$ and $\langle C_2, s\rangle$ deducible using the rules for $\mathsf{LC}^{\mathrm{io}}$ labelled transitions in Figure 8 are those given in Figure 9. Consequently it is not hard (just tedious) to check that the relation in Figure 10 is a bisimulation establishing the fact that $\langle C_1, s\rangle \approx \langle C_2, s\rangle$.

To show that $\langle C_2, s\rangle \not\approx \langle C_3, s\rangle$ we suppose the contrary and derive a contradiction. So suppose $\langle C_2, s\rangle$ and $\langle C_3, s\rangle$ are related by some bisimulation $\mathcal{R}$. Since

$$\langle C_3, s\rangle \xrightarrow{\tau^*\mathrm{get}(2)\tau^*\mathrm{get}(3)\tau^*} \langle \textbf{put}(1) \,;\, C_3, s[\ell \mapsto 2]\rangle$$

(cf. Slide 56) and $\langle C_2, s\rangle \;\mathcal{R}\; \langle C_3, s\rangle$, there must be some $\langle C', s'\rangle$ with

$$\langle C_2, s\rangle \xrightarrow{\tau^*\mathrm{get}(2)\tau^*\mathrm{get}(3)\tau^*} \langle C', s'\rangle \tag{37}$$

and

$$\langle C', s'\rangle \;\mathcal{R}\; \langle \textbf{put}(1) \,;\, C_3, s[\ell \mapsto 2]\rangle. \tag{38}$$

In view of Figure 9, the only way that (37) can hold is if $\langle C', s'\rangle$ is either $\langle \textbf{put}(2-3) \,;\, C_2, s\rangle$ or $\langle \textbf{put}(-1) \,;\, C_2, s\rangle$. In either case we have

$$\langle C', s'\rangle \xrightarrow{\tau^*\mathrm{put}(-1)}$$

For any $n, n' \in \mathbb{Z}$

$$\begin{aligned}
\langle C_1, s\rangle &\xrightarrow{\tau} \langle \textbf{if true then } (\ell := \textbf{get} \,;\, \textbf{put}(!\ell - \textbf{get}))\,;\, C_1 \textbf{ else skip}, s\rangle \\
&\xrightarrow{\tau} \langle (\ell := \textbf{get} \,;\, \textbf{put}(!\ell - \textbf{get}))\,;\, C_1, s\rangle \\
&\xrightarrow{\mathrm{get}(n)} \langle (\ell := n \,;\, \textbf{put}(!\ell - \textbf{get}))\,;\, C_1, s\rangle \\
&\xrightarrow{\tau} \langle (\textbf{skip} \,;\, \textbf{put}(!\ell - \textbf{get}))\,;\, C_1, s[\ell \mapsto n]\rangle \\
&\xrightarrow{\tau} \langle \textbf{put}(!\ell - \textbf{get})\,;\, C_1, s[\ell \mapsto n]\rangle \\
&\xrightarrow{\tau} \langle \textbf{put}(n - \textbf{get})\,;\, C_1, s[\ell \mapsto n]\rangle \\
&\xrightarrow{\mathrm{get}(n')} \langle \textbf{put}(n - n')\,;\, C_1, s[\ell \mapsto n]\rangle \\
&\xrightarrow{\tau} \langle \textbf{put}(n'')\,;\, C_1, s[\ell \mapsto n]\rangle \quad \text{where } n'' = n - n' \\
&\xrightarrow{\mathrm{put}(n'')} \langle \textbf{skip}\,;\, C_1, s[\ell \mapsto n]\rangle \\
&\xrightarrow{\tau} \langle C_1, s[\ell \mapsto n]\rangle \\
&\ldots
\end{aligned}$$

and

$$\begin{aligned}
\langle C_2, s\rangle &\xrightarrow{\tau} \langle \textbf{if true then put}(\textbf{get} - \textbf{get})\,;\, C_2 \textbf{ else skip}, s\rangle \\
&\xrightarrow{\tau} \langle \textbf{put}(\textbf{get} - \textbf{get})\,;\, C_2, s\rangle \\
&\xrightarrow{\mathrm{get}(n)} \langle \textbf{put}(n - \textbf{get})\,;\, C_2, s\rangle \\
&\xrightarrow{\mathrm{get}(n')} \langle \textbf{put}(n - n')\,;\, C_2, s\rangle \\
&\xrightarrow{\tau} \langle \textbf{put}(n'')\,;\, C_2, s\rangle \quad \text{where } n'' = n - n' \\
&\xrightarrow{\mathrm{put}(n'')} \langle \textbf{skip}\,;\, C_2, s\rangle \\
&\xrightarrow{\tau} \langle C_2, s\rangle \\
&\ldots
\end{aligned}$$

Figure 9: Transitions from $\langle C_1, s\rangle$ and $\langle C_2, s\rangle$

$$
\begin{aligned}
\mathcal{R} \stackrel{\text{def}}{=} \quad & \{(\langle C_1, s_1\rangle, \langle C_2, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\} \\
\cup\ & \{(\langle \textbf{if true then } (\ell := \textbf{get} \,;\, \textbf{put}(!\ell - \textbf{get}))\,;\, C_1 \textbf{ else skip}, s_1\rangle, \\
& \quad \langle \textbf{if true then put}(\textbf{get} - \textbf{get})\,;\, C_2 \textbf{ else skip}, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\} \\
\cup\ & \{(\langle (\ell := \textbf{get} \,;\, \textbf{put}(!\ell - \textbf{get}))\,;\, C_1, s_1\rangle, \\
& \quad \langle \textbf{put}(\textbf{get} - \textbf{get})\,;\, C_2, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\} \\
\cup\ & \{(\langle (\ell := n \,;\, \textbf{put}(!\ell - \textbf{get}))\,;\, C_1, s_1\rangle, \\
& \quad \langle \textbf{put}(n - \textbf{get})\,;\, C_2, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\ \&\ n \in \mathbb{Z}\} \\
\cup\ & \{(\langle (\textbf{skip} \,;\, \textbf{put}(!\ell - \textbf{get}))\,;\, C_1, s_1[\ell \mapsto n]\rangle, \\
& \quad \langle \textbf{put}(n - \textbf{get})\,;\, C_2, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\ \&\ n \in \mathbb{Z}\} \\
\cup\ & \{(\langle \textbf{put}(!\ell - \textbf{get})\,;\, C_1, s_1[\ell \mapsto n]\rangle, \\
& \quad \langle \textbf{put}(n - \textbf{get})\,;\, C_2, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\ \&\ n \in \mathbb{Z}\} \\
\cup\ & \{(\langle \textbf{put}(n - \textbf{get})\,;\, C_1, s_1[\ell \mapsto n]\rangle, \\
& \quad \langle \textbf{put}(n - \textbf{get})\,;\, C_2, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\ \&\ n \in \mathbb{Z}\} \\
\cup\ & \{(\langle \textbf{put}(n - n')\,;\, C_1, s_1[\ell \mapsto n]\rangle, \\
& \quad \langle \textbf{put}(n - n')\,;\, C_2, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\ \&\ n, n' \in \mathbb{Z}\} \\
\cup\ & \{(\langle \textbf{put}(n'')\,;\, C_1, s_1[\ell \mapsto n]\rangle, \\
& \quad \langle \textbf{put}(n'')\,;\, C_2, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\ \&\ n, n'' \in \mathbb{Z}\} \\
\cup\ & \{(\langle \textbf{skip}\,;\, C_1, s_1[\ell \mapsto n]\rangle, \langle \textbf{skip}\,;\, C_2, s_2\rangle) \mid s_1, s_2 \in \mathit{States}\ \&\ n \in \mathbb{Z}\}.
\end{aligned}
$$

Figure 10: A bisimulation relating $\langle C_1, s\rangle$ and $\langle C_2, s\rangle$

So in view of (38) we must also have

$$\langle \mathbf{put}(1)\,;C_3, s[\ell \mapsto 2]\rangle \xrightarrow{\tau^*\mathrm{put}(-1)\tau^*}$$

which clearly is impossible. Therefore no such bisimulation $\mathcal{R}$ can exist. □

Remark 6.2.2. With regard to the basic properties of equality listed on Slide 29, the bisimilarity relation associated with a labelled transition system is always reflexive, symmetric and transitive: see Exercise 6.4.6. It does not make sense to ask whether $\approx$ is a congruence unless the configurations of the labelled transition system have some syntactic structure—for example, are the configurations of some programming language. For $\mathsf{LC}^{\mathrm{io}}$, we have so far defined bisimilarity for configurations rather than for phrases. However, if we define $P \approx P'$ to mean that $\langle P, s\rangle \approx \langle P', s\rangle$ holds for all states s, then it is in fact the case that the $\mathsf{LC}^{\mathrm{io}}$ bisimilarity relation is a congruence.

6.3 Communicating processes

In $\mathsf{LC}^{\mathrm{io}}$ there is interaction between a program and its environment. In this section we consider the more complicated situation in which there is interaction between several concurrently executing processes within a system. Two common mechanisms for achieving such interaction are via shared state of various kinds and via various kinds of message-passing. Here we will look at a simple Language of Communicating Processes, LCP, based on the CCS calculus of Milner (1989).

LCP syntax

Process expressions:

$$P ::= c(x)\,.\,P \mid \bar{c}\langle E\rangle\,.\,P \mid \mathbf{0} \mid P + P \mid P|P \mid \nu\, c\,.\,P \mid A$$

Integer expressions: $E ::= x \mid n \mid E\ iop\ E$

Actions: $\alpha ::= \tau \mid \bar{c}\langle n\rangle \mid c(n)$

where

- $x \in \mathbb{V}$, a fixed, infinite set of *integer variables*;
- $c \in \mathbb{CV}$, a fixed, infinite set of *channel variables*;
- $n \in \mathbb{Z}$, the set of *integers*;
- $iop \in \mathbb{I}op \stackrel{\text{def}}{=} \{+, -, *, \dots\}$ a fixed, finite set of integer-valued binary operations on integers;
- $A \in \mathbb{PC}$, a fixed, infinite set of *process constants*.

Slide 59

The syntax of LCP is given on Slide 59. LCP process expressions represent multi-threaded computations which evolve by performing sequences of actions. Actions are of three types:

$\bar{c}\langle n\rangle$**:** *output* number n on channel named c;

$c(n)$**:** *input* number n on channel named c;

τ**:** *internal* (externally unobservable)action.

The intended meaning of the various forms of process expression are as follows.

Input prefix: $c(x) \,.\, P$ is a process ready to receive an integer value x on channel c and then behave like P. This is a binding construct: free occurrences of x in P are bound in $c(x) \,.\, P$.

Output prefix: $\bar{c}\langle E\rangle \,.\, P$ is a process ready to output the value of the integer expression E on channel c and then behave like P.

Inactive process: $\mathbf{0}$ can perform no actions.

Choice: $P + P'$ is a process which can do any of the actions of either P or P'.

Parallel composition: $P|P'$ can perform the actions of P and P' independently, and can perform synchronised communications between them on a common channel.

Restricted process: $\nu\, c \,.\, P$ can perform the actions of P except for those involving input/output on channel c. This is a binding construct: free occurrences of c in P are bound in $\nu\, c \,.\, P$.

Process constants: we assume that all the process constants A occurring in process expressions that we use have been given a (recursive) *definition*, as on Slide 60.

Recursive definitions of LCP process constants

take the form

$$\begin{cases} A_1 & \stackrel{\text{def}}{=} & P_1 \\ A_2 & \stackrel{\text{def}}{=} & P_2 \\ & \vdots & \\ A_n & \stackrel{\text{def}}{=} & P_n \end{cases}$$

where $A_1, A_2, \ldots, A_n \in \mathbb{PC}$ are mutually distinct and $P_1, P_2, \ldots, P_n$ are process expressions with no free integer variables and containing no process constants other than $A_1, A_2, \ldots, A_n$.

Slide 60

There are two kinds of variable in LCP—standing for integers and communication channels—which may occur both free and bound in process expressions, as indicated above. Correspondingly, there are two kinds of substitution:

- $P[E/x]$: substitute the integer expression E for all free occurrences of x in P
- $P[c'/c]$: substitute the channel variable c' for all free occurrences of c in P

These substitutions can be defined along the lines of Figure 5 (i.e. in a way that avoids capture of free variables by the binding operations). We will identify process expressions up to α-conversion of bound integer, channel, and function variables.

Given a recursive definition of some LCP process constants as on Slide 60, we define the operational semantics of LCP process expressions involving those constants by means of a labelled transition system. Its configurations are the LCP *processes*, which by definition are the process expressions with no free integer variables (but possibly with free channel variables) and with process constants drawn from the defined ones. Its set of actions is generated by the grammar on Slide 59 and its set of labelled transitions is inductively defined by the axioms and rules given on Slides 61–63.

Prefixes and choice

$(\xrightarrow{\text{in}})$ $$c(x) \,.\, P \xrightarrow{c(n)} P[n/x]$$

$(\xrightarrow{\text{out}})$ $$\bar{c}\langle E\rangle \,.\, P \xrightarrow{\bar{c}\langle n\rangle} P \quad \text{if } E \Downarrow n$$

$(\xrightarrow{+})$ $$\frac{P_i \xrightarrow{\alpha} P}{P_1 + P_2 \xrightarrow{\alpha} P} \quad (i = 1, 2)$$

where $\Downarrow$ is inductively defined by the following axiom and rules:

$$n \Downarrow n \quad (n \in \mathbb{Z}) \qquad \frac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{E_1 \; iop \; E_2 \Downarrow n} \quad \text{if } n = n_1 \; iop \; n_2$$

Slide 61

Parallel composition

Independent action:

$(\xrightarrow{\text{par}})$ $$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1|P_2 \xrightarrow{\alpha} P_1'|P_2} \quad \text{and symmetrically.}$$

Synchronised communication:

$(\xrightarrow{\text{com}})$ $$\frac{P_1 \xrightarrow{c(n)} P_1' \quad P_2 \xrightarrow{\bar{c}\langle n\rangle} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'} \quad \text{and symmetrically.}$$

Slide 62

Restriction and constants

$$(\xrightarrow{\nu}) \qquad \frac{P \xrightarrow{\alpha} P'}{\nu\, c \,.\, P \xrightarrow{\alpha} \nu\, c \,.\, P'} \quad \text{if } \alpha \neq c(n), \bar{c}\langle n\rangle \text{ (any } n\text{)}$$

$$(\xrightarrow{\text{con}}) \qquad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad \text{if } A \stackrel{\text{def}}{=} P$$

Slide 63

Transitions labelled $c(n)$ (respectively $\bar{c}\langle n\rangle$) occur when a process inputs (respectively outputs) the integer n on channel c; transitions labelled τ record the synchronised communication of integers on channels, *without* recording which channel or which integer (cf. rule $(\xrightarrow{\text{com}})$). Note that the rule $(\xrightarrow{\nu})$ for the restriction operator can prevent explicit input- or output-actions (typically those arising from a parallel composition via rule $(\xrightarrow{\text{par}})$), but can never prevent a τ-transition.

Remark 6.3.1. Note that unlike LC^{io}, the labelled transition system for LCP is non-deterministic in the sense that there may be several different transitions from a given process with a given action. The choice operator, $+$, clearly introduces such non-determinism, but it is present also because of the rules for parallel composition. For example

$$\mathbf{0} \mid \mathbf{0} \mid \bar{c}\langle 0\rangle \,.\, \mathbf{0} \xleftarrow{\tau} \bar{c}\langle 0\rangle \,.\, \mathbf{0} \mid c(x) \,.\, \mathbf{0} \mid \bar{c}\langle 0\rangle \,.\, \mathbf{0} \xrightarrow{\tau} \bar{c}\langle 0\rangle \,.\, \mathbf{0} \mid \mathbf{0} \mid \mathbf{0}.$$

Example 6.3.2. Here is an example illustrating the combined use of parallel composition and restriction for 'plugging' processes together—in this case a number of single-capacity buffers to get a finite-capacity buffer. Some simple LCP processes modelling buffers of various finite capacities are shown on Slide 64. Using $(\xrightarrow{\text{in}})$, $(\xrightarrow{\text{out}})$ and $(\xrightarrow{\text{con}})$, it is not hard to deduce that the possible transitions of $B_{i,o}$ are

$$\begin{tikzcd} B_{i,o} \arrow[r, bend left, "i(n)"] & \bar{o}\langle n\rangle \,.\, B_{i,o} \arrow[l, bend left, "\bar{o}\langle n\rangle"] \end{tikzcd} \qquad (n \in \mathbb{Z}).$$

Example LCP processes

Buffer of capacity one, inputting on channel i and outputting on channel o:

$$B_{i,o} \stackrel{\text{def}}{=} i(x) \,.\, \bar{o}\langle x\rangle \,.\, B_{io}$$

Buffers of capacity two, three, . . . :

$$B'_{i,o} \stackrel{\text{def}}{=} \nu\, c \,.\, (B_{i,c}|B_{c,o})$$
$$B''_{i,o} \stackrel{\text{def}}{=} \nu\, c \,.\, \nu\, c' \,.\, (B_{i,c}|B_{c,c'}|B_{c',o})$$
$$\ldots$$

Slide 64

$$\begin{tikzcd}
B_{i,c} \arrow[r, bend left, "i(n)"] & \bar{c}\langle n\rangle \,.\, B_{i,c} \arrow[l, bend left, "\bar{c}\langle n\rangle"]
\end{tikzcd}$$

$$\begin{tikzcd}
B_{c,o} \arrow[r, bend left, "c(n)"] & \bar{o}\langle n\rangle \,.\, B_{c,o} \arrow[l, bend left, "\bar{o}\langle n\rangle"]
\end{tikzcd}$$

Slide 65

It follows from the structural nature of the rules on Slides 61–63 that the only possible transitions from the process $B'_{i,o}$ defined on Slide 64 are all of the form

$$B'_{i,o} \xrightarrow{i(n)} \nu\, c\,.\,((\bar{c}\langle n\rangle\,.\,B_{i,c})|B_{c,o}) \qquad \text{(any } n \in \mathbb{Z}\text{).}$$

Similarly, the only possible transitions from the process $\nu\, c\,.\,((\bar{c}\langle n\rangle\,.\,B_{i,c})|B_{c,o})$ is

$$\nu\, c\,.\,((\bar{c}\langle n\rangle\,.\,B_{i,c})|B_{c,o}) \xrightarrow{\tau} \nu\, c\,.\,(B_{i,c}|(\bar{o}\langle n\rangle\,.\,B_{c,o}))$$

which is deduced via a proof ending

$$\dfrac{\dfrac{\begin{matrix}\vdots & & \vdots\\ \bar{c}\langle n\rangle\,.\,B_{i,c} \xrightarrow{\bar{c}\langle n\rangle} B_{i,c} & \quad & B_{c,o} \xrightarrow{c(n)} \bar{o}\langle n\rangle\,.\,B_{c,o}\end{matrix}}{(\bar{c}\langle n\rangle\,.\,B_{i,c})|B_{c,o} \xrightarrow{\tau} B_{i,c}|(\bar{o}\langle n\rangle\,.\,B_{c,o})}\;(\xrightarrow{\text{com}})}{\nu\, c\,.\,((\bar{c}\langle n\rangle\,.\,B_{i,c})|B_{c,o}) \xrightarrow{\tau} \nu\, c\,.\,(B_{i,c}|(\bar{o}\langle n\rangle\,.\,B_{c,o}))}\;(\xrightarrow{\nu})$$

The process $\nu\, c\,.\,(B_{i,c}|(\bar{o}\langle n\rangle\,.\,B_{c,o}))$ is capable of two types of action, both deduced via proofs whose penultimate rule is an instance of $(\xrightarrow{\text{par}})$:

$$\begin{aligned}
&\nu\, c\,.\,(B_{i,c}|(\bar{o}\langle n\rangle\,.\,B_{c,o})) \xrightarrow{\bar{o}\langle n\rangle} \nu\, c\,.\,(B_{i,c}|B_{c,o})\\
&\nu\, c\,.\,(B_{i,c}|(\bar{o}\langle n\rangle\,.\,B_{c,o})) \xrightarrow{i(n')} \nu\, c\,.\,((\bar{c}\langle n'\rangle\,.\,B_{i,c})|(\bar{o}\langle n\rangle\,.\,B_{c,o})).
\end{aligned}$$

Here is a diagram showing all the processes that can be reached from the process $\nu\, c\,.\,(B_{i,c}|B_{c,o})$ which defines $B'_{i,o}$:

$$\begin{tikzcd}
\nu\, c\,.\,(B_{i,c}|(\bar{o}\langle n'\rangle\,.\,B_{c,o})) \arrow[rr, "i(n)"] \arrow[drr, "\bar{o}\langle n'\rangle"'] & & \nu\, c\,.\,((\bar{c}\langle n\rangle\,.\,B_{i,c})|(\bar{o}\langle n'\rangle\,.\,B_{c,o})) \arrow[rr, "\bar{o}\langle n'\rangle"] & & \nu\, c\,.\,((\bar{c}\langle n\rangle\,.\,B_{i,c})|B_{c,o}) \arrow[dd, "\tau"] \\
& & \nu\, c\,.\,(B_{i,c}|B_{c,o}) \arrow[urr, "i(n)"'] \arrow[dll, "i(n')"] & & \\
\nu\, c\,.\,((\bar{c}\langle n'\rangle\,.\,B_{i,c})|B_{c,o}) \arrow[uu, "\tau"] & & \nu\, c\,.\,((\bar{c}\langle n'\rangle\,.\,B_{i,c})|(\bar{o}\langle n\rangle\,.\,B_{c,o})) \arrow[ll, "\bar{o}\langle n\rangle"] & & \nu\, c\,.\,(B_{i,c}|(\bar{o}\langle n\rangle\,.\,B_{c,o})). \arrow[ll, "i(n')"] \arrow[ull, "\bar{o}\langle n\rangle"]
\end{tikzcd}$$

In this diagram n and n' are arbitrary integers (possibly equal). Ignoring internal actions (τ), note that $\nu\, c\,.\,(B_{i,c}|B_{c,o})$ can input at most two integers before it has to output one of them; and integers are output in the same order that they are input. In this sense it models a capacity-two buffer.

6.4 Exercises

Exercise 6.4.1. Prove the validity of all the labelled transitions on Slide 56.

Exercise 6.4.2. Prove the determinacy property of $\mathsf{LC}^{\mathrm{io}}$ labelled transitions mentioned in Remark 6.1.1.

Exercise 6.4.3. Prove the validity of all the labelled transitions mentioned in Example 6.3.2.

Exercise 6.4.4. Consider the LCP process recursively defined by

$$\mathit{Buff} \stackrel{\text{def}}{=} i(x)\,.\,(\bar{o}\langle x\rangle\,.\,\mathit{Buff} \;+\; i(x')\,.\,\bar{o}\langle x\rangle\,.\,\bar{o}\langle x'\rangle\,.\,\mathit{Buff})$$

Calculate all the labelled transitions for processes that can be reached from Buff. Hence show that Buff is *not* bisimilar to the capacity-two buffer $B'_{i,o}$ given on Slide 64. [Hint: Buff can input two numbers and output the first to reach a state in which it *must* output the second before inputting a third number. This is not true of $B'_{i,o}$. Hence show there can be no bisimulation relation containing the pair $(\mathit{Buff}, B'_{i,o})$.]

Exercise 6.4.5. Consider generalising LCP process constants A by allowing them to carry integer arguments, $A(E_1, \ldots, E_n)$. So recursive definitions (Slide 60) now take the form

$$\begin{cases} A_1(x_{1,1}\ldots, x_{1,k_1}) & \stackrel{\text{def}}{=} & P_1 \\ & \vdots & \\ A_n(x_{n,1}\ldots, x_{n,k_n}) & \stackrel{\text{def}}{=} & P_n \end{cases}$$

where now each right-hand side is a process which can involve free occurrences of the integer variables mentioned in the argument of the left-hand side. Rule ($\xrightarrow{\text{con}}$) becomes

$$\frac{P[n_1/x_1, \ldots, n_k/x_k] \xrightarrow{\alpha} P'}{A(E_1, \ldots, E_k) \xrightarrow{\alpha} P'} \quad \text{if } A(x_1, \ldots, x_k) \stackrel{\text{def}}{=} P \text{ and } E_i \Downarrow n_i \text{ (for } i = 1, \ldots, k).$$

For this extension of LCP, consider the definition

$$\begin{cases} B_0 & \stackrel{\text{def}}{=} & i(x)\,.\,B_1(x) \\ B_1(x) & \stackrel{\text{def}}{=} & \bar{o}\langle x\rangle\,.\,B_0 + i(x')\,.\,B_2(x, x') \\ B_2(x, x') & \stackrel{\text{def}}{=} & \bar{o}\langle x\rangle\,.\,B_1(x') \end{cases}$$

Show that B_0 is bisimilar to the process $B'_{i,o}$ given on Slide 64.

Exercise 6.4.6. Suppose given a labelled transition system as on Slide 53 and consider the notions of bisimulation and bisimilarity for it, as defined on Slide 57.

(i) Prove that the identity relation

$$\mathcal{I}d \stackrel{\text{def}}{=} \{(c, c) \mid c \in \mathit{Config}\}$$

is a bisimulation .

(ii) Show that if $\mathcal{R}$ and $\mathcal{R}'$ are bisimulations, then so is their composition

$$\mathcal{R} \circ \mathcal{R}' \stackrel{\text{def}}{=} \{(c_1, c_3) \mid \exists c_2\ (c_1\ \mathcal{R}\ c_2\ \&\ c_2\ \mathcal{R}'\ c_3)\}.$$

(iii) Show that if $\mathcal{R}$ is a bisimulation then so is its reciprocal

$$\mathcal{R}^{-1} \stackrel{\text{def}}{=} \{(c_2, c_1) \mid c_1\ \mathcal{R}\ c_2\}.$$

(iv) Deduce from (i)–(iii) that $\approx$ is an equivalence relation.

References

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics* (revised ed.). North-Holland.

Gunter, C. A. (1992). *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press.

Hennessy, M. (1990). *The Semantics of Programming Languages. An Elementary Introduction using Structural Operational Semantics.* John Wiley & Sons Ltd.

Kahn, G. (1987). Natural semantics. Rapport de Recherche 601, INRIA, Sophia-Antipolis, France.

Landin, P. (1966). The next 700 programming languages. *Communications of the ACM 9*(3), 157–166.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

Milner, R., M. Tofte, and R. Harper (1990). *The Definition of Standard ML*. MIT Press.

Naur, P. and M. Woodger (editors) (1963). Revised report on the algorithmic language Algol 60. *Communications of the ACM 60*(1), 1–17.

Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

Stoughton, A. (1988). Substitution revisited. *Theoretical Computer Science 59*, 317–325.

Tennent, R. D. (1991). *Semantics of Programming Languages*. Prentice Hall International (UK) Ltd.

Winskel, G. (1993). *The Formal Semantics of Programming Languages*. Foundations of Computing. Cambridge, Massachusetts: The MIT Press.

Lectures Appraisal Form

If lecturing standards are to be maintained where they are high, and improved where they are not, it is important for the lecturers to receive feedback about their lectures. Consequently, we would be grateful if you would complete this questionnaire, and either return it to the lecturer in question, or to the Student Administration, Computer Laboratory, William Gates Building. Thank you.

1. Name of Lecturer: `Dr Andrew M. Pitts`
2. Title of Course: `CST Part IB Semantics of Programming Languages`
3. How many lectures have you attended in this series so far? ....

   Do you intend to go to the rest of them? Yes/No/Series finished
4. What do you expect to gain from these lectures? (Underline as appropriate)

Detailed coverage of selected topics	*or*	Advanced material
Broad coverage of an area	*or*	Elementary material
Other (please specify)		

5. Did you find the content: (place a vertical mark across the line)

   Too basic --- Too complex
   Too general --- Too specific
   Well organised --- Poorly organised
   Easy to follow --- Hard to follow
6. Did you find the lecturer's delivery: (place a vertical mark across the line)

   Too slow --- Too fast
   Too general --- Too specific
   Too quiet --- Too loud
   Halting --- Smooth
   Monotonous --- Lively

   Other comments on the delivery:
7. Was a satisfactory reading list provided? Yes/No

   How might it be improved.
8. Apart from the recommendations suggested by your answers above, how else might these lectures be improved? Do any specific lectures in this series require attention? (Continue overleaf if necessary)