University of Leicester

SEMANTICS OF PROGRAMMING LANGUAGES

Course Notes

for

MC 308

Dr. R. L. Crole

Department

of

Mathematics and Computer Science

Preface

These notes are to accompany the module MC 308. They contain all of the core material for the course. For more motivation and background, as well as further comments about some of the details of proofs, please attend the lectures.

These notes are new for this year: please do let me know about any typos or other errors which you find. If you have any other (constructive) comments, please tell me about them.

Books recommended for MC 308 are listed in the Module Guide: ask if you need advice.

If you are to do well in this course, you must attend the lectures. They will give you additional examples, highlight key issues which may not appear quite as important from the notes as in fact the issues are, and give guidance towards what you need to know for the examinations. Chapter summaries will be given out in the lectures.

Acknowledgements

Some sections of these notes were developed from a course given by Dr. A. M. Pitts, University of Cambridge.

©R. L. Crole July 13, 2011

Contents

1 Introduction and Background **1**

1.1 Course Overview 1

1.2 Notation Summary 2

1.3 Inductively Defined Sets 4

1.4 Rule Induction 9

2 Operational Semantics of an Imperative Language **12**

2.1 Introduction 12

2.2 The Syntax of Expressions 12

2.3 A Transition Relation 15

2.4 An Evaluation Relation 21

2.5 Semantic Equivalence 26

2.6 Command Contexts 27

3 The Denotational Semantics of an Imperative Language **29**

3.1 Introduction 29

3.2 Preliminary Denotational Definitions 29

3.3 Denotational Semantics 34

4 The CSS Machine **38**

4.1 Architecture of the CSS Machine 38

4.2 Correctness of the CSS Machine 39

4.3 CSS Executions 47

5 Elementary Domain Theory **49**

5.1 Introduction 49

5.2 Cpos and Continuous Functions 49

5.3 Constructions on Cpos 52

5.4 Denotational Semantics of $\mathbb{IMP}$ 59

6 Operational Semantics of Functional Languages **61**

6.1 Introduction . . . 61

6.2 Types and Expressions for $\mathbb{FUN}^e$ . . . 61

6.3 Function Declarations and Programs for $\mathbb{FUN}^e$ . . . 66

6.4 Operational Semantics for $\mathbb{FUN}^e$ . . . 68

6.5 The Language $\mathbb{FUN}^l$ . . . 72

6.6 Operational Semantics for $\mathbb{FUN}^l$ . . . 72

6.7 The Language $\mathbb{TUR}^e$ and its Operational Semantics . . . 75

6.8 The Language $\mathbb{TUR}^l$ and its Operational Semantics . . . 77

7 The Denotational Semantics of Functional Languages **79**

7.1 Introduction . . . 79

7.2 Denotations for Type Assignments in $\mathbb{TUR}^e$ . . . 79

7.3 Denotations of Function Declarations and Programs in $\mathbb{TUR}^e$ . . . 83

7.4 Equivalence of Operational and Denotational Semantics . . . 86

7.5 Further Denotational Semantics . . . 90

List of Tables

2.1 Expressions ie, be, and co in $\mathbb{IMP}$ 14

2.2 Configuration Transitions $(e\,,\,s) \rightsquigarrow (e'\,,\,s')$ in $\mathbb{IMP}$ 17

2.3 Evaluation Functions $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{C}$ for $\mathbb{IMP}$ 20

2.4 Evaluation Relation $(e\,,\,s) \Downarrow s'$ in $\mathbb{IMP}$ 22

4.1 The CSS Re-Writes 40

5.1 Properties of Continuous Functions 60

6.1 Type Assignment Relation $\Delta \mid \Gamma \vdash E :: \sigma$ in $\mathbb{FUN}^e$ 65

6.2 Evaluation Relation $dec_\Delta \vdash P \Downarrow^e V$ in $\mathbb{FUN}^e$ 70

6.3 Evaluation Relation $dec_\Delta \vdash P \Downarrow^l V$ in $\mathbb{FUN}^l$ 74

6.4 Type Assignment Relation $\Delta \mid \Gamma \vdash E :: \sigma$ in $\mathbb{TUR}^e$ 76

6.5 Evaluation Relation $dec_\Delta \vdash P \Downarrow^e \underline{c}$ in $\mathbb{TUR}^e$ 77

6.6 Evaluation Relation $dec_\Delta \vdash P \Downarrow^l \underline{c}$ in $\mathbb{TUR}^l$ 78

7.1 Denotational Semantics $[\![\Delta \mid \Gamma \vdash E]\!]$ in $\mathbb{TUR}^e$ 81

7.2 Denotational Semantics $[\![\Delta \mid \Gamma \vdash E]\!]$ in $\mathbb{TUR}^l$ 91

List of Figures

2.1 A Transition Sequence in $\mathbb{IMP}$ 18

2.2 An Example Deduction for $\leadsto$ 18

2.3 An Example Deduction of an Evaluation 23

1

Introduction and Background

1.1 Course Overview

Motivation 1.1.1 At least two quite distinct issues can be associated with a programming language:

(1) The definition of the SYNTAX of the language. This involves specifying an alphabet of symbols and characters, along with a definition of the expressions, phrases, programs, values and so on of the language.

(2) The definition of the SEMANTICS of the language. This involves specifying the meaning of the expressions, phrases, programs, values, modules, datatypes and so on of the language. We talk about giving a semantics to the syntax.

In this course we shall study (2). We shall see how techniques of mathematics and logic can be used to give completely rigorous definitions of the meanings of programs written in a particular syntax. What are the benefits of a formal semantics for a programming language?

- It gives a basis for:

    - the correctness of implementations;
    - verifications that programs meet their specification;
    - efficiency analysis; and
    - proving that certain fragments of code in a program are interchangeable.

- It can help to detect hidden, or non-obvious features of a programming language. For example, the *dynamic binding* found in many versions of LISP was regarded as a "bug" in the first, experimental implementations, but this soon became an accepted "feature" of LISP.

- It provides a mathematical analysis of computational and programming constructs which are independent of the actual programming language.

- Semantic techniques can often be used in the design of a programming language; for example ML arose in this way.

There are a number of different kinds of semantic styles:

(1) In OPERATIONAL semantics, the meaning of programming language expressions are defined by giving rules which specify ways in which the expressions evaluate or execute. The rules used make use of the syntactic structure of program expressions.

(2) In DENOTATIONAL semantics, the meaning of a programming language is given by specifying various mathematical structures and functions which provide a model of the programming language.

(3) In AXIOMATIC semantics, the meaning of the expressions of a programming language are given indirectly through a formal logic which expresses properties of programs.

In MC 308 we concentrate on operational and denotational semantics. Before we begin in earnest, we give a summary of some very basic mathematical facts, and then present an account of some techniques of induction which will be used throughout the course.

1.2 Notation Summary

Logic

We use superscripts and primes to denote variants of mathematical entities. For example, if x and y are variables, so too are x_1, x_2, y_{10}, x', y'''' and so on. We write $A \equiv B$ to indicate **syntactic identity**. Thus $2+3=5$ but $2+3 \not\equiv 5$.

If P and Q are mathematical propositions, we can form new propositions as follows:

- P *and* Q (sometimes written $P \wedge Q$);
- P *or* Q (sometimes written $P \vee Q$);
- P *implies* Q (sometimes written $P \Longrightarrow Q$ or $P \rightarrow Q$);
- *not* P (sometimes written $\neg P$);
- P *if and only if* Q (often written P *iff* Q or $P \Longleftrightarrow Q$)—this is simply an abbreviation for

$$(P \Longrightarrow Q) \quad \textit{and} \quad (Q \Longrightarrow P);$$

- *for all* x, P (sometimes written $\forall x.\ P$);
- *there exists* x, P (sometimes written $\exists x.\ P$).

Sets

We shall use the following sets throughout MC 308:

empty set	$\varnothing$
natural numbers	$\mathbb{N} = \{ 0, 1, 2, 3, \ldots \}$
integers	$\mathbb{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}$
Booleans	$\mathbb{B} = \{ T, F \}$

$a \in A$ denotes set membership, for example $T \in \mathbb{B}$. The following definitions are assumed to be well known:

Subset	$A \subseteq B$	$\iff$	for all $a \in A, a \in B$
		$\iff$	for all $x, (x \in A \Longrightarrow x \in B)$
Union	$A \cup B$	$\stackrel{\text{def}}{=}$	$\{ x \mid x \in A \text{ or } x \in B \}$
Intersection	$A \cap B$	$\stackrel{\text{def}}{=}$	$\{ x \mid x \in A \text{ and } x \in B \}$
Difference	$A \setminus B$	$\stackrel{\text{def}}{=}$	$\{ x \mid x \in A \text{ and } x \notin B \}$
Powerset	$\mathcal{P}(A)$	$\stackrel{\text{def}}{=}$	$\{ S \mid S \subseteq A \}$
Cartesian Product	$A \times B$	$\stackrel{\text{def}}{=}$	$\{ (a, b) \mid a \in A \text{ and } b \in B \}$

Total Functions

We define the **set of total functions** between sets A and B to be

$$[A, B]_{tot} \stackrel{\text{def}}{=} \{ f \in \mathcal{P}(A \times B) \mid \forall a \in A, \exists \text{ a unique } b \in B, (a, b) \in f \}.$$

We usually refer to a total function simply as a function. We write $f : A \to B$ for $f \in [A, B]_{tot}$. If $a \in A$ and $f : A \to B$ then $f(a)$ denotes the unique $b \in B$ for which $(a, b) \in f$. If also $g : B \to C$ is a function, then there is a function denoted by $g \circ f : A \to C$, which is defined by $(g \circ f)(a) \stackrel{\text{def}}{=} g(f(a))$ on each $a \in A$. We call $g \circ f$ the **composition** of f and g. Informally, $g \circ f$ is the function which first applies f and then applies g. The **identity** function, written $id_A : A \to A$ is the function defined by $id_A(a) \stackrel{\text{def}}{=} a$ on each $a \in A$.

Partial Functions

We define the **set of partial functions** between sets A and B to be

$$[A, B]_{par} \stackrel{\text{def}}{=} \{ f \in \mathcal{P}(A \times B) \mid \forall a \in A, \forall b, b' \in B, ((a, b) \in f \text{ and } (a, b') \in f) \Longrightarrow b = b' \}.$$

We write $f : A \rightharpoonup B$ to mean that $f \in [A, B]_{par}$. If $a \in A$ and $f : A \rightharpoonup B$ either there exists a unique $b \in B$ for which $(a, b) \in f$, or such a b does not exist. In the former case we say that "$f(a)$ is defined" and in this case $f(a)$ denotes the unique b. In the latter case we say that "$f(a)$ is undefined". Note that $\varnothing \in [A, B]_{par}$ satisfies the definition of a partial function, so $\varnothing : A \rightharpoonup B$. We say $\varnothing$ is the totally undefined partial function between A and B—why is this?

Monotone Functions

If $f : (D, \preceq) \to (D, \preceq)$ is a *monotone* function between posets, that is $d \preceq d'$ implies $f(d) \preceq f(d')$ for all $d, d' \in D$, then $x \in D$ is a **prefixpoint** of f if $f(x) \preceq x$, and a **fixpoint** of f if $f(x) = x$. We write $\mathit{fix}(f)$ for the least element in the set of fixpoints of f, if the least element exists.

1.3 Inductively Defined Sets

Motivation 1.3.1 As motivation for this section, consider the following:

The set $E \subseteq \mathbb{N}$ of even natural numbers is the least subset of the natural numbers satisfying

(a) $0 \in E$, and

(b) for all $n \in \mathbb{N}$, if $n \in E$ then $n + 2 \in E$.

Note that "*least*" means[1] that if another subset $S \subseteq \mathbb{N}$ satisfies (a) and (b) (by which we mean $0 \in S$, and for all $n \in \mathbb{N}$, $n \in S$ implies $n+2 \in S$) then $E \subseteq S$. The above definition of E amounts to saying that the elements of E are created by the rules (a) and (b), and that (by leastness) there can be no other elements in E. We say that E is *inductively defined* by the *rules* (a) and (b). So $E = \{ 0, 2, 4, 6, 8, \dots \}$, another set satisfying (a) and (b) is (for example) $S \stackrel{\text{def}}{=} \{ 0, 2, 4, 5, 6, 7, 8, 9, \dots \}$, and indeed $E \subseteq S$.

More generally, an inductively defined set I is the least (or smallest) set for which

(a) certain elements are always in I, such as $c \in I$; and

(b) whenever certain elements $h_1 \in I$ and $h_2 \in I$ and ... and $h_k \in I$, then $c' \in I$.

(a) is sometimes called the "base clause" and (b) the "inductive clause". In the last example, I is E, c is 0, h_1 is n, $k = 1$, and c' is $n + 2$. We shall now give some machinery in which we can give a very precise formulation of inductively defined sets.

Definitions 1.3.2 A **rule** R for inductively defining a set denoted by I is a pair (H, c) where H is any finite set, and c is an element. Note that H might be $\varnothing$, in which case we say that R is a **base** rule. If H is non-empty we say R is an **inductive** rule. In the case that H is non-empty we might write $H = \{ h_1, \dots, h_k \}$ where $1 \leq k$. We can write down a base rule $R = (\varnothing, c)$ for inductively defining the set I using the following notation

Base

$$\frac{}{c\ \texttt{in}\ I}\ (R)$$

and an inductive rule $R = (H, c) = (\{ h_1, \dots, h_k \}, c)$ as

[1] Let $\mathcal{E}$ denote the set of all sets satisfying (a) and (b). Partially order $\mathcal{E}$ by $\subseteq$. Then E is the least element of $\mathcal{E}$.

Inductive

$$\frac{h_1 \texttt{ in } I \quad h_2 \texttt{ in } I \quad \ldots \quad h_k \texttt{ in } I}{c \texttt{ in } I}\,(R)$$

Note that the order of the statements $h_1 \texttt{ in } I \quad h_2 \texttt{ in } I \quad \ldots \quad h_k \texttt{ in } I$ appearing above the line is irrelevant: the h_i are elements of the *set* H. You may like to think of the $h_i \texttt{ in } I$ as *hypotheses* and $c \texttt{ in } I$ as a *conclusion*. The notation $h_i \texttt{ in } I$ is meant to suggest that h_i is an element of the set I.

Any set S[2] is **closed** under a base rule $\frac{}{c \texttt{ in } I}$ if $c \in S$; and is **closed** under an inductive rule $\frac{h_1 \texttt{ in } I \quad h_2 \texttt{ in } I \quad \ldots \quad h_k \texttt{ in } I}{c \texttt{ in } I}$ if whenever $h_1 \in S$ and $h_2 \in S$ and $\ldots$ and $h_k \in S$, then $c \in S$. The set S is **closed under** a set of rules $\mathcal{R}$ if S is closed under each rule in $\mathcal{R}$. We can now say that:

Inductively Defined Sets

A set I is **inductively defined** by a set of rules $\mathcal{R}$ if

IC I is closed under $\mathcal{R}$; and

IL for *every* set S which is closed under $\mathcal{R}$, we have $I \subseteq S$.

Note that a base rule corresponds to the "base clause" and an inductive rule corresponds to the "inductive clause" as described in Motivation 1.3.1.

Example 1.3.3 A set[3] $\mathcal{R}$ of rules for defining the set E of even numbers is $\mathcal{R} = \{1, 2\}$ where

$$\frac{}{0 \texttt{ in } E}\,(1) \qquad\qquad \frac{e \texttt{ in } E}{e+2 \texttt{ in } E}\,(2)$$

IC means that elements of the inductively defined set may be built up by applying the rules: it says that

$$(1) \qquad 0 \in E$$
$$(2) \qquad \text{for all } e,\ e \in E \Longrightarrow e+2 \in E.$$

and thus the elements of E are 0, 2 (that is, $0 \in E$ implies $0 + 2 = 2 \in E$), 4 and so on. **IL** amounts to saying that there can be no elements of E other than those arising by successive application of the rules: any other set S closed under the rules must contain E as a subset. An example of such an S is $\{0, 2, 4, 6, 7, 8, 9, 10, \ldots\}$. Check this!!

[2] S is any set, and might well be I!

[3] Strictly speaking, the elements of the set $\mathcal{R}$ are the numbers 1 and 2. But these are just intended to be labels for our two rules, and no confusion should result.

Definitions 1.3.4 If I is inductively defined by a set of rules $\mathcal{R}$, a **deduction** of $x \texttt{ in } I$ is given by a finite list

$$y_1 \texttt{ in } I, y_2 \texttt{ in } I, \ldots, y_m \texttt{ in } I \qquad (*)$$

where

(i) $m \in \mathbb{N}$ and $m \geq 1$;

(ii) $y_1 \texttt{ in } I$ is a conclusion of a base rule;

(iii) for any $2 \leq i \leq m$, $y_i \texttt{ in } I$ is the conclusion of some rule R for which the hypotheses of R form a (possibly empty) subset of $\{ y_1 \texttt{ in } I, \ldots, y_{i-1} \texttt{ in } I \}$ (that is, the hypotheses have already been deduced); and

(iv) $y_m \texttt{ in } I$ is $x \texttt{ in } I$.

Note that $(*)$ is a *list*—the order of the y_i is crucial. We call m the **length** of the deduction.

Proposition 1.3.5 Suppose that I is inductively defined by a set of rules $\mathcal{R}$. Then

$$I = \{ x \mid \text{there exists a deduction of } x \texttt{ in } I \}$$

Proof Write $J \stackrel{\text{def}}{=} \{ x \mid \text{there exists a deduction of } x \texttt{ in } I \}$. One can check that J is closed under $\mathcal{R}$ (do it!) so that $I \subseteq J$ by **IL**. We show that $J \subseteq I$ as follows: we prove by Mathematical Induction on n that

$\forall n \geq 1$, for all deductions l of length $\leq n$, $\forall x$, if l is a deduction of $x \texttt{ in } I$, then $x \in I$.

Check this! Thus if $x \in J$ there must be a deduction of $x \texttt{ in } I$ which has length n for some $n \geq 1$, so that $x \in I$. Hence $J \subseteq I$. We conclude that $I = J$ as required. $\square$

Remark 1.3.6 Proposition 1.3.5 amounts to saying that for any x,

$x \in I$ if and only if there exists a deduction of $x \texttt{ in } I$.

⋙ **Warning 1.3.7** ***IC** **means that the elements of the** Inductively **defined set** I **are** Constructed **by "applying" the rules in** $\mathcal{R}$—$x \in I$ if **there exists a deduction of** $x \texttt{ in } I$. **IL** **captures precisely the idea that** I **is the** Least **set satisfying the rules, that is, there can be no elements of** I **other than those constructed by the rules**—$x \in I$ only if **there exists a deduction of** $x \texttt{ in } I$. **Here, least refers to the subset ordering** $\subseteq$ **on sets.***

Motivation 1.3.8 We now generalise the definition of an inductively defined set to that of *simultaneously* inductively defined sets. This idea is *crucial* to the rest of the course!!

Definitions 1.3.9 Let $I_1, I_2, \ldots, I_n$ where $n \geq 1$ denote sets. A **base rule** takes the form

Base

$$\frac{}{c \texttt{ in } I_i}\,(R)$$

where $1 \leq i \leq n$, and an **inductive rule** takes the form

Inductive

$$\frac{h_1 \texttt{ in } I_{i_1} \quad h_2 \texttt{ in } I_{i_2} \quad \ldots \quad h_k \texttt{ in } I_{i_k}}{c \texttt{ in } I_i}\,(R)$$

where $i_1, i_2, \ldots, i_k, i$ are elements of $\{1, 2, \ldots, n\}$.

Sets $S_1, S_2, \ldots, S_n$ are **closed** under a base rule $\frac{}{c \texttt{ in } I_i}$ if $c \in S_i$; and are **closed** under an inductive rule $\frac{h_1 \texttt{ in } I_{i_1} \quad h_2 \texttt{ in } I_{i_2} \quad \ldots \quad h_k \texttt{ in } I_{i_k}}{c \texttt{ in } I_i}$ if whenever $h_1 \in S_{i_1}$ and $h_2 \in S_{i_2}$ and $\ldots$ and $h_k \in S_{i_k}$, then $c \in S_i$. The sets $S_1, S_2, \ldots, S_n$ are **closed under** $\mathcal{R}$ if $S_1, S_2, \ldots, S_n$ are closed under each rule in $\mathcal{R}$. We can now say that:

Simultaneously Inductively Defined Sets

Sets $I_1, I_2, \ldots, I_n$ are **simultaneously inductively defined** by a set of rules $\mathcal{R}$ if

IC $I_1, I_2, \ldots, I_n$ are closed under $\mathcal{R}$; and

IL for *every* collection of n sets (say $S_1, S_2, \ldots, S_n$) which are closed under $\mathcal{R}$, we have $I_1 \subseteq S_1$ and $I_2 \subseteq S_2$ and $\ldots$ and $I_n \subseteq S_n$.

Definitions 1.3.10 If $I_1, I_2, \ldots, I_n$ are inductively defined by a set of rules $\mathcal{R}$ a **deduction** that $x \texttt{ in } I_i$ is given by a list

$$y_1 \texttt{ in } I_{i_1}, y_2 \texttt{ in } I_{i_2}, \ldots, y_m \texttt{ in } I_{i_m} \qquad (*)$$

with $m \geq 1$, $i_1, i_2, \ldots, i_m \in \{1, \ldots, n\}$, where

(i) $y_1 \texttt{ in } I_{i_1}$ is a conclusion of a base rule;

(ii) for any $2 \leq j \leq m$, $y_j \texttt{ in } I_{i_j}$ is the conclusion of some rule R for which the hypotheses of R form a (possibly empty) subset of $\{y_1 \texttt{ in } I_{i_1}, \ldots, y_{j-1} \texttt{ in } I_{i_{j-1}}\}$ (that is, the hypotheses have already been deduced); and

(iii) $y_m \texttt{ in } I_{i_m}$ is $x \texttt{ in } I_i$.

Note that $(*)$ is a *list*—the order of the y_j `in` I_{i_j} is crucial.

A **labelled** deduction that x `in` I_i looks like

$$\begin{array}{ll} y_1 \;\texttt{in}\; I_{i_1} & (Ri_1) \\ y_2 \;\texttt{in}\; I_{i_2} & (Ri_2) \\ \ldots & \\ y_m \;\texttt{in}\; I_{i_m} & (Ri_m) \end{array}$$

in which the vertical sequence of the y_j `in` I_{i_j} is a deduction of x `in` I_i, and each Ri_j is the rule from $\mathcal{R}$ which has been used to deduce that y_j `in` I_{i_j}.

Remark 1.3.11 One can check that if $I_1, I_2, \ldots, I_n$ are simultaneously inductively defined sets, then for each $i \in \{1, \ldots, n\}$,

$$x \in I_i \qquad \text{if and only if} \qquad \text{there exists a deduction of } x \;\texttt{in}\; I_i.$$

Examples 1.3.12

(1) Suppose that Σ is any set, which we think of as an **alphabet**. Each element l of Σ is called a **letter**. We inductively define the set Σ^* of *all non-empty* **words** over the alphabet Σ by the set of rules $\mathcal{R} \stackrel{\text{def}}{=} \{1, 2\}$ (so 1 and 2 are just labels for rules!) given by[4]

$$\frac{}{l \;\texttt{in}\; \Sigma^*}\,[l \in \Sigma]\;\;(1) \qquad\qquad \frac{w \;\texttt{in}\; \Sigma^* \quad w' \;\texttt{in}\; \Sigma^*}{ww' \;\texttt{in}\; \Sigma^*}\,(2)$$

A word is just a list of letters. **IC** says that Σ^* is closed under the rules 1 and 2. Closure under Rule 1 says that any letter l is a word, that is, $l \in \Sigma^*$. Closure under Rule 2 says that if w and w' are any two words, that is $w \in \Sigma^*$ and $w' \in \Sigma^*$, then the list of letters ww' obtained by writing down the list of letters w followed immediately by the list of letters w' is a word (that is, $ww' \in \Sigma^*$). Note that it may be helpful to think of l, w and w' in rules (1) and (2) as variables.

As an example, let $\Sigma = \{a, b, c\}$. We can show that $abac \in \Sigma^*$ by giving a labelled deduction of $abac$ `in` Σ^*:

$$\begin{array}{rl} a \;\texttt{in}\; \Sigma^* & (1) \\ b \;\texttt{in}\; \Sigma^* & (1) \\ ab \;\texttt{in}\; \Sigma^* & (2) \\ c \;\texttt{in}\; \Sigma^* & (1) \\ ac \;\texttt{in}\; \Sigma^* & (2) \\ abac \;\texttt{in}\; \Sigma^* & (2) \end{array}$$

If we compare this labelled deduction with the general definition in Definitions 1.3.10, we see that $m = 6$, and $y_1 = a$, $y_2 = b$, etc to $y_6 = abac$. We have

(i) $y_1 = a$ is a conclusion to the base rule (1);

[4]In rule (1), $[l \in \Sigma]$ is called a **side condition**. It means that in reading the rule, l can be any element of Σ.

(ii) (for example if $i = 5$) $y_5 = ac$ is a conclusion to (2). Here, the set of hypotheses is $\{\, a \ \mathtt{in}\ \Sigma^*, c \ \mathtt{in}\ \Sigma^* \,\}$, and certainly the set of hypotheses is a subset of those $\xi \ \mathtt{in}\ \Sigma^*$ already deduced:

$$\begin{aligned} \{\, a \ \mathtt{in}\ \Sigma^*, c \ \mathtt{in}\ \Sigma^* \,\} &\subseteq \{\, a \ \mathtt{in}\ \Sigma^*, b \ \mathtt{in}\ \Sigma^*, ab \ \mathtt{in}\ \Sigma^*, c \ \mathtt{in}\ \Sigma^* \,\} \\ &= \{\, y_1 \ \mathtt{in}\ \Sigma^*, \ldots, y_{5-1} \ \mathtt{in}\ \Sigma^* \,\} \end{aligned}$$

(iii) $y_m = y_6 = abac$.

We can also write a **deduction tree** which makes explicit which hypotheses are used when a rule is applied:

$$\dfrac{\dfrac{\dfrac{}{a \ \mathtt{in}\ \Sigma^*}\,(1) \qquad \dfrac{}{b \ \mathtt{in}\ \Sigma^*}\,(1)}{ab \ \mathtt{in}\ \Sigma^*}\,(2) \qquad \dfrac{\dfrac{}{a \ \mathtt{in}\ \Sigma^*}\,(1) \qquad \dfrac{}{c \ \mathtt{in}\ \Sigma^*}\,(1)}{ac \ \mathtt{in}\ \Sigma^*}\,(2)}{abac \ \mathtt{in}\ \Sigma^*}\,(2)$$

(2) Let $\Sigma = \{\, a, b, c, d, e \,\}$ and let sets I_1 and I_2 of words be simultaneously inductively defined by the rules

$$\dfrac{}{b \ \mathtt{in}\ I_1}\,(1) \qquad \dfrac{}{c \ \mathtt{in}\ I_2}\,(2) \qquad \dfrac{w \ \mathtt{in}\ I_1 \quad w' \ \mathtt{in}\ I_2}{ww' \ \mathtt{in}\ I_2}\,(3)$$

$$\dfrac{w \ \mathtt{in}\ I_1}{aadwe \ \mathtt{in}\ I_2}\,(4) \qquad \dfrac{w' \ \mathtt{in}\ I_2}{w'e \ \mathtt{in}\ I_2}\,(5) \qquad \dfrac{w' \ \mathtt{in}\ I_2}{aw'a \ \mathtt{in}\ I_1}\,(6)$$

A deduction tree for $aaadbeace \ \mathtt{in}\ I_2$ is

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b \ \mathtt{in}\ I_1}\,(1)}{aadbe \ \mathtt{in}\ I_2}\,(4)}{aaadbea \ \mathtt{in}\ I_1}\,(6) \qquad \dfrac{\dfrac{}{c \ \mathtt{in}\ I_2}\,(2)}{ce \ \mathtt{in}\ I_2}\,(5)}{aaadbeace \ \mathtt{in}\ I_2}\,(3)$$

1.4 Rule Induction

Definitions 1.4.1 We state the Principle of Rule Induction for Simultaneously Inductively Defined sets:

Rule Induction

Let $I_1, I_2, \ldots, I_n$ be inductively defined by a set of rules $\mathcal{R}$. Suppose we wish to show that for all $i \in \{1, 2, \ldots, n\}$ the property $Prop_{I_i}(x)$ holds for all elements $x \in I_i$, that is, we wish to prove

$$\begin{aligned}&\text{for all } x \in I_1, Prop_{I_1}(x) \text{ and}\\ &\text{for all } x \in I_2, Prop_{I_2}(x) \text{ and}\\ &\vdots\\ &\text{for all } x \in I_n, Prop_{I_n}(x).\end{aligned}$$

Then all we need to do is

- for every base rule $\frac{}{b \texttt{ in } I_i} \in \mathcal{R}$ prove that if $b \in I_i$ then $Prop_{I_i}(b)$ holds; and
- for every inductive rule $\frac{h_1 \texttt{ in } I_{i_1}, \ldots, h_k \texttt{ in } I_{i_k}}{c \texttt{ in } I_i} \in \mathcal{R}$ prove that if $h_1 \in I_{i_1}$ and $h_2 \in I_{i_2}$ and ... and $h_k \in I_{i_k}$, and $Prop_{I_{i_1}}(h_1)$ and $Prop_{I_{i_2}}(h_2)$ and ... and $Prop_{I_{i_k}}(h_k)$ all hold, so does $Prop_{I_i}(c)$.

We call the assertions $Prop_{I_{i_j}}(h_j)$ (where $1 \leq j \leq k$) **inductive hypotheses**. We refer to carrying out • above as *showing that the properties are closed under the rules in $\mathcal{R}$, or sometimes as verifying* ***property closure***.

To see that Rule Induction works, write

$$S_i \stackrel{\text{def}}{=} \{ x \mid x \in I_i \text{ and } Prop_{I_i}(x) \text{ holds } \}.$$

Notice that checking the rules in $\mathcal{R}$ are closed under the properties amounts to verifying that $S_1, S_2, \ldots, S_n$ *are* closed under $\mathcal{R}$. Thus property **IL** tells us that $I_i \subseteq S_i$ for each i. Also, $S_i \subseteq I_i$ for each i by definition. Hence $S_i = I_i$ for each i. So if $i \in \{1, 2, \ldots, n\}$ and x is *any* element of I_i, then $x \in S_i$, and so $Prop_{I_i}(x)$ holds.

Examples 1.4.2 Refering to Examples 1.3.12 part (2), suppose that we wish to prove that

every word in I_2 has an even number of occurrences of a.

Write $\#(w)$ for the number of occurrences of a in w. For a word w, what shall we take $Prop_{I_1}(w)$ and $Prop_{I_2}(w)$ to be? Obviously we want

$$Prop_{I_2}(w) \stackrel{\text{def}}{=} \#(w) \text{ is even }.$$

If we look at rule (4), it is clear that $\#(aadwe)$ will be even if $\#(w)$ is even, *where* $w \in I_1$. So we guess that (maybe) $\#(w)$ is also even for all words in I_1, and set

$$Prop_{I_1}(w) \stackrel{\text{def}}{=} \#(w) \text{ is even }.$$

Let us now apply Rule Induction: we check the closure of each rule (1) to (6) under the given properties.

(*Rule (1)*): $\#(b) = 0$, even, so $Prop_{I_1}(b)$ holds.

(*Rule (2)*): $\#(c) = 0$, even, so $Prop_{I_2}(c)$ holds.

(*Rule (3)*): Suppose that $Prop_{I_1}(w)$ and $Prop_{I_2}(w')$ hold (these are the Inductive Hypotheses). Note that $\#(ww') = \#(w) + \#(w')$, and so $\#(ww')$ is even by the inductive hypotheses. Thus $Prop_{I_2}(ww')$ holds.

(*Rule (4)*): Suppose that $Prop_{I_1}(w)$ holds. Then clearly we have $\#(aadwe) = \#(w) + 2$ is even, so $Prop_{I_2}(aadwe)$ holds.

(*Rule (5)*): Suppose that $Prop_{I_2}(w')$ holds. Then $\#(w'e) = \#(w')$ is even. Hence $Prop_{I_2}(w'e)$ holds.

(*Rule (6)*): Suppose that $Prop_{I_2}(w')$ holds. Then $\#(aw'a) = 2 + \#(w')$ is even. Hence $Prop_{I_1}(aw'a)$ holds.

Thus by Rule Induction we are done, and we can conclude that

$$\text{for all } w \in I_1,\ \#(w) \text{ is even, and}$$
$$\text{for all } w \in I_2,\ \#(w) \text{ is even.}$$

Thus we have proved both the original proposition, and into the bargain that all words in I_1 also have an even number of occurrences of a.

2

Operational Semantics of an Imperative Language

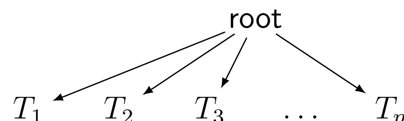
2.1 Introduction

Motivation 2.1.1 We shall look at a formal definition of a simple imperative language which we call $\mathbb{IMP}$. We define the syntax of this language, and then describe how programs in the language execute—the *operational semantics* of $\mathbb{IMP}$. The expressions of the language comprise integers, Booleans and commands. As our language is imperative, it has a concept of *state*. Thus $\mathbb{IMP}$ has a collection of (memory) locations which hold data—a state is any particular assignment of data to the locations. The commands of the language comprise instructions for changing the state. A program, or *configuration*, in $\mathbb{IMP}$ consists of an expression together with a specified state. The program executes by using instructions coded by the expression to manipulate the state. The (final) results of a program execution are given by the state at the end of execution.

If e is an expression and s a state, then a configuration will be any pair of the form $(e\,,\,s)$. We shall define assertions of the form $(e\,,\,s) \rightsquigarrow (e'\,,\,s')$ which assert that in state s, e executes in one cycle to e' with the state after the computation cycle being s'. Such assertions comprise a formal *operational semantics*. We shall also give an operational semantics which shows how expressions can execute in a multiple number of steps to produce a final state (and a program output if the expression is an integer or a Boolean), and show how this style of operational semantics matches the former "single cycle" definition in an exact way.

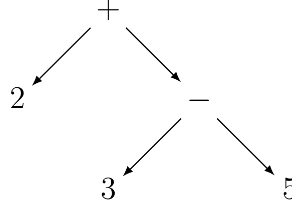
2.2 The Syntax of Expressions

Motivation 2.2.1 We begin to describe formally the language $\mathbb{IMP}$. The first step is to give a definition of the syntax of expressions of the language. In this course, expression syntax will in fact be abstract syntax—every syntactic object will be a finitely branching tree. We shall adopt the following notation for finite trees: If T_1, T_2, T_3 and so on to T_n is a (finite) sequence of finite trees, then we shall write $\mathsf{root}(T_1, T_2, T_3, \ldots, T_n)$ for the finite tree



whose root is denoted by the symbol root.

For example, $+(2,-(3,5))$ denotes the tree



We shall often use infix notation, writing (for example) $2+3$ instead of $+(2,3)$ if such notation is clearer to read. The above example would be written $2+(3-5)$.

Definitions 2.2.2 The expression syntax of $\mathbb{IMP}$ will be built out of various sets of symbols. These are

Loc	$\stackrel{\text{def}}{=}$	$\{x_1, x_2, \ldots\}$	the set of **locations**;
$ICst$	$\stackrel{\text{def}}{=}$	$\{\underline{n} \mid n \in \mathbb{Z}\}$	the set of **integer constants**;
$BCst$	$\stackrel{\text{def}}{=}$	$\{\underline{b} \mid b \in \mathbb{B}\}$	the set of **Boolean constants**;
$IOpr$	$\stackrel{\text{def}}{=}$	$\{+,-,*\}$	a fixed, finite set of **integer valued operators**;
$BOpr$	$\stackrel{\text{def}}{=}$	$\{=,<,\leq,\ldots\}$	a fixed, finite set of **Boolean valued operators**;

We shall let the symbol c range over elements of $\mathbb{Z} \cup \mathbb{B}$. Note that the operator symbols will be regarded as denoting the obvious mathematical functions. For example, $\leq$ is the function which takes a pair of integers and returns a truth value. Thus $\leq : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ is the function given by $(m,n) \mapsto m \leq n$, where

$$m \leq n = \begin{cases} T & \text{if } m \text{ is less than or equal to } n \\ F & \text{otherwise} \end{cases}$$

For example, $5 \leq 2 = F$.

Note that we write $\underline{c}$ to indicate that the constant c is "held in memory". We shall require that $\underline{c} = \underline{c'}$ if and only if $c = c'$. Given (for example) $\underline{2}$ and $\underline{3}$ we cannot add these "numbers" until our programming language $\mathbb{IMP}$ instructs that the contents of the memory locations be added—thus $\underline{2}+\underline{3} \neq \underline{5}$. However, when $\underline{2}$ is added to $\underline{3}$ by $\mathbb{IMP}$, the result is $\underline{5}$, and we *shall* write

$$\underline{2+3} = \underline{5}.$$

The set of expression **constructors** is specified by

$$Loc \cup ICst \cup BCst \cup IOpr \cup BOpr \cup \{\,\mathsf{skip}, \mathsf{assign}, \mathsf{sequence}, \mathsf{cond}, \mathsf{while}\,\}.$$

We now define the expressions of the language $\mathbb{IMP}$. We begin by specifying three sets, *IExp*, of **integer expressions**, *BExp*, of **Boolean expressions**, and *Com*, of **commands**. The set *Exp* of expressions of the language is given by the (disjoint) union of these three sets:

$$Exp \stackrel{\text{def}}{=} IExp \cup BExp \cup Com.$$

Each expression is a finite tree, whose nodes are expression constructors. If e ranges over expressions, we shall adopt the following abbreviations:

$$\frac{}{x \texttt{ in } \mathit{IExp}} \; [\text{when } x \in \mathit{Loc}] \quad e\text{LOC} \qquad \frac{}{\underline{n} \texttt{ in } \mathit{IExp}} \; [\text{when } n \in \mathbb{Z}] \quad e\text{INT}$$

$$\frac{}{\underline{T} \texttt{ in } \mathit{BExp}} \; e\text{TRUE} \qquad \frac{}{\underline{F} \texttt{ in } \mathit{BExp}} \; e\text{FALSE}$$

$$\frac{ie_1 \texttt{ in } \mathit{IExp} \quad ie_2 \texttt{ in } \mathit{IExp}}{ie_1 \; op \; ie_2 \texttt{ in } \mathit{IExp}} \; [\text{when } op \in \mathit{IOpr}] \quad e\text{IOP}$$

$$\frac{ie_1 \texttt{ in } \mathit{IExp} \quad ie_2 \texttt{ in } \mathit{IExp}}{ie_1 \; op \; ie_2 \texttt{ in } \mathit{BExp}} \; [\text{when } op \in \mathit{BOpr}] \quad e\text{BOP}$$

$$\frac{}{\mathsf{skip} \texttt{ in } \mathit{Com}} \; e\text{SKIP} \qquad \frac{ie \texttt{ in } \mathit{IExp}}{x := ie \texttt{ in } \mathit{Com}} \; [\text{when } x \in \mathit{Loc}] \quad e\text{ASS}$$

$$\frac{co_1 \texttt{ in } \mathit{Com} \quad co_2 \texttt{ in } \mathit{Com}}{co_1 \; ; \; co_2 \texttt{ in } \mathit{Com}} \; e\text{SEQ}$$

$$\frac{be \texttt{ in } \mathit{BExp} \quad co_1 \texttt{ in } \mathit{Com} \quad co_2 \texttt{ in } \mathit{Com}}{\mathsf{if} \; be \; \mathsf{then} \; co_1 \; \mathsf{else} \; co_2 \texttt{ in } \mathit{Com}} \; e\text{COND} \qquad \frac{be \texttt{ in } \mathit{BExp} \quad co \texttt{ in } \mathit{Com}}{\mathsf{while} \; be \; \mathsf{do} \; co \texttt{ in } \mathit{Com}} \; e\text{LOOP}$$

Table 2.1: Expressions ie, be, and co in $\mathbb{IMP}$

- We write $e := e'$ for the finite tree $\mathsf{assign}(e, e')$;
- $e \; ; \; e'$ for $\mathsf{sequence}(e, e')$;
- $\mathsf{if} \; e \; \mathsf{then} \; e' \; \mathsf{else} \; e''$ for $\mathsf{cond}(e, e', e'')$; and
- $\mathsf{while} \; e \; \mathsf{do} \; e'$ for $\mathsf{while}(e, e')$.

The sets IExp, BExp and Com are simultaneously inductively defined by the rules in Table 2.1. We shall also adopt the following bracketing and scoping conditions:

- Arithmetic operators group to the left. Thus $ie_1 \; op \; ie_2 \; op \; ie_3$ abbreviates $(ie_1 \; op \; ie_2) \; op \; ie_3$ with the expected extension to any finite number of integer expressions.
- Sequencing associates to the right.
- $\mathsf{if} \; be \; \mathsf{then} \; co \; \mathsf{else} \; co'$ means $\mathsf{if} \; (be) \; \mathsf{then} \; (co) \; \mathsf{else} \; (co')$.
- $\mathsf{while} \; be \; \mathsf{do} \; co$ means $\mathsf{while} \; (be) \; \mathsf{do} \; (co)$.

Remark 2.2.3 We will usually denote elements of any given set of syntactic objects by one or two fixed symbols. So for example, e is always used to denote expressions, that is,

elements of *Exp*. This allows us to cut down on notation. As an example, the following all have identical meanings:

- For all e, ...
- For all expressions e, ...
- For all $e \in Exp$, ...

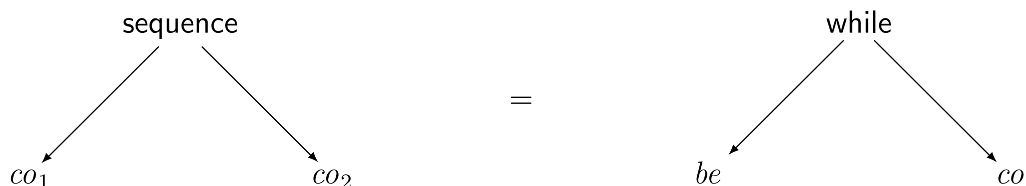
The first sentence is shorter than the others, but conveys the same meaning. *ie* ranges over integer expressions, *be* over Boolean expressions, *co* over commands, and *op* over arithmetic operators.

We shall use brackets as informal punctuation when writing expressions, for example compare the following two commands:

$$\text{if } be \text{ then } co_1 \text{ else } (co_2 \; ; \; co_3) \qquad \text{and} \qquad (\text{if } be \text{ then } co_1 \text{ else } co_2) \; ; \; co_3.$$

⋙ **Warning 2.2.4** ***A feature of inductively defined syntax is that whenever a syntactic expression is known to be an element of an inductively defined set, we can determine which rules were used to construct the expression. This is precisely where we make use of the fact that*** $\mathbb{IMP}$ ***expressions are finite trees. We illustrate by example. Suppose that*** while be do co ***is a command, that is*** while be do $co \in Com$. ***We know that there is a deduction of this fact, using the rules in Table 2.1. Then the only rule which could be used in the last step of the deduction must be*** *eLOOP*. ***This follows from the fact that the root of the finite tree*** while be do co ***is*** while.

By way of illustration, if while be do co in Com ***had been deduced from another rule,*** *eSEQ* ***say, then there would be commands*** co_1 ***and*** co_2 ***for which***



and so sequence $\equiv$ while ***which is nonsense. Similarly, no other rule (apart from*** *eLOOP*) ***could be used to deduce that*** while be do co in Com.

It also follows that we must have $be \in BExp$ ***and*** $co \in Com$***—these facts must hold if*** while be do $co \in Com$. ***Why?***

2.3 A Transition Relation

Motivation 2.3.1 We shall consider the locations of $\mathbb{IMP}$ as being elements of some given set *Loc*. A state is given by specifying what data is held in the locations. For us, the data is simple and only consists of integers. Thus a state is a function from the set of locations to $\mathbb{Z}$. As we have mentioned, a configuration is given by a pair (e, s) where e is an

expression and s a state. We shall in fact define a binary relation on configurations which takes the form $(e_1\,,\,s_1) \rightsquigarrow (e_2\,,\,s_2)$. Its intended meaning is that the first configuration computes in one step or cycle to the second configuration. We shall also prove that the computation steps are deterministic.

Definitions 2.3.2 The set *States* of **states** is given by $[Loc, \mathbb{Z}]_{tot}$. If $s \in States$ and $x \in Loc$, we refer to $s(x)$ as "the integer held in x at state s". If $s \in States$, $x \in Loc$ and $n \in \mathbb{Z}$, then there is a state denoted by $s\{x \mapsto n\} : Loc \to \mathbb{Z}$ which is the function defined by

$$(s\{x \mapsto n\})(y) \stackrel{\text{def}}{=} \begin{cases} n & \text{if } y = x \\ s(y) & \text{otherwise} \end{cases}$$

for each $y \in Loc$. We say that state s is **updated** at x by n.

The elements of the set $Exp \times States$ will be known as **configurations**. We shall inductivley define a binary relation on $Exp \times States$ by the rules in Table 2.2, where we shall write $(e_1\,,\,s_1) \rightsquigarrow (e_2\,,\,s_2)$ instead of $((e_1\,,\,s_1), (e_2\,,\,s_2))$ `in` $\rightsquigarrow$. We call $\rightsquigarrow$ a **transition** relation, and any instance of a relationship in $\rightsquigarrow$ is called a **transition step**.

Proposition 2.3.3 The binary relation $\rightsquigarrow$ enjoys the following properties:

(i) For every transition $(e_1\,,\,s_1) \rightsquigarrow (e_2\,,\,s_2)$, then either $e_1, e_2 \in IExp$, or $e_1, e_2 \in BExp$, or $e_1, e_2 \in Com$; and

(ii) for every transition $(e_1\,,\,s_1) \rightsquigarrow (e_2\,,\,s_2)$, if neither e_1 nor e_2 is a command, then $s_1 = s_2$.

Proof Both parts follow by a simple Rule Induction for $\rightsquigarrow$. Details are left as an easy exercise. We shall soon give a very detailed example of a more complicated proof by Rule Induction. □

Example 2.3.4 Let us write co for while $x > \underline{0}$ do co' where co' is the command $y := y + \underline{2}\,;\, x := x - \underline{1}$. Suppose that s is a state for which $s(x) = 1$ and $s(y) = 0$. Let us write $s' \stackrel{\text{def}}{=} s\{y \mapsto 2\}$ and $s'' \stackrel{\text{def}}{=} s\{y \mapsto 2\}\{x \mapsto 0\} = s'\{x \mapsto 0\}$. We give an example of a sequence of configuration transitions for the language $\mathbb{IMP}$ in Figure 2.1. We also give, as an example, the deduction of the transition step $\rightsquigarrow_*$ in Figure 2.2. Of course, each of the transition steps given in Figure 2.1 have similar deductions to that for $\rightsquigarrow_*$, but in practice one can write down (correct) transition steps directly, without formal deduction trees, simply by understanding the intended meaning of the language $\mathbb{IMP}$.

Theorem 2.3.5 The operational semantics of $\mathbb{IMP}$, as specified by the transition relation $\rightsquigarrow$, is **deterministic**, that is to say that for all expressions e, e' and e'', and states s, s' and s'', if

$$(e\,,\,s) \rightsquigarrow (e'\,,\,s') \qquad \text{and} \qquad (e\,,\,s) \rightsquigarrow (e''\,,\,s'')$$

then $e' = e''$ and $s' = s''$.

Rules for integer and Boolean Expressions

$$\frac{}{(x\,,\,s) \rightsquigarrow (s(x)\,,\,s)} \rightsquigarrow\text{LOC} \qquad \frac{(ie_1\,,\,s) \rightsquigarrow (ie_2\,,\,s)}{(ie_1\ op\ ie\,,\,s) \rightsquigarrow (ie_2\ op\ ie\,,\,s)} \rightsquigarrow\text{OP}_1$$

$$\frac{(ie_1\,,\,s) \rightsquigarrow (ie_2\,,\,s)}{(\underline{n}\ op\ ie_1\,,\,s) \rightsquigarrow (\underline{n}\ op\ ie_2\,,\,s)} \rightsquigarrow\text{OP}_2 \qquad \frac{}{(\underline{n_1}\ op\ \underline{n_2}\,,\,s) \rightsquigarrow (\underline{n_1\ op\ n_2}\,,\,s)} \rightsquigarrow\text{OP}_3$$

Rules for Commands

$$\frac{(ie_1\,,\,s) \rightsquigarrow (ie_2\,,\,s)}{(x := ie_1\,,\,s) \rightsquigarrow (x := ie_2\,,\,s)} \rightsquigarrow\text{ASS}_1 \qquad \frac{}{(x := \underline{n}\,,\,s) \rightsquigarrow (\mathsf{skip}\,,\,s\{x \mapsto n\})} \rightsquigarrow\text{ASS}_2$$

$$\frac{(co_1\,,\,s_1) \rightsquigarrow (co_2\,,\,s_2)}{(co_1\ ;\ co\,,\,s_1) \rightsquigarrow (co_2\ ;\ co\,,\,s_2)} \rightsquigarrow\text{SEQ}_1 \qquad \frac{}{(\mathsf{skip}\ ;\ co\,,\,s) \rightsquigarrow (co\,,\,s)} \rightsquigarrow\text{SEQ}_2$$

$$\frac{(be_1\,,\,s) \rightsquigarrow (be_2\,,\,s)}{(\mathsf{if}\ be_1\ \mathsf{then}\ co_1\ \mathsf{else}\ co_2\,,\,s) \rightsquigarrow (\mathsf{if}\ be_2\ \mathsf{then}\ co_1\ \mathsf{else}\ co_2\,,\,s)} \rightsquigarrow\text{COND}_1$$

$$\frac{}{(\mathsf{if}\ \underline{T}\ \mathsf{then}\ co_1\ \mathsf{else}\ co_2\,,\,s) \rightsquigarrow (co_1\,,\,s)} \rightsquigarrow\text{COND}_2$$

$$\frac{}{(\mathsf{if}\ \underline{F}\ \mathsf{then}\ co_1\ \mathsf{else}\ co_2\,,\,s) \rightsquigarrow (co_2\,,\,s)} \rightsquigarrow\text{COND}_3$$

$$\frac{}{(\mathsf{while}\ be\ \mathsf{do}\ co\,,\,s) \rightsquigarrow (\mathsf{if}\ be\ \mathsf{then}\ (co\ ;\ \mathsf{while}\ be\ \mathsf{do}\ co)\ \mathsf{else}\ \mathsf{skip}\,,\,s)} \rightsquigarrow\text{LOOP}$$

Table 2.2: Configuration Transitions $(e\,,\,s) \rightsquigarrow (e'\,,\,s')$ in $\mathbb{IMP}$

$$
\begin{array}{rcl}
(co\,,\,s) & \rightsquigarrow & (\mathsf{if}\ x > \underline{0}\ \mathsf{then}\ co'\ ;\ co\ \mathsf{else}\ \mathsf{skip}\,,\,s) \\
& \rightsquigarrow & (\mathsf{if}\ \underline{1} > \underline{0}\ \mathsf{then}\ co'\ ;\ co\ \mathsf{else}\ \mathsf{skip}\,,\,s) \\
& \rightsquigarrow & (\mathsf{if}\ \underline{T}\ \mathsf{then}\ co'\ ;\ co\ \mathsf{else}\ \mathsf{skip}\,,\,s) \\
& \rightsquigarrow & (co'\ ;\ co\,,\,s) \\
& \rightsquigarrow & ((y := \underline{0} + \underline{2}\ ;\ x := x - \underline{1})\ ;\ co\,,\,s) \\
& \rightsquigarrow & ((y := \underline{2}\ ;\ x := x - \underline{1})\ ;\ co\,,\,s) \\
& \rightsquigarrow_* & ((\mathsf{skip}\ ;\ x := x - \underline{1})\ ;\ co\,,\,s') \\
& \rightsquigarrow & (x := x - \underline{1}\ ;\ co\,,\,s') \\
& \rightsquigarrow & (x := \underline{1} - \underline{1}\ ;\ co\,,\,s') \\
& \rightsquigarrow & (x := \underline{0}\ ;\ co\,,\,s') \\
& \rightsquigarrow & (\mathsf{skip}\ ;\ co\,,\,s'') \\
& \rightsquigarrow & (co\,,\,s'') \\
& \rightsquigarrow & (\mathsf{if}\ x > \underline{0}\ \mathsf{then}\ co'\ ;\ co\ \mathsf{else}\ \mathsf{skip}\,,\,s'') \\
& \rightsquigarrow & (\mathsf{if}\ \underline{0} > \underline{0}\ \mathsf{then}\ co'\ ;\ co\ \mathsf{else}\ \mathsf{skip}\,,\,s'') \\
& \rightsquigarrow & (\mathsf{if}\ \underline{F}\ \mathsf{then}\ co'\ ;\ co\ \mathsf{else}\ \mathsf{skip}\,,\,s'') \\
& \rightsquigarrow & (\mathsf{skip}\,,\,s'')
\end{array}
$$

Figure 2.1: A Transition Sequence in $\mathbb{IMP}$

$$
\dfrac{\dfrac{\dfrac{}{(y := \underline{2}\,,\,s) \rightsquigarrow (\mathsf{skip}\,,\,s')}\ {\rightsquigarrow}\mathrm{ASS}_2}{(y := \underline{2}\ ;\ x := x - \underline{1}\,,\,s) \rightsquigarrow (\mathsf{skip}\ ;\ x := x - \underline{1}\,,\,s')}\ {\rightsquigarrow}\mathrm{SEQ}_1}{((y := \underline{2}\ ;\ x := x - \underline{1})\ ;\ co\,,\,s) \rightsquigarrow ((\mathsf{skip}\ ;\ x := x - \underline{1})\ ;\ co\,,\,s')}\ {\rightsquigarrow}\mathrm{SEQ}_1
$$

Figure 2.2: An Example Deduction for $\rightsquigarrow$

Proof We can prove this result by Rule Induction. If we write

$$Prop(((e\,,\,s),(e'\,,\,s'))) \stackrel{\text{def}}{=} \text{ for all } (e''\,,\,s''),\quad (e\,,\,s) \rightsquigarrow (e''\,,\,s'') \Longrightarrow (e''=e' \text{ and } s''=s')$$

then we can prove that

$$\text{for all } (e\,,\,s) \rightsquigarrow (e'\,,\,s'),\quad Prop(((e\,,\,s),(e'\,,\,s'))) \qquad (*)$$

holds by using Rule Induction, and this latter statement is equivalent to the statement of the theorem.

We consider property closure for just one rule, say

$$\frac{(ie_1\,,\,s) \rightsquigarrow (ie_2\,,\,s)}{(x := ie_1\,,\,s) \rightsquigarrow (x := ie_2\,,\,s)} \rightsquigarrow_{\text{ASS}_1}$$

The inductive hypothesis is $Prop(((ie_1\,,\,s),(ie_2\,,\,s)))$, that is

$$\text{for all } (y\,,\,z),\quad (ie_1\,,\,s) \rightsquigarrow (y\,,\,z) \Longrightarrow (y = ie_2 \text{ and } z = s) \qquad (IH)$$

We need to prove $Prop(((x := ie_1\,,\,s),(x := ie_2\,,\,s)))$, that is

$$\text{for all } (u\,,\,v),\quad (x := ie_1\,,\,s) \rightsquigarrow (u\,,\,v) \Longrightarrow (u = (x := ie_2) \text{ and } v = s) \qquad (C)$$

To prove (C) we choose an arbitrary configuration $(e'\,,\,s')$ and suppose that $(x{:=}ie_1\,,\,s) \rightsquigarrow (e'\,,\,s')$. This could only be deduced from $\rightsquigarrow_{\text{ASS}_1}$ and so $e' = (x := ie_3)$ and $s' = s$ for some ie_3, where

$$(ie_1\,,\,s) \rightsquigarrow (ie_3\,,\,s).$$

Hence using (IH) we can deduce $ie_3 = ie_2$ (and $s = s$!). Thus $e' = (x := ie_2)$ and we already showed that $s' = s$. As $(e'\,,\,s')$ was arbitrary, we have proved (C).

Checking property closure of the remaining rules is left as an easy exercise. □

⋙ **Warning 2.3.6** ***You may care to compare carefully the above proof with the general exposition of Rule Induction. Note that*** $(*)$ ***is***

$$\text{for all } \underbrace{((e\,,\,s),(e'\,,\,s'))}_{i} \in \underbrace{\rightsquigarrow}_{I},\ \underbrace{Prop(((e\,,\,s),(e'\,,\,s')))}_{Prop(i)}$$

Now, $Prop(i)$ ***is a mathematical statement involving*** i***. Precisely what is it? It is in fact***

$$\text{for all } (e''\,,\,s''),\quad fst(i) \rightsquigarrow (e''\,,\,s'') \Longrightarrow e'' = fst(snd(i)) \text{ and } s'' = snd(snd(i))$$

where fst ***and*** snd ***are functions which extract the first or second coordinates of a pair.***

$$States \stackrel{\text{def}}{=} [Loc, \mathbb{Z}]_{tot}$$

$$\mathcal{I}\colon IExp \longrightarrow [States, \mathbb{Z}]_{tot}$$

$$\mathcal{B}\colon BExp \longrightarrow [States, \mathbb{B}]_{tot}$$

$$\mathcal{C}\colon Com \longrightarrow [States, States]_{par}$$

$\mathcal{I}(ie)(s) = n$ where $n \in \mathbb{Z}$ is the unique integer such that $(ie\,,\,s) \rightsquigarrow^* (\underline{n}\,,\,s)$

$\mathcal{B}(be)(s) = b$ where $b \in \mathbb{B}$ is the unique Boolean such that $(be\,,\,s) \rightsquigarrow^* (\underline{b}\,,\,s)$

$$\mathcal{C}(co)(s) \stackrel{\text{def}}{=} \begin{cases} \text{unique } s' \text{ such that } (co\,,\,s) \rightsquigarrow^* (\mathsf{skip}\,,\,s') \text{ if } s' \text{ exists} \\ \text{undefined otherwise} \end{cases}$$

Table 2.3: Evaluation Functions $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{C}$ for $\mathbb{IMP}$

Definitions 2.3.7 We say that a configuration $(e\,,\,s)$ is **terminal** if there is no configuration $(e'\,,\,s')$ for which $(e\,,\,s) \rightsquigarrow (e'\,,\,s')$. One can see from the rules which define $\rightsquigarrow$ that the terminal configurations are $(\underline{c}\,,\,s)$ where $c \in \mathbb{Z} \cup \mathbb{B}$ is any integer or Boolean and s is any state, and $(\mathsf{skip}\,,\,s)$. An **infinite transition sequence** for a configuration $(e\,,\,s)$ takes the form

$$(e\,,\,s) \rightsquigarrow (e_1\,,\,s_1) \rightsquigarrow (e_2\,,\,s_2) \rightsquigarrow \ldots \rightsquigarrow (e_i\,,\,s_i) \rightsquigarrow \ldots$$

where no configuration $(e_i\,,\,s_i)$ is terminal. A **finite transition sequence** for a configuration $(e\,,\,s)$ takes the form

$$(e\,,\,s) \rightsquigarrow (e_1\,,\,s_1) \rightsquigarrow (e_2\,,\,s_2) \rightsquigarrow \ldots \rightsquigarrow (e_m\,,\,s_m)$$

where $(e_m\,,\,s_m)$ is terminal. You should note that Theorem 2.3.5 implies that each non-terminal configuration $(e\,,\,s)$ has a *unique* transition sequence which is either finite or infinite. We shall write $\rightsquigarrow^*$ for the reflexive, transitive closure of $\rightsquigarrow$.

If it is the case that $e \in IExp \cup BExp$, then for any state s we can prove that $(e\,,\,s)$ has a finite transition sequence. (This can be proved using Rule Induction—try it as an exercise.) Thus there are (well defined) functions $\mathcal{I}$ and $\mathcal{B}$ as detailed in Table 2.3: for example, for any integer expression ie, there is a total function $\mathcal{I}(ie) : States \to \mathbb{Z}$ whose value $\mathcal{I}(ie)(s)$ at any state s is given in the Table. In the case that $co \in Com$, then for any state s it is *possible* $(co\,,\,s) \rightsquigarrow (\mathsf{skip}\,,\,s')$ for some state s'. If so, by Theorem 2.3.5 we know that s' is the unique state for which the latter relationship holds. So, for any given co, there are certain states s for which we can produce a unique s' which depends on co and s. Thus there is a function $\mathcal{C}$ as detailed in Table 2.3, where for each command co, $\mathcal{C}(co)$ is a partial function from states to states.

Examples 2.3.8

(1) Let co be $\mathsf{while}\ \underline{T}\ \mathsf{do\ skip}$. Then we have

$$\begin{array}{rcl}(co\,,\,s) & \rightsquigarrow & (\mathsf{if}\ \underline{T}\ \mathsf{then}\ (\mathsf{skip}\ ;\ co)\ \mathsf{else\ skip}\,,\,s)\\ & \rightsquigarrow & (\mathsf{skip}\ ;\ co\,,\,s)\\ & \rightsquigarrow & (co\,,\,s)\\ & \rightsquigarrow & \ldots\end{array}$$

and this cycle repeats forever. Thus $(co\,,\,s)$ has an infinite transition sequence, and note that as s was arbitrary, $\mathcal{C}(co) : States \rightharpoonup States$ is undefined on all states, that is $\mathcal{C}(co) = \varnothing$.

(2) Suppose that co is the command

$$y := \underline{1}\ ;\ (\mathsf{while}\ x > \underline{1}\ \mathsf{do}\ (y := x * y\ ;\ x := x - \underline{1})).$$

Then $\mathcal{C}(co) : States \rightharpoonup States$ is the (total) function defined by

$$\mathcal{C}(co)(s) \stackrel{\text{def}}{=} \begin{cases} s\{y \mapsto n!\}\{x \mapsto 1\} & \text{if } n > 1\\ s\{y \mapsto 1\} & \text{if } n \leq 1\end{cases}$$

where $n \stackrel{\text{def}}{=} s(x)$.

Remark 2.3.9 Clearly $\mathbb{IMP}$ is not a particulary useful or practical programming language. But it is Turing powerful in the usual sense. Fix a pair of locations x and y. If $f : \mathbb{N} \to \mathbb{N}$ is any partial recursive function, then we can find a command co such that for each $n \in \mathbb{N}$, if state s satisfies $n = s(x)$ then

$$\mathcal{C}(co)(s) \text{ is defined} \iff f(n) \text{ is defined}$$

and when they are both defined, $\mathcal{C}(co)(s)(y) = f(n)$.

2.4 An Evaluation Relation

Motivation 2.4.1 We shall now describe an operational semantics for $\mathbb{IMP}$ which, in the case of integer expressions, specifies how such expressions can compute to integers. The operational semantics has assertions which look like $(ie\,,\,s) \Downarrow_{IExp} \underline{n}$. The idea is that such an assertion corresponds to the configuration $(ie\,,\,s)$ making a finite number of transition steps to the configuration $(\underline{n}\,,\,s)$. A similar idea applies to Boolean expressions and commands. In Theorem 2.4.4, we clarify these intuitive ideas precisely.

Definitions 2.4.2 We shall inductively define the sets $\Downarrow_{IExp}$, $\Downarrow_{BExp}$ and $\Downarrow_{Com}$. These sets are in fact ternary relations of the following form:

$$\begin{array}{rcl}\Downarrow_{IExp} & \subseteq & IExp \times States \times \mathbb{Z}\\ \Downarrow_{BExp} & \subseteq & BExp \times States \times \mathbb{B}\\ \Downarrow_{Com} & \subseteq & Com \times States \times States\end{array}$$

These sets are defined by the rules in Table 2.4. We shall use the following conventions:

$$\frac{}{(x\ ,\ s) \Downarrow_{IExp} \underline{s(x)}} \Downarrow\text{LOC} \qquad \frac{}{(\underline{c}\ ,\ s) \Downarrow_{IExp} \underline{c}} [\text{ where } c \in \mathbb{Z} \cup \mathbb{B}] \quad \Downarrow \text{CONST}$$

$$\frac{(ie_1\ ,\ s) \Downarrow_{IExp} \underline{n_1} \quad (ie_2\ ,\ s) \Downarrow_{IExp} \underline{n_2}}{(ie_1\ op\ ie_2\ ,\ s) \Downarrow_{IExp} \underline{n_1\ op\ n_2}} [\text{ where } op \in IOpr] \quad \Downarrow\text{OP}_1$$

$$\frac{(ie_1\ ,\ s) \Downarrow_{IExp} \underline{n_1} \quad (ie_2\ ,\ s) \Downarrow_{IExp} \underline{n_2}}{(ie_1\ op\ ie_2\ ,\ s) \Downarrow_{BExp} \underline{n_1\ op\ n_2}} [\text{ where } op \in BOpr] \quad \Downarrow\text{OP}_2$$

$$\frac{}{(\mathsf{skip}\ ,\ s) \Downarrow_{Com} s} \Downarrow\text{SKIP} \qquad \frac{(ie\ ,\ s) \Downarrow_{IExp} \underline{n}}{(x := ie\ ,\ s) \Downarrow_{Com} s\{x \mapsto n\}} \Downarrow\text{ASS}$$

$$\frac{(co_1\ ,\ s_1) \Downarrow_{Com} s_2 \quad (co_2\ ,\ s_2) \Downarrow_{Com} s_3}{(co_1\ ;\ co_2\ ,\ s_1) \Downarrow_{Com} s_3} \Downarrow\text{SEQ} \qquad \frac{(be\ ,\ s_1) \Downarrow_{BExp} \underline{T} \quad (co_1\ ,\ s_1) \Downarrow_{Com} s_2}{(\mathsf{if}\ be\ \mathsf{then}\ co_1\ \mathsf{else}\ co_2\ ,\ s_1) \Downarrow_{Com} s_2} \Downarrow\text{COND}_1$$

$$\frac{(be\ ,\ s_1) \Downarrow_{BExp} \underline{F} \quad (co_2\ ,\ s_1) \Downarrow_{Com} s_2}{(\mathsf{if}\ be\ \mathsf{then}\ co_1\ \mathsf{else}\ co_2\ ,\ s_1) \Downarrow_{Com} s_2} \Downarrow\text{COND}_2$$

$$\frac{(be\ ,\ s_1) \Downarrow_{BExp} \underline{T} \quad (co\ ,\ s_1) \Downarrow_{Com} s_2 \quad (\mathsf{while}\ be\ \mathsf{do}\ co\ ,\ s_2) \Downarrow_{Com} s_3}{(\mathsf{while}\ be\ \mathsf{do}\ co\ ,\ s_1) \Downarrow_{Com} s_3} \Downarrow\text{LOOP}_1$$

$$\frac{(be\ ,\ s) \Downarrow_{BExp} \underline{F}}{(\mathsf{while}\ be\ \mathsf{do}\ co\ ,\ s) \Downarrow_{Com} s} \Downarrow\text{LOOP}_2$$

Table 2.4: Evaluation Relation $(e\ ,\ s) \Downarrow s'$ in $\mathbb{IMP}$

We write $(ie\ ,\ s) \Downarrow_{IExp} \underline{n}$ instead of (ie, s, n) `in` $\Downarrow_{IExp}$

We write $(be\ ,\ s) \Downarrow_{BExp} \underline{b}$ instead of (be, s, b) `in` $\Downarrow_{BExp}$

We write $(co\ ,\ s_1) \Downarrow_{Com} s_2$ instead of (co, s_1, s_2) `in` $\Downarrow_{Com}$

Example 2.4.3 Let us write co for $\mathsf{while}\ x > \underline{0}\ \mathsf{do}\ co'$ where co' is the command $y := y + \underline{2}\ ;\ x := x - \underline{1}$. Suppose that s is a state for which $s(x) = 1$ and $s(y) = 0$. A proof of $(co\ ,\ s) \Downarrow s\{y \mapsto 2\}\{x \mapsto 0\}$ is given in Figure 2.3. It is an exercise to add in the appropriate labels to the deduction tree, and to fill in T.

Theorem 2.4.4 For any $ie \in IExp$, $be \in BExp$, $co \in Com$, $s, s' \in States$, $n \in \mathbb{Z}$ and

$$\frac{P_1 \qquad P_2 \qquad P_3}{(co\ ,\ s) \Downarrow s\{y\mapsto 2\}\{x\mapsto 0\}}$$

where P_1 is

$$\frac{\dfrac{}{(x\ ,\ s) \Downarrow \underline{1}} \qquad \dfrac{}{(\underline{0}\ ,\ s) \Downarrow \underline{0}}}{(x > \underline{0}\ ,\ s) \Downarrow \underline{T}}$$

and P_2 is

$$\frac{\dfrac{\dfrac{\dfrac{}{(y\ ,\ s) \Downarrow \underline{0}} \qquad \dfrac{}{(\underline{2}\ ,\ s) \Downarrow \underline{2}}}{(y + \underline{2}\ ,\ s) \Downarrow \underline{2}}}{(y := y + \underline{2}\ ,\ s) \Downarrow s\{y\mapsto 2\}} \qquad \dfrac{T}{(x := x - \underline{1}\ ,\ s\{y\mapsto 2\}) \Downarrow s\{y\mapsto 2\}\{x\mapsto 0\}}}{(y := y + \underline{2}\ ;\ x := x - \underline{1}\ ,\ s) \Downarrow s\{y\mapsto 2\}\{x\mapsto 0\}}$$

and P_3 is

$$\frac{\dfrac{\dfrac{}{(x\ ,\ s\{y\mapsto 2\}\{x\mapsto 0\}) \Downarrow \underline{0}} \qquad \dfrac{}{(\underline{0}\ ,\ s\{y\mapsto 2\}\{x\mapsto 0\}) \Downarrow \underline{0}}}{(x > \underline{0}\ ,\ s\{y\mapsto 2\}\{x\mapsto 0\}) \Downarrow \underline{F}}}{(co\ ,\ s\{y\mapsto 2\}\{x\mapsto 0\}) \Downarrow s\{y\mapsto 2\}\{x\mapsto 0\}}$$

Figure 2.3: An Example Deduction of an Evaluation

$b \in \mathbb{B}$ we have that

$$(ie\,,\,s) \rightsquigarrow^* (\underline{n}\,,\,s) \iff (ie\,,\,s) \Downarrow_{IExp} \underline{n}$$
$$(be\,,\,s) \rightsquigarrow^* (\underline{b}\,,\,s) \iff (be\,,\,s) \Downarrow_{BExp} \underline{b}$$
$$(co\,,\,s) \rightsquigarrow^* (\mathsf{skip}\,,\,s') \iff (co\,,\,s) \Downarrow_{Com} s'$$

where $\rightsquigarrow^*$ denotes reflexive, transitive closure of $\rightsquigarrow$.

Proof We break the proof into three parts:

(a) Prove the right to left implications by Rule Induction.

(b) Prove by Rule Induction for $\rightsquigarrow$ that

$$(ie\,,\,s) \rightsquigarrow (ie'\,,\,s) \text{ and } (ie'\,,\,s) \Downarrow \underline{n} \implies (ie\,,\,s) \Downarrow \underline{n}$$
$$(be\,,\,s) \rightsquigarrow (be'\,,\,s) \text{ and } (be'\,,\,s) \Downarrow \underline{b} \implies (be\,,\,s) \Downarrow \underline{b}$$
$$(co\,,\,s) \rightsquigarrow (co'\,,\,s') \text{ and } (co'\,,\,s') \Downarrow s'' \implies (co\,,\,s) \Downarrow s''$$

(c) Use (b) to deduce the left to right implications.

(a) We shall prove by Rule Induction that

$$\text{for all } (ie\,,\,s) \Downarrow_{IExp} \underline{n}, \quad (ie\,,\,s) \rightsquigarrow^* (\underline{n}\,,\,s)$$
$$\text{for all } (be\,,\,s) \Downarrow_{BExp} \underline{b}, \quad (be\,,\,s) \rightsquigarrow^* (\underline{b}\,,\,s)$$
$$\text{for all } (co\,,\,s) \Downarrow_{Com} s', \quad (co\,,\,s) \rightsquigarrow^* (\mathsf{skip}\,,\,s')$$

We shall just check the property closure of rule ($\Downarrow$ LOOP$_1$). Suppose that the appropriate properties hold of the hypotheses, that is we have

$$(be\,,\,s_1) \rightsquigarrow^* (\underline{T}\,,\,s_1) \qquad (H1)$$
$$(co\,,\,s_1) \rightsquigarrow^* (\mathsf{skip}\,,\,s_2) \qquad (H2)$$
$$(\mathsf{while}\ be\ \mathsf{do}\ co\,,\,s_2) \rightsquigarrow^* (\mathsf{skip}\,,\,s_3) \qquad (H3)$$

We need to prove that

$$(\mathsf{while}\ be\ \mathsf{do}\ co\,,\,s_1) \rightsquigarrow^* (\mathsf{skip}\,,\,s_3) \qquad (C)$$

Let us write co_1 for $\mathsf{while}\ be\ \mathsf{do}\ co$. Then

$(co_1\,,\,s_1)$	$\rightsquigarrow$	$(\mathsf{if}\ be\ \mathsf{then}\ co\ ;\ co_1\ \mathsf{else\ skip}\,,\,s_1)$	($\rightsquigarrow$ LOOP)
	$\rightsquigarrow^*$	$(\mathsf{if}\ \underline{T}\ \mathsf{then}\ co\ ;\ co_1\ \mathsf{else\ skip}\,,\,s_1)$	$(H1)$ and several uses of ($\rightsquigarrow$ COND$_1$)
	$\rightsquigarrow$	$(co\ ;\ co_1\,,\,s_1)$	($\rightsquigarrow$ COND$_2$)
	$\rightsquigarrow^*$	$(\mathsf{skip}\ ;\ co_1\,,\,s_2)$	$(H2)$ and several uses of ($\rightsquigarrow$ SEQ$_1$)
	$\rightsquigarrow$	$(co_1\,,\,s_2)$	($\rightsquigarrow$ SEQ$_2$)
	$\rightsquigarrow^*$	$(\mathsf{skip}\,,\,s_3)$	$(H3)$

which proves (C).

(b) We shall define a binary relation on the set $Exp \times States$ of configurations with relationships denoted by

$$(e\,,\,s) \twoheadrightarrow (e'\,,\,s')$$

which hold if and only if either

$$e, e' \in IExp, \quad s = s' \quad \text{and} \quad \text{for all } n \in \mathbb{Z}, (e'\,,\,s) \Downarrow_{IExp} \underline{n} \Longrightarrow (e\,,\,s) \Downarrow_{IExp} \underline{n}$$

or

$$e, e' \in BExp, \quad s = s' \quad \text{and} \quad \text{for all } b \in \mathbb{B}, (e'\,,\,s) \Downarrow_{BExp} \underline{b} \Longrightarrow (e\,,\,s) \Downarrow_{BExp} \underline{b}$$

or

$$e, e' \in Com, \quad \text{and} \quad \text{for all } s'' \in States, (e'\,,\,s') \Downarrow_{Com} s'' \Longrightarrow (e\,,\,s) \Downarrow_{Com} s''.$$

Then proving (b) is equivalent to proving that

$$\text{for all } (e\,,\,s) \rightsquigarrow (e'\,,\,s'), \quad (e\,,\,s) \twoheadrightarrow (e'\,,\,s')$$

which we can show by Rule Induction for the set $\rightsquigarrow$. Let us just consider property closure for the rule ($\rightsquigarrow$ LOOP). We have to prove that

$$(co_1\,,\,s) \twoheadrightarrow (\mathsf{if}\ be\ \mathsf{then}\ (co\ ;\ co_1)\ \mathsf{else\ skip}\,,\,s)$$

where $co_1 \stackrel{\text{def}}{=} \mathsf{while}\ be\ \mathsf{do}\ co$, that is for all states s'' if

$$(\mathsf{if}\ be\ \mathsf{then}\ (co\ ;\ co_1)\ \mathsf{else\ skip}\,,\,s) \Downarrow s'' \tag{1}$$

then

$$(co_1\,,\,s) \Downarrow s'' \tag{2}$$

But (1) can hold only if it has been deduced either from ($\Downarrow$ COND$_1$) or ($\Downarrow$ COND$_2$). We consider the two cases:

(*Case* ($\Downarrow$ COND$_1$)): (1) was deduced from the hypotheses

$$(be\,,\,s) \Downarrow \underline{T} \tag{3}$$

and

$$(co\ ;\ co_1\,,\,s) \Downarrow s'' \tag{4}$$

where the latter assertion is deduced using ($\Downarrow$ SEQ) from the hypotheses

$$(co\,,\,s) \Downarrow s' \tag{5}$$

and

$$(co_1\,,\,s') \Downarrow s'' \tag{6}$$

for some state s'. If we apply ($\Downarrow$ LOOP$_1$) to (3), (5) and (6) we obtain (2).

$(Case\ (\Downarrow \text{COND}_2))$: (1) was deduced from the hypotheses

$$(be\,,\,s) \Downarrow \underline{F} \tag{7}$$

and

$$(\mathsf{skip}\,,\,s) \Downarrow s'' \tag{8}$$

But (8) can only be deduced using $(\Downarrow \text{SKIP})$ so that $s = s''$ and then $(\Downarrow \text{LOOP}_2)$ applied to (7) yields (2) as required.

(c) It is easy to see that the relation $\twoheadrightarrow$ defined in the proof of (b) is reflexive and transitive; hence since it contains $\rightsquigarrow$ (which is exactly what we proved in (b)) it also contains $\rightsquigarrow^*$, that is

$$\text{for all } (e\,,\,s) \rightsquigarrow^* (e'\,,\,s'), \quad (e\,,\,s) \twoheadrightarrow (e'\,,\,s').$$

Thus if $(ie\,,\,s) \rightsquigarrow^* (\underline{n}\,,\,s)$ then we have $(ie\,,\,s) \twoheadrightarrow (\underline{n}\,,\,s)$ and hence by the definition of $\twoheadrightarrow$

$$\text{for all } m \in \mathbb{Z}, \quad (\underline{n}\,,\,s) \Downarrow \underline{m} \Longrightarrow (ie\,,\,s) \Downarrow \underline{m}$$

Taking $n = m$ and using $(\Downarrow \text{CONST})$ gives $(ie\,,\,s) \Downarrow \underline{n}$, as required. This shows the first left to right implication. The remaining two implications are similar.

□

2.5 Semantic Equivalence

Motivation 2.5.1 We shall consider two commands to have the same meaning, that is to be *semantically equivalent*, if they have the same affect on any arbitrary state. This rather rough and ready idea is made precise in the definitions which follow:

Definitions 2.5.2 We say that two $\mathbb{IMP}$ commands co_1 and co_2 are **semantically equivalent** if for all states s and s'

$$\mathcal{C}(co_1)(s) \text{ is defined and equal to } s' \quad \Longleftrightarrow \quad \mathcal{C}(co_2)(s) \text{ is defined and equal to } s'$$

and when this happens we write $co_1 \sim co_2$. It is a corollary of Theorem 2.4.4 that we have $co_1 \sim co_2$ if and only if

$$\text{for all } s, s' \in States, \quad (co_1\,,\,s) \Downarrow s' \Longleftrightarrow (co_2\,,\,s) \Downarrow s'.$$

Example 2.5.3 Let us prove that the two commands

$$(\mathsf{if}\ be\ \mathsf{then}\ co\ \mathsf{else}\ co')\ ;\ co'' \quad \text{and} \quad \mathsf{if}\ be\ \mathsf{then}\ (co\ ;\ co'')\ \mathsf{else}\ (co'\ ;\ co'')$$

are semantically equivalent.

Let us write co_1 for the first command, and co_2 for the second. First, suppose that we have

$$(co_1\,,\,s) \Downarrow s' \tag{1}$$

Then this can only have been deduced from ($\Downarrow$ SEQ) with the hypotheses

$$(\mathsf{if}\ be\ \mathsf{then}\ co\ \mathsf{else}\ co'\ ,\ s) \Downarrow s'' \tag{2}$$

and

$$(co''\ ,\ s'') \Downarrow s' \tag{3}$$

for some state s''. The rule used to deduce (2) must have been either ($\Downarrow$ COND$_1$) or ($\Downarrow$ COND$_2$) so that either

$$(be\ ,\ s) \Downarrow \underline{T} \quad \text{and} \quad (co\ ,\ s) \Downarrow s'' \tag{4}$$

or

$$(be\ ,\ s) \Downarrow \underline{F} \quad \text{and} \quad (co'\ ,\ s) \Downarrow s'' \tag{5}$$

In the first case, if we apply ($\Downarrow$ SEQ) to (3) and (4) we get

$$(be\ ,\ s) \Downarrow \underline{T} \quad \text{and} \quad (co\ ;\ co''\ ,\ s) \Downarrow s' \tag{6}$$

and in the second case, if we apply ($\Downarrow$ SEQ) to (3) and (5) we get

$$(be\ ,\ s) \Downarrow \underline{F} \quad \text{and} \quad (co'\ ;\ co''\ ,\ s) \Downarrow s' \tag{7}$$

If (6) holds, then using ($\Downarrow$ COND$_1$) we get $(co_2\ ,\ s) \Downarrow s'$. If (7) holds, then using ($\Downarrow$ COND$_2$) we also get $(co_2\ ,\ s) \Downarrow s'$, as required.

In a similar fashion, we can begin with $(co_2\ ,\ s) \Downarrow s'$ and deduce (1) so that $co_1 \sim co_2$ as required.

2.6 Command Contexts

Motivation 2.6.1 We pose a question which will be answered in the next chapter. Suppose that $co_1 \sim co_2$ and that $co[co_1]$ is a command with an occurrence of co_1 in it. If $co[co_2]$ denotes the command $co[co_1]$ with the command co_1 changed to co_2, then is it the case that $co[co_1] \sim co[co_2]$? We can make this precise by defining *command contexts* $\mathbb{C}$ which are basically commands co possibly containing “gaps in the code”. These gaps will be denoted by $\Box$. An example of a command context is $\mathsf{if}\ \underline{T}\ \mathsf{then}\ \Box\ \mathsf{else}\ x := \underline{2}$—code for the first branch of the conditional is missing.

Definitions 2.6.2 We inductively define a **command context** $\mathbb{C}$ using the grammar

$$\begin{array}{rcl} \mathbb{C} & ::= & \Box \\ & \mid & \mathsf{skip} \\ & \mid & x := ie \\ & \mid & \mathbb{C}\ ;\ \mathbb{C} \\ & \mid & \mathsf{if}\ be\ \mathsf{then}\ \mathbb{C}\ \mathsf{else}\ \mathbb{C} \\ & \mid & \mathsf{while}\ be\ \mathsf{do}\ \mathbb{C} \end{array}$$

Given any command co, we shall write $\mathbb{C}[co]$ to denote the command which results from replacing all occurrences of $\Box$ in $\mathbb{C}$ by co (we omit the formal recursive definition). Note that $\mathbb{C}[co]$ is *indeed* another command.

Theorem 2.6.3 For any commands co_1 and co_2, we have

$$co_1 \sim co_2 \quad \Longleftrightarrow \quad \text{for all command contexts } \mathbb{C},\ \mathbb{C}[co_1] \sim \mathbb{C}[co_2].$$

Proof

($\Longrightarrow$): See the end of Chapter 3.

($\Longleftarrow$): This is trivial: take $\mathbb{C}$ to be $\Box$. $\Box$

3

The Denotational Semantics of an Imperative Language

3.1 Introduction

Motivation 3.1.1 Recall the following functions, where $States \stackrel{\text{def}}{=} [Loc, \mathbb{Z}]_{tot}$:

$$\mathcal{I} : IExp \longrightarrow [States, \mathbb{Z}]_{tot}$$
$$\mathcal{B} : BExp \longrightarrow [States, \mathbb{B}]_{tot}$$
$$\mathcal{C} : Com \longrightarrow [States, States]_{par}$$

Each of these functions has a similar definition. In the case of $\mathcal{I}$, if ie is any integer expression and s is any state, then the integer $\mathcal{I}(ie)(s)$ is defined to be the unique n for which $(ie\,,\ s) \rightsquigarrow^* (\underline{n}\,,\ s)$. This, as we saw, is a perfectly sensible definition, but how can we calculate $\mathcal{I}(ie)(s)$ directly? If we were asked to prove that $\mathcal{I}(ie)(s) = n$, we can do this by using the rules defining $\rightsquigarrow$ to give a deduction of $(ie\,,\ s) \rightsquigarrow^* (\underline{n}\,,\ s)$, but this does not help us *calculate* n if we are just given ie and s. In fact this is a similar situation to the simple functional languages in MC208. One has an operational semantics defining $P \Downarrow V$ (where the latter relationship means that program P evaluates to a value V) but given any P the definition of $\Downarrow$ does not let us calculate V directly.

However, in this chapter we shall describe another semantics for $\mathbb{IMP}$ which goes some way towards allowing us to calculate (execute) $\mathbb{IMP}$ expressions directly. We introduce a **denotational semantics** which will provide a framework from which we can (almost) directly calculate the functions above.

A denotational semantics for a language can be thought of as a *mathematical model* of the language. It is intended to be far more abstract than an operational semantics, and the model attempts to free itself from the particular syntax and implementation details of the programming language. One pay off is that using a denotational semantics we can compare apparently quite distinct constructs from different programming languages—a comparison is much easier in the setting of an abstract mathematical model.

3.2 Preliminary Denotational Definitions

Motivation 3.2.1 We shall define functions

$$[\![-]\!] : IExp \longrightarrow [States, \mathbb{Z}]_{tot}$$
$$[\![-]\!] : BExp \longrightarrow [States, \mathbb{B}]_{tot}$$
$$[\![-]\!] : Com \longrightarrow [States, States]_{par}$$

which have an obvious computation algorithm, and for which we can prove that

$$[\![ie]\!] = \mathcal{I}(ie) \quad \text{and} \quad [\![be]\!] = \mathcal{B}(be) \quad \text{and} \quad [\![co]\!] = \mathcal{C}(co).$$

Note that it is traditional to use the same symbol $[\![-]\!]$ to denote (three) different functions; $[\![-]\!]$ is said to be *overloaded*. By "obvious computation algorithm" we mean that given any ie, be, co and s, it is clear how to calculate $[\![ie]\!](s)$, $[\![be]\!](s)$ and $[\![co]\!](s)$ directly. How might we define $[\![-]\!]$? Suppose that e is an expression. We know that e is a finite tree, and that it was constructed inductively. So e is of the form $C(e_1, \dots, e_n)$ where C is a constructor and the e_i are the immediate subtrees (subexpressions) of e. If we already knew the definitions of the $[\![e_i]\!]$ then it might be possible to define $[\![e]\!]$ in terms of the (already defined) $[\![e_i]\!]$. But this just amounts to a recursive definition of the function $[\![e]\!]$. This is preceisely how we formulate our denotational semantics—and the "computation algorithm" referred to above amounts to computing $[\![e]\!]$ recursively.

In fact it is very easy to give definitions of the first two functions, but the third will take some work. We shall have to introduce some mathematical machinery in order to allow us to define $[\![-]\!]$ on commands. Once the definitions of the machinery are complete, we shall give a full definition of $[\![-]\!]$.

Motivation 3.2.2 The function $\mathcal{C} : Com \to [States, States]_{par}$ involves dealing with *partial* functions between *States* and *States*. In order to specify such a partial function, say f, for any state s we have to either say that f is *undefined* at the state s (in which case we shall say informally that "$f(s)$ is undefined") or we have to say that f is *defined* at s and specify a unique state which we denote by $f(s)$ (in which case we say informally that "$f(s)$ is defined"). In fact we can give an alternative definition of the set of partial functions between states. By alternative, we mean that there is in fact a set of total functions which, in a precise sense, very much resembles the set of partial functions between states. This latter set of (total) functions is conceptually easier to deal with, but will still provide us with a perfectly good model for commands.

Basically the idea is this: We shall consider, just for the moment, any partial function $f : A \rightharpoonup B$ between sets A and B. We consider replacing the target set B by a set $B \cup \{ \perp \}$ where $\perp$ is distinct from the elements of B. Then the partial function $f : A \rightharpoonup B$ "corresponds" to a total function $\overline{f} : A \to B \cup \{ \perp \}$ where $\overline{f}(a) \stackrel{\text{def}}{=} f(a)$ if $f(a)$ is defined, and $\overline{f}(a) \stackrel{\text{def}}{=} \perp$ otherwise. Conversely, given a function $g : A \to B \cup \{ \perp \}$, it is clear that there is a partial function $\tilde{g} : A \rightharpoonup B$ (what is $\tilde{g}$?). It is easy to verify that $\tilde{\overline{f}} = f$ and $\overline{\tilde{g}} = g$ and thus there is a bijection

$$[A, B]_{par} \cong [A, B \cup \{ \perp \}]_{tot}$$

and the two sets (of functions) are "mathematically identical". We need to make this a little more precise for our current setting:

Definitions 3.2.3 We shall define a set

$$States_{\perp} \stackrel{\text{def}}{=} \{ [s] \mid s \in States \} \cup \{ \perp \}$$

where you should think of $[s]$ as a copy of s so that if $[s] = [s']$ then $s = s'$ for all states s and s', and $\bot$ is a fixed element such that $\bot \neq [s]$ for any state s. We call $\bot$ **bottom** or the **undefined** state. Note that this amounts to there being an injection

$$\iota : States \to States_\bot$$

where $\iota(s) \stackrel{\text{def}}{=} [s]$.

Proposition 3.2.4 There is a bijection

$$I : [States, States]_{par} \cong [States, States_\bot]_{tot}$$

where for $f \in [States, States]_{par}$ we define $I(f) \in [States, States_\bot]_{tot}$ by

$$I(f)(s) \stackrel{\text{def}}{=} \begin{cases} [f(s)] \text{ if } f(s) \text{ is defined} \\ \bot \text{ otherwise} \end{cases}$$

for any state s.

Proof We shall prove that I is injective, and leave surjectivity as an exercise. Suppose that $I(f) = I(f')$. We have to show that $f = f'$, that is if s is any state we need to see that either both $f(s)$ and $f'(s)$ are undefined or else both are defined and $f(s) = f'(s)$. By hypothesis we have $I(f)(s) = I(f')(s)$. We consider cases according to whether $I(f)(s)$ is bottom or not:

(*Case $I(f)(s)$ is bottom*): Clearly $I(f')(s)$ must also be bottom and thus neither $f(s)$ or $f'(s)$ is defined.

(*Case $I(f)(s)$ is $[f(s)]$*): We must have that $I(f')(s)$ is $[f'(s)]$ (why?) and thus $[f(s)] = [f'(s)]$. So $f(s) = f'(s)$ as required.

As s was assumed to denote any arbitrary state, we are done. □

Motivation 3.2.5 As we have said, the function $[\![-]\!] : Com \to [States, States]_{par}$ will be equal to $\mathcal{C}$, but is to have a definition which allows for direct calculation of $[\![co]\!](s)$ for any co and s. Now that we have seen that the sets $[States, States]_{par}$ and $[States, States_\bot]_{tot}$ are bijective, and thus essentially "mathematically identical", we shall try to define a function $[\![-]\!] : Com \to [States, States_\bot]_{tot}$. This will give us a perfectly good mathematical model. To do this we shall need some auxiliary definitions. We shall now give each of these, and then give our full definition of $[\![-]\!]$. We shall motivate each of the auxiliary definitions by seeing how they arise naturally when we consider what properties our denotational model of $\mathbb{IMP}$ should have.

λ-notation

Suppose that $f : X \to Y$ is any function which is given by a mapping $x \mapsto E(x)$ where $E(x) \in Y$ is an expression involving x. Then it is sometimes convenient to be able to refer to the function f and at the same time make its definition explicit. We write $\lambda_{x \in X}.E(x)$ for the function f. Thus, for example, $\lambda_{x \in \mathbb{N}}.x + 1 : \mathbb{N} \to \mathbb{N}$ is the function which maps any number to its successor; for example $(\lambda_{x \in \mathbb{N}}.x + 1)(4) = 5$.

Lifting

Motivation 3.2.6 What is $[\![co_1 \ ; \ co_2]\!]$? Let us suppose that we know the definitions of

$$[\![co_1]\!] : \mathit{States} \to \mathit{States}_\perp \quad \text{and} \quad [\![co_2]\!] : \mathit{States} \to \mathit{States}_\perp$$

Operationally, the intended meaning of $co_1 \ ; \ co_2$ is that the command takes any state s, performs command co_1 which either loops or produces a new state s', and then in the latter case performs command co_2 in state s'.

Hence $[\![co_1]\!](s)$ is either $\perp$ or is a new value state[1] $[s']$. Thus $[\![co_1 \ ; \ co_2]\!](s)$ should be $\perp$ if $[\![co_1]\!](s)$ is $\perp$, and otherwise should be $[\![co_2]\!](s')$. We can capture this neatly with the following definitions:

Definitions 3.2.7 Suppose that $g : \mathit{States} \to \mathit{States}_\perp$ and that $x \in \mathit{States}_\perp$. Then we define a function $g_\perp : \mathit{States}_\perp \to \mathit{States}_\perp$ by setting

$$g_\perp(x) \stackrel{\text{def}}{=} \begin{cases} g(s) \text{ if } x = [s] \text{ for some } s \in \mathit{States} \\ \perp \text{ otherwise} \end{cases}$$

where x is any element of $\mathit{States}_\perp$. Thus we could define

$$[\![co_1 \ ; \ co_2]\!] \stackrel{\text{def}}{=} \lambda_{s \in \mathit{States}}.[\![co_2]\!]_\perp([\![co_1]\!](s))$$

which states precisely the intended denotation described above.

Conditionals

Motivation 3.2.8 Let us think about $\mathsf{if}\ be\ \mathsf{then}\ co\ \mathsf{else}\ co'$. Given a state s, the latter expression computes co in state s if be computes to $\underline{T}$ in state s, or computes co' in state s if be computes to $\underline{F}$ in state s. Thus we might wish to say that $[\![\mathsf{if}\ be\ \mathsf{then}\ co\ \mathsf{else}\ co']\!](s)$ is $[\![co]\!](s)$ or $[\![co']\!](s)$ according to whether $[\![be]\!](s)$ is T or F.

Definitions 3.2.9 Given any $b \in \mathbb{B}$ and $x, x' \in \mathit{States}_\perp$ we define a function

$$\mathit{cond} : \mathbb{B} \times \mathit{States}_\perp \times \mathit{States}_\perp \longrightarrow \mathit{States}_\perp$$

by setting

$$\mathit{cond}\,(b, x, x') \stackrel{\text{def}}{=} \begin{cases} x & \text{if } b = T \\ x' & \text{if } b = F \end{cases}$$

We call this the **conditional** function for $\mathit{States}_\perp$. We can use this conditional function to define the semantics of if-then expressions as

$$[\![\mathsf{if}\ be\ \mathsf{then}\ co\ \mathsf{else}\ co']\!] \stackrel{\text{def}}{=} \lambda_{s \in \mathit{States}}.\mathit{cond}\,([\![be]\!](s), [\![co]\!](s), [\![co']\!](s))$$

assuming that, recursively, we already have definitions of $[\![be]\!]$, $[\![co]\!]$ and $[\![co']\!]$.

[1] We refer to elements $[s]$ of $\mathit{States}_\perp$ as **value** states.

Least Fixed Points

Motivation 3.2.10 Let us think about what $[\![\mathsf{while}\ be\ \mathsf{do}\ co]\!] : States \to States_\bot$ should be. Operationally, $\mathsf{while}\ be\ \mathsf{do}\ co$ executes in a state s by first evaluating be in s. If the result is $\underline{T}$, then the command $co\ ;\ \mathsf{while}\ be\ \mathsf{do}\ co$ is executed, and if the result is $\underline{F}$ then skip is executed (and in this case the final state is s). This means that in our model we should have

$$[\![\mathsf{while}\ be\ \mathsf{do}\ co]\!](s) = cond\,([\![be]\!](s), [\![co\ ;\ \mathsf{while}\ be\ \mathsf{do}\ co]\!](s), [s])$$

and writing f for $[\![\mathsf{while}\ be\ \mathsf{do}\ co]\!]$ this amounts to

$$f(s) = cond\,([\![be]\!](s), f_\bot([\![co]\!](s)), [s]).$$

Thus whatever f is, it must satisfy the latter equation. Hence $[\![\mathsf{while}\ be\ \mathsf{do}\ co]\!]$ should be an element $f \in [States, States_\bot]_{tot}$ for which $\Phi(f) = f$ where

$$\Phi : [States, States_\bot]_{tot} \longrightarrow [States, States_\bot]_{tot}$$

is the function given by

$$\lambda_{f \in [States, States_\bot]_{tot}}.\lambda_{s \in States}.cond\,([\![be]\!](s), f_\bot([\![co]\!](s)), [s]).$$

Recall that such an f satisfying $\Phi(f) = f$ is called a **fixpoint** of Φ.

We have a problem in that Φ may have more than one fixed point. Which one of them ought to be chosen for the semantics of our while expression? The answer is that it does not matter, provided we can produce a *useful* model of our programming language. One test of the usefulness of the denotational semantics is that, in some sense, it corresponds closely to the operational semantics. We shall now show how we can choose our fixpoint uniquely, and later show that we made a good choice in the sense just described. We make our choice by considering a partial order $\preceq$ on the set $[States, States_\bot]_{tot}$. We can then consider the set of prefixpoints of

$$\Phi : ([States, States_\bot]_{tot}, \preceq) \longrightarrow ([States, States_\bot]_{tot}, \preceq)$$

and choose the least one (assuming it exists) for our semantic definition. Let us define a suitable partial order.

Definitions 3.2.11 Recall that we have

$$I : [States, States]_{par} \cong [States, States_\bot]_{tot}$$

The set on the left hand side is by definition (see page 3) a subset of $\mathcal{P}(States \times States)$ and so we can consider relationships between its elements given by subset inclusion, say $f \subseteq f'$. A moments thought reveals that this means for all states s, if $f(s)$ is defined then so too is $f'(s)$ and $f(s) = f'(s)$—make sure you understand this!

We can then transport these (partial order) relationships across the bijection, so that for any

$$g, g' \in [States, States_\perp]_{tot},$$

we define the relationship $g \preceq g'$ to hold if and only if $f \subseteq f'$ where f and f' are the unique elements for which $I(f) = g$ and $I(f') = g'$. Thus $g \preceq g'$ if and only if

$$\text{for all } s \in States, \quad g(s) \neq \bot \Longrightarrow g(s) = g'(s).$$

With Φ defined as above, we have

Proposition 3.2.12 There is a function $\mathit{fix}(\Phi) : States \to States_\perp$ for which

(i) $\Phi(\mathit{fix}(\Phi)) = \mathit{fix}(\Phi)$, that is $\mathit{fix}(\Phi)$ is a fixpoint of Φ, and

(ii) for all $g : States \to States_\perp$, if $\Phi(g) \preceq g$ (that is g is a *prefixpoint* of Φ) then $\mathit{fix}(\Phi) \preceq g$ (that is $\mathit{fix}(\Phi)$ is a *least* prefixpoint of Φ).

Proof This is delayed until Chapter 5. □

3.3 Denotational Semantics

Definitions 3.3.1 We can now give the full definition of the denotational semantics of $\mathbb{IMP}$. We shall define the denotational functions

$$[\![-]\!] : IExp \longrightarrow [States, \mathbb{Z}]_{tot}$$

$$[\![-]\!] : BExp \longrightarrow [States, \mathbb{B}]_{tot}$$

$$[\![-]\!] : Com \longrightarrow [States, States_\perp]_{tot}$$

by recursion on the structure of e using the following clauses, where s is an arbitrary state:

- $[\![\underline{c}]\!] \stackrel{\text{def}}{=} \lambda_{s \in States}.c$;
- $[\![x]\!] \stackrel{\text{def}}{=} \lambda_{s \in States}.s(x)$;
- $[\![e_1 \ op \ e_2]\!] \stackrel{\text{def}}{=} \lambda_{s \in States}.[\![e_1]\!](s) \ op \ [\![e_2]\!](s)$;
- $[\![\mathsf{skip}]\!] \stackrel{\text{def}}{=} \lambda_{s \in States}.[s]$;
- $[\![x := ie]\!] \stackrel{\text{def}}{=} \lambda_{s \in States}.[s\{x \mapsto [\![ie]\!](s)\}]$;
- $[\![co \ ; \ co']\!] \stackrel{\text{def}}{=} \lambda_{s \in States}.[\![co']\!]_\perp([\![co]\!](s))$;
- $[\![\mathsf{if} \ be \ \mathsf{then} \ co \ \mathsf{else} \ co']\!] \stackrel{\text{def}}{=} \lambda_{s \in States}.cond\,([\![be]\!](s), [\![co]\!](s), [\![co']\!](s))$; and
- $[\![\mathsf{while} \ be \ \mathsf{do} \ co]\!] \stackrel{\text{def}}{=} \mathit{fix}(\Phi)$ where Φ is the function

$$\lambda_{g \in [States, States_\perp]_{tot}}.\lambda_{s \in States}.cond\,([\![be]\!](s), g_\perp([\![co]\!](s)), [s]).$$

Motivation 3.3.2 We shall show that there is a very precise correspondence between the operational and denotational semantics of $\mathbb{IMP}$. This correspondence is stated in the following theorem:

Theorem 3.3.3 For all $ie \in IExp$, $be \in BExp$, $co \in Com$, $n \in \mathbb{Z}$, $b \in \mathbb{B}$ and $s, s' \in States$ we have

$$\begin{array}{lllll} \mathcal{I}(ie)(s) = \underline{n} & \iff_* & (ie\ ,\ s) \Downarrow_{IExp} \underline{n} & \iff & [\![ie]\!](s) = n \\ \mathcal{B}(be)(s) = \underline{b} & \iff_* & (be\ ,\ s) \Downarrow_{BExp} \underline{b} & \iff & [\![be]\!](s) = b \\ \mathcal{C}(co)(s) = s' & \iff_* & (co\ ,\ s) \Downarrow_{Com} s' & \iff & [\![co]\!](s) = [s'] \end{array}$$

Proof The bi-implications $\iff_*$ were established in Chapter 2, and are included for completeness in the statement of the theorem.

($\Longrightarrow$) We prove the left to right implications by using Rule Induction for the simultaneously inductively defined sets $\Downarrow_{IExp}$, $\Downarrow_{BExp}$ and $\Downarrow_{Com}$. For example,

$$Prop_{\Downarrow_{IExp}}((ie, s, n)) \stackrel{\text{def}}{=} [\![ie]\!](s) = n;$$

what are the other properties?

We shall just check property closure for the rules $\Downarrow$ OP_2 and $\Downarrow$ LOOP_1.

(*Case* $\Downarrow$ OP_2): The inductive hypotheses are

$$[\![ie_1]\!](s) = n_1 \quad \text{and} \quad [\![ie_2]\!](s) = n_2$$

and we have to prove that $[\![ie_1\ op\ ie_2]\!](s) = n_1\ op\ n_2$. But this is trivial, as

$$[\![ie_1\ op\ ie_2]\!](s) \stackrel{\text{def}}{=} [\![ie_1]\!](s)\ op\ [\![ie_2]\!](s) = n_1\ op\ n_2$$

with the latter equality following by the Inductive Hypotheses.

(*Case* $\Downarrow$ LOOP_1): Let co' be while be do co and suppose that the following Inductive Hypotheses hold for some states s_1, s_2 and s_3:

$$[\![be]\!](s_1) = T \quad \text{and} \quad [\![co]\!](s_1) = [s_2] \quad \text{and} \quad [\![co']\!](s_2) = [s_3] \qquad (H)$$

We have to prove that

$$[\![co']\!](s_1) = [s_3] \qquad (C)$$

Now $[\![co']\!] = \mathit{fix}(\Phi)$ where

$$\Phi = \lambda_{g \in [States, States_\bot]_{tot}}.\lambda_{s_1 \in States}.cond\,([\![be]\!](s_1), g_\bot([\![co]\!](s_1)), [s_1])$$

Hence we have

$$[\![co']\!] = \mathit{fix}(\Phi) = \Phi(\mathit{fix}(\Phi)) = \Phi([\![co']\!]) = \lambda_{s_1 \in States}.cond\,([\![be]\!](s_1), [\![co']\!]_\bot([\![co]\!](s_1)), [s_1])$$

Thus we have

$$
\begin{aligned}
[\![co']\!](s_1) &= cond\,([\![be]\!](s_1), [\![co']\!]_\perp([\![co]\!](s_1)), [s_1]) \\
&= cond\,(T, [\![co']\!]_\perp([s_2]), [s_1]) && \text{by } (H) \\
&= [\![co']\!]_\perp([s_2]) \\
&= [\![co']\!](s_2) \\
&= [s_3] && \text{by } (H)
\end{aligned}
$$

which is (C). We leave the reader to check property closure for the remaining rules.

$(\Longleftarrow)$ We use Rule Induction for the sets *IExp*, *BExp* and *Com*. So for example we take

$$Prop_{IExp}(ie) \stackrel{\text{def}}{=} \text{ for all } n \in \mathbb{N}, \text{ for all } s, \quad [\![ie]\!](s) = n \Longrightarrow (ie\ ,\ s) \Downarrow_{IExp} \underline{n};$$

what are the other properties?

We only consider property closure for the rule eLOOP. We write co' for while be do co. The Inductive Hypotheses are

$$\text{for all } s, s' \in States, \quad [\![co]\!](s) = [s'] \Longrightarrow (co\ ,\ s) \Downarrow_{Com} s' \qquad (H_1)$$

and

$$\text{for all } b \in \mathbb{B}, \text{ for all } s \in States, \quad [\![be]\!](s) = b \Longrightarrow (be\ ,\ s) \Downarrow_{BExp} \underline{b} \qquad (H_2)$$

and then we have to prove that

$$\text{for all } s, s' \in States, \quad [\![co']\!](s) = [s'] \Longrightarrow (co'\ ,\ s) \Downarrow_{Com} s' \qquad (C)$$

Now, (C) is equivalent to

$$[\![co']\!] \preceq I(\mathcal{C}(co')) \quad \in \quad [States, States_\perp]_{tot} \qquad (1)$$

where we recall from page 20 that

$$\mathcal{C}(co')(s) \stackrel{\text{def}}{=} \begin{cases} \text{unique } s' \text{ such that } (co'\ ,\ s) \Downarrow_{Com} s' \text{ if } s' \text{ exists} \\ \text{undefined otherwise} \end{cases}$$

and I is the bijection of page 31.

But we have $[\![co']\!] = \mathit{fix}(\Phi)$ and so using Proposition 3.2.12 part (ii), (1) will hold providing that

$$\Phi(I(\mathcal{C}(co'))) \preceq I(\mathcal{C}(co')) \qquad (2)$$

which is equivalent to saying that for all states s,

$$\Phi(I(\mathcal{C}(co')))(s) \neq \perp \quad \Longrightarrow \quad \Phi(I(\mathcal{C}(co')))(s) = I(\mathcal{C}(co'))(s) \qquad (C')$$

Now for any arbitrary s,

$$\Phi(I(\mathcal{C}(co')))(s) = cond\,([\![be]\!](s), (I(\mathcal{C}(co')))_\perp([\![co]\!](s)), [s])$$

and so if $\Phi(I(\mathcal{C}(co')))(s) \neq \bot$ then either

$$[\![be]\!](s) = T \quad \text{and} \quad \Phi(I(\mathcal{C}(co')))(s) = (I(\mathcal{C}(co')))_\bot([\![co]\!](s)) \neq \bot \tag{3}$$

or

$$[\![be]\!](s) = F \quad \text{and} \quad \Phi(I(\mathcal{C}(co')))(s) = [s]. \tag{4}$$

In the first case (3) we have

$$\bot \neq \Phi(I(\mathcal{C}(co')))(s) = (I(\mathcal{C}(co')))_\bot([\![co]\!](s)) = \begin{cases} I(\mathcal{C}(co'))(s') \\ \text{if } [\![co]\!](s) = [s'] \text{ for some } s' \\ \\ \bot \text{ otherwise} \end{cases}$$

and thus $[\![co]\!](s) = [s']$ and $I(\mathcal{C}(co'))(s') = [s'']$ for some states s' and s''. Then by (H_1) and the definition of $\mathcal{C}$ we have $(co\,,\,s) \Downarrow s'$ and $(co'\,,\,s') \Downarrow s''$. As $[\![be]\!](s) = T$ by assumption, (H_2) tells us that $(be\,,\,s) \Downarrow \underline{T}$. Applying ($\Downarrow$ LOOP$_1$) we conclude that $(co'\,,\,s) \Downarrow s''$. So $\mathcal{C}(co')(s) = s''$ and thus $I(\mathcal{C}(co'))(s) = [s'']$ using Proposition 3.2.4. Hence

$$\Phi(I(\mathcal{C}(co')))(s) = [s''] = I(\mathcal{C}(co'))(s).$$

In the second case (4) we have $\Phi(I(\mathcal{C}(co')))(s) = [s]$ and using (H_2) we deduce that $(be\,,\,s) \Downarrow_{BExp} \underline{F}$. Applying ($\Downarrow$ LOOP$_2$) we have $(co'\,,\,s) \Downarrow_{Com} s$ and so $I(\mathcal{C}(co'))(s) = [s]$. Hence

$$\Phi(I(\mathcal{C}(co')))(s) = [s] = I(\mathcal{C}(co'))(s).$$

From cases (3) and (4) the desired conclusion (C') follows. □

Corollary 3.3.4 For commands co_1 and co_2 we have $co_1 \sim co_2 \iff [\![co_1]\!] = [\![co_2]\!]$.

Proof This follows immediately from Theorem 3.3.3. Why? □

Motivation 3.3.5 We can now complete the proof of Theorem 2.6.3:

Proof

($\Longrightarrow$): We can prove by induction on $\mathbb{C}$ that

$$\text{for all } co_1, \text{ for all } co_2, \text{ for all } \mathbb{C}, \quad ([\![co_1]\!] = [\![co_2]\!] \Longrightarrow [\![\mathbb{C}[co_1]]\!] = [\![\mathbb{C}[co_2]]\!]). \tag{$*$}$$

Then if $co_1 \sim co_2$, the corollary gives $[\![co_1]\!] = [\![co_2]\!]$, $(*)$ gives $[\![\mathbb{C}[co_1]]\!] = [\![\mathbb{C}[co_2]]\!]$, and thus the corollary gives $\mathbb{C}[co_1] \sim \mathbb{C}[co_2]$. This is true for any arbitrary $\mathbb{C}$. □

4
The CSS Machine

4.1 Architecture of the CSS Machine

Motivation 4.1.1 We have seen that a denotational semantics gives a direct method for calculating the functions $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{C}$. This involves a mathematical model of $\mathbb{IMP}$, and while this situtation is fine for a theoretical examination of $\mathbb{IMP}$, we would like to have a more direct, computational method for calculating the above functions. We provide just that in this chapter, by defining an abstract machine which executes via single step re-write rules.

You may like to compare these ideas with those in MC208 where we introduced the SECD machine, which was an operational architecture for computing a value V from any program P (provided that such a V satisfying $P \Downarrow^e V$ exists). In this chapter we shall describe an architecture for directly computing/executing $\mathbb{IMP}$ expressions. An analogy with MC208 is

functional language	compares to	$\mathbb{IMP}$
functional programs	compare to	$\mathbb{IMP}$ *ie* and *be* expressions
functional values	compare to	constants $\underline{c}$
calculating values from programs	compares to	calculating $\mathcal{I}$ and $\mathcal{B}$
SECD machine	compares to	CSS machine
SECD re-writes	compare to	CSS re-writes

You should note that this comparison is very rough and ready, and should be treated with caution. But it ought to give you a clearer idea of the aims of the current chapter.

Definitions 4.1.2 In order to define the CSS machine, we first need a few preliminary definitions. The CSS machine consists of rules for transforming *CSS configurations*. Each configuration is composed of *code* which is executed, a *stack* which consists of a list of integers or Booleans, and a *state* which is the same as for $\mathbb{IMP}$.

A CSS **code** C is a list which is produced by the following grammars:

$$ins ::= e \mid \ op \ \mid \mathsf{STO}(x) \mid \mathsf{BR}(co_1, co_2) \qquad\qquad C ::= \mathsf{nil} \mid ins : C$$

where e is any $\mathbb{IMP}$ expression, op is any operator, x is any variable and co_1 and co_2 are any two commands. The objects *ins* are CSS **instructions**. A **stack** σ is produced by the grammar

$$\sigma ::= \mathsf{nil} \mid \underline{c} : \sigma$$

where c is any integer or Boolean. A **state** s is indeed an $\mathbb{IMP}$ state. We shall write $-$ instead of nil for the empty code or stack list.

A CSS **configuration** is a triple (C, σ, s) whose components are defined as above. A CSS **transition** takes the form

$$(\, C_1 \,,\, \sigma_1 \,,\, s_1 \,) \longmapsto (\, C_2 \,,\, \sigma_2 \,,\, s_2 \,)$$

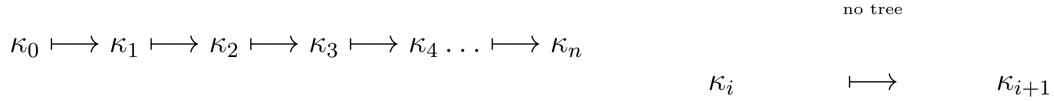
and indicates a relationship between CSS configurations. Thus $\longmapsto$ is a binary relation on the set of all CSS configurations. This binary relation is defined inductively by a set of rules, each rule having the form

$$\frac{}{(\, C_1 \,,\, \sigma_1 \,,\, s_1 \,) \longmapsto (\, C_2 \,,\, \sigma_2 \,,\, s_2 \,)}\; R$$

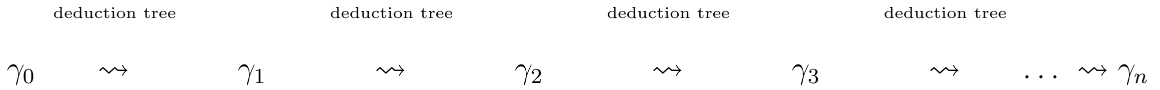
that is, every rule has no hypotheses. We call such a binary relation as $\longmapsto$ which is inductively defined by rules with no hypotheses a **re-write** relation. The CSS re-writes are defined in Table 4.1, where each rule R is written

$$\boxed{C_1}\,\|\,\boxed{\sigma_1}\,\|\,\boxed{s_1} \longmapsto \boxed{C_2}\,\|\,\boxed{\sigma_2}\,\|\,\boxed{s_2}$$

Remark 4.1.3 You may like to compare such re-write rules with the transition steps of $\mathbb{IMP}$ which we met in Chapter 2. They are similar, except that any individual re-write does not require justifying via a deduction tree, because by definition a re-write step is defined by a rule with empty hypotheses.



Rewrite Rules (Abstract Machine)



Transition Semantics

In this *informal* picture, κ denotes a typical CSS configuration, and γ a typical $\mathbb{IMP}$ configuration.

4.2 Correctness of the CSS Machine

Motivation 4.2.1 We prove that the CSS machine is correct for our operational semantics. This means that whenever we execute an expression according to the semantics in Chapter 2, the result matches that of the CSS machine, and vice versa. We make this precise in the following theorem:

Control	Stack	Store		Control	Stack	Store
$\underline{n} : C$	σ	s	$\longmapsto$	C	$\underline{n} : \sigma$	s
$ie_1\ op\ ie_2 : C$	σ	s	$\longmapsto$	$ie_2 : ie_1 :\ op\ : C$	σ	s
$x : C$	σ	s	$\longmapsto$	C	$\underline{s(x)} : \sigma$	s
$op\ : C$	$\underline{n_1} : \underline{n_2} : \sigma$	s	$\longmapsto$	C	$\underline{n_1\ op\ n_2} : \sigma$	s
$\underline{T} : C$	σ	s	$\longmapsto$	C	$\underline{T} : \sigma$	s
$\underline{F} : C$	σ	s	$\longmapsto$	C	$\underline{F} : \sigma$	s
$\mathsf{skip} : C$	σ	s	$\longmapsto$	C	σ	s
$x := ie : C$	σ	s	$\longmapsto$	$ie : \mathsf{STO}(x) : C$	σ	s
$\mathsf{STO}(x) : C$	$\underline{n} : \sigma$	s	$\longmapsto$	C	σ	$s\{x \mapsto n\}$
$(co_1\ ;\ co_2) : C$	σ	s	$\longmapsto$	$co_1 : co_2 : C$	σ	s
$\mathsf{if}\ be\ \mathsf{then}\ co_1\ \mathsf{else}\ co_2 : C$	σ	s	$\longmapsto$	$be : \mathsf{BR}(co_1, co_2) : C$	σ	s
$\mathsf{BR}(co_1, co_2) : C$	$\underline{T} : \sigma$	s	$\longmapsto$	$co_1 : C$	σ	s
$\mathsf{BR}(co_1, co_2) : C$	$\underline{F} : \sigma$	s	$\longmapsto$	$co_2 : C$	σ	s
$\mathsf{while}\ be\ \mathsf{do}\ co : C$	σ	s	$\longmapsto$	$be : \mathsf{BR}((co\ ;\ \mathsf{while}\ be\ \mathsf{do}\ co), \mathsf{skip}) : C$	σ	s

Table 4.1: The CSS Re-Writes

Theorem 4.2.2 For all $n \in \mathbb{Z}$, $b \in \mathbb{B}$, $ie \in \mathit{IExp}$, $be \in \mathit{BExp}$, $co \in \mathit{Com}$ and $s, s_1, s_2 \in$ *States* we have

$$(ie\ ,\ s) \Downarrow_{IExp} \underline{n} \quad\Longleftrightarrow\quad \boxed{ie \parallel - \parallel s} \longmapsto^t \boxed{- \parallel \underline{n} \parallel s}$$

$$(be\ ,\ s) \Downarrow_{BExp} \underline{b} \quad\Longleftrightarrow\quad \boxed{be \parallel - \parallel s} \longmapsto^t \boxed{- \parallel \underline{b} \parallel s}$$

$$(co\ ,\ s_1) \Downarrow_{Com} s_2 \quad\Longleftrightarrow\quad \boxed{co \parallel - \parallel s_1} \longmapsto^t \boxed{- \parallel - \parallel s_2}$$

where $\longmapsto^t$ denotes the transitive closure of $\longmapsto$. (Note that in the CSS configurations, we have written ie instead of $ie : \mathsf{nil}$, $\underline{n}$ instead of $\underline{n} : \mathsf{nil}$, and so on. This slight abuse of notation should keep things tidy, and not cause any problems).

Motivation 4.2.3 The proof method for Theorem 4.2.2 is as follows: For the $\Longrightarrow$ implication(s) we use Rule Induction for the sets $\Downarrow_{IExp}$, $\Downarrow_{BExp}$ and $\Downarrow_{Com}$. For the $\Longleftarrow$ implication(s) we use Mathematical Induction on k, where of course $\kappa \longmapsto^t \kappa'$ iff for some $k \in \mathbb{N}$,

$$\kappa = \kappa_1 \longmapsto \kappa_2 \longmapsto \ldots \longmapsto \kappa_k = \kappa'.$$

If it is not immediately clear to you how Mathematical Induction will be used, then look ahead to page 45. We shall need a few preliminary results before we can prove the theorem.

Lemma 4.2.4 The CSS machine re-writes are deterministic, that is each CSS configuration re-writes to a unique CSS configuration:

More precisely, if

$$\boxed{C \parallel \sigma \parallel s} \longmapsto \boxed{C_1 \parallel \sigma_1 \parallel s_1} \quad\text{and}\quad \boxed{C \parallel \sigma \parallel s} \longmapsto \boxed{C_2 \parallel \sigma_2 \parallel s_2}$$

then $C_1 = C_2$, $\sigma_1 = \sigma_2$ and $s_1 = s_2$.

Proof This follows from simply inspecting the definition of $\longmapsto$: given any $\boxed{C \parallel \sigma \parallel s}$, either there is no transition (the configuration is stuck), or there is only one transition which is valid. □

Lemma 4.2.5 Given any sequence of CSS re-writes, we can (uniformly) extend both the code and stack of each configuration, without affecting the execution of the original code and stack:

For any codes C_i, stacks σ_i, states s_i and $k \in \mathbb{N}$,

$$\boxed{C_1 \parallel \sigma_1 \parallel s_1} \longmapsto^k \boxed{C_2 \parallel \sigma_2 \parallel s_2}$$

$\Longrightarrow$

$$\boxed{C_1 : C_3 \parallel \sigma_1 : \sigma_3 \parallel s_1} \longmapsto^k \boxed{C_2 : C_3 \parallel \sigma_2 : \sigma_3 \parallel s_2}$$

where we define $\longmapsto^0$ to be the identity binary relation on the set of all CSS configurations, and of course $\longmapsto^1$ means just $\longmapsto$; and we write $C : C'$ to mean that the list C is appended to the list C'.

Proof We use induction on $k \in \mathbb{N}$, that is we prove $Prop(k)$ holds for all $k \in \mathbb{N}$ where $Prop(k)$ is the assertion that

for all appropriate codes, stacks and states

$$\boxed{C_1}\boxed{\sigma_1}\boxed{s_1} \longmapsto^{k} \boxed{C_2}\boxed{\sigma_2}\boxed{s_2}$$

$\Longrightarrow$

$$\boxed{C_1 : C_3}\boxed{\sigma_1 : \sigma_3}\boxed{s_1} \longmapsto^{k} \boxed{C_2 : C_3}\boxed{\sigma_2 : \sigma_3}\boxed{s_2}.$$

(*Proof of* $Prop(0)$): This is trivially true (why?).

(*Proof of* for all $k_0 \in \mathbb{N}$, $Prop(k)_{k \leq k_0} \Longrightarrow Prop(k_0 + 1)$): Let k_0 be arbitrary and assume (inductively) that $Prop(k)$ holds for all $k \leq k_0$. We prove $Prop(k_0 + 1)$ from these assumptions. Spelling this out, we shall show that if

for all codes, stacks and states,

$$\boxed{C_1}\boxed{\sigma_1}\boxed{s_1} \longmapsto^{k} \boxed{C_2}\boxed{\sigma_2}\boxed{s_2}$$

$\Longrightarrow$

$$\boxed{C_1 : C_3}\boxed{\sigma_1 : \sigma_3}\boxed{s_1} \longmapsto^{k} \boxed{C_2 : C_3}\boxed{\sigma_2 : \sigma_3}\boxed{s_2}$$

holds for each $k \leq k_0$, then

for all codes, stacks and states,

$$\boxed{C_1}\boxed{\sigma_1}\boxed{s_1} \longmapsto^{k_0+1} \boxed{C_2}\boxed{\sigma_2}\boxed{s_2}$$

$\Longrightarrow$

$$\boxed{C_1 : C_3}\boxed{\sigma_1 : \sigma_3}\boxed{s_1} \longmapsto^{k_0+1} \boxed{C_2 : C_3}\boxed{\sigma_2 : \sigma_3}\boxed{s_2}.$$

Let us choose arbitrary codes, stacks and states for which

$$\boxed{C_1}\boxed{\sigma_1}\boxed{s_1} \longmapsto^{k_0+1} \boxed{C_2}\boxed{\sigma_2}\boxed{s_2}$$

We now consider the possible forms that C_1 can take; here we just give a couple of cases:

(*Case* C_1 *is* $-$): We have to prove that

$$\boxed{-}\boxed{\sigma_1}\boxed{s_1} \longmapsto^{k_0+1} \boxed{C_2}\boxed{\sigma_2}\boxed{s_2}$$

$\Longrightarrow$

$$\boxed{- : C_3}\boxed{\sigma_1 : \sigma_3}\boxed{s_1} \longmapsto^{k_0+1} \boxed{C_2 : C_3}\boxed{\sigma_2 : \sigma_3}\boxed{s_2}$$

But there are no transitions from a configuration with empty code. Thus the above implication asserts that "*false* $\Longrightarrow$??" which is true. (Ask if you are confused by this).

(*Case* C_1 *is* $\underline{n} : C_1$): Suppose that we have

$$\boxed{\underline{n} : C_1}\boxed{\sigma_1}\boxed{s_1} \longmapsto^{k_0+1} \boxed{C_2}\boxed{\sigma_2}\boxed{s_2}$$

We need to prove that

$$\boxed{\underline{n} : C_1 : C_3 \,\|\, \sigma_1 : \sigma_3 \,\|\, s_1} \longmapsto^{k_0+1} \boxed{C_2 : C_3 \,\|\, \sigma_2 : \sigma_3 \,\|\, s_2} \tag{1}$$

By Lemma 4.2.4 we must have[1]

$$\boxed{\underline{n} : C_1 \,\|\, \sigma_1 \,\|\, s_1} \longmapsto^{1} \boxed{C_1 \,\|\, \underline{n} : \sigma_1 \,\|\, s_1} \longmapsto^{k_0} \boxed{C_2 \,\|\, \sigma_2 \,\|\, s_2}$$

and so by induction ($k_0 \leq k_0$!!)

$$\boxed{C_1 : C_3 \,\|\, \underline{n} : \sigma_1 : \sigma_3 \,\|\, s_1} \longmapsto^{k_0} \boxed{C_2 : C_3 \,\|\, \sigma_2 : \sigma_3 \,\|\, s_2} \tag{2}$$

But

$$\boxed{\underline{n} : C_1 : C_3 \,\|\, \sigma_1 : \sigma_3 \,\|\, s_1} \longmapsto^{1} \boxed{C_1 : C_3 \,\|\, \underline{n} : \sigma_1 : \sigma_3 \,\|\, s_1} \tag{3}$$

and then (2) and (3) prove (1) as required.

(*Case* C_1 *is* $\mathsf{BR}(co_1, co_2) : C_1$): Assume that[2]

$$\boxed{\mathsf{BR}(co_1, co_2) : C_1 \,\|\, \underline{T} : \sigma_1 \,\|\, s_1} \longmapsto^{k_0+1} \boxed{C_2 \,\|\, \sigma_2 \,\|\, s_2}$$

We need to prove that

$$\boxed{\mathsf{BR}(co_1, co_2) : C_1 : C_3 \,\|\, \underline{T} : \sigma_1 : \sigma_3 \,\|\, s_1} \longmapsto^{k_0+1} \boxed{C_2 : C_3 \,\|\, \sigma_2 : \sigma_3 \,\|\, s_2} \tag{4}$$

Now

$$\boxed{\mathsf{BR}(co_1, co_2) : C_1 \,\|\, \underline{T} : \sigma_1 \,\|\, s_1} \longmapsto^{1} \boxed{co_1 : C_1 \,\|\, \sigma_1 \,\|\, s_1}$$

and so by induction we have

$$\boxed{co_1 : C_1 : C_3 \,\|\, \sigma_1 : \sigma_3 \,\|\, s_1} \longmapsto^{k_0} \boxed{C_2 : C_3 \,\|\, \sigma_2 : \sigma_3 \,\|\, s_2} \tag{5}$$

But

$$\boxed{\mathsf{BR}(co_1, co_2) : C_1 : C_3 \,\|\, \underline{T} : \sigma_1 : \sigma_3 \,\|\, s_1} \longmapsto^{1} \boxed{co_1 : C_1 : C_3 \,\|\, \sigma_1 : \sigma_3 \,\|\, s_1} \tag{6}$$

and then (5) and (6) imply (4) as required. We omit the remaining cases. □

Lemma 4.2.6 Given a sequence of re-writes in which the code of the first configuration takes the form of two appended codes, then each of these codes may be executed separately:

For all $k \in \mathbb{N}$, and

[1] All we are saying here is that any sequence of re-write steps must have a unique form. We often use determinism of $\longmapsto$ in the next few pages, without always quoting Lemma 4.2.4.

[2] Given that the code begins with the instruction $\mathsf{BR}(co_1, co_2)$ and we know that there is a valid re-write, the stack *must* begin with $\underline{T}$.

for all appropriate codes, stacks and states, if

$$\boxed{C_1 : C_2 \,\|\, \sigma \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, \sigma'' \,\|\, s''}$$

then there is a stack and state σ' and s', and $k_1, k_2 \in \mathbb{N}$ for which

$$\boxed{C_1 \,\|\, \sigma \,\|\, s} \longmapsto^{k_1} \boxed{- \,\|\, \sigma' \,\|\, s'}$$
$$\boxed{C_2 \,\|\, \sigma' \,\|\, s'} \longmapsto^{k_2} \boxed{- \,\|\, \sigma'' \,\|\, s''}$$

where $k_1 + k_2 = k$.

Proof We use Mathematical Induction on k; let $Prop(k)$ denote the property of k given in the above box.

(*Proof of* $Prop(0)$): This is trivially true (why?).

(*Proof of* for all $k_0 \in \mathbb{N}$, $Prop(k)_{k \leq k_0} \Longrightarrow Prop(k_0 + 1)$): Let k_0 be arbitrary and assume (inductively) that $Prop(k)$ holds for all $k \leq k_0$. We prove $Prop(k_0 + 1)$ from these assumptions. Let us choose arbitrary codes, stacks and states for which

$$\boxed{C_1 : C_2 \,\|\, \sigma \,\|\, s} \longmapsto^{k_0+1} \boxed{- \,\|\, \sigma'' \,\|\, s''}$$

and then consider the possible forms that C_1 can take.

(*Case* C_1 *is* while be do $co : C_1$):

We suppose that

$$\boxed{\mathsf{while}\ be\ \mathsf{do}\ co : C_1 : C_2 \,\|\, \sigma \,\|\, s} \longmapsto^{k_0+1} \boxed{- \,\|\, \sigma'' \,\|\, s''}$$

and hence by Lemma 4.2.4

$$\begin{aligned}
&\boxed{\mathsf{while}\ be\ \mathsf{do}\ co : C_1 : C_2 \,\|\, \sigma \,\|\, s} \\
&\longmapsto^{1} \boxed{be : \mathsf{BR}((co\ ;\ \mathsf{while}\ be\ \mathsf{do}\ co), \mathsf{skip}) : C_1 : C_2 \,\|\, \sigma \,\|\, s} \\
&\longmapsto^{k_0} \boxed{- \,\|\, \sigma'' \,\|\, s''}
\end{aligned}$$

So as $k_0 \leq k_0$ (!), by induction we have k_1, k_2 where $k_0 = k_1 + k_2$ and σ' and s' such that

$$\boxed{be : \mathsf{BR}((co\ ;\ \mathsf{while}\ be\ \mathsf{do}\ co), \mathsf{skip}) : C_1 \,\|\, \sigma \,\|\, s} \longmapsto^{k_1} \boxed{- \,\|\, \sigma' \,\|\, s'} \qquad (1)$$

and

$$\boxed{C_2 \,\|\, \sigma' \,\|\, s'} \longmapsto^{k_2} \boxed{- \,\|\, \sigma'' \,\|\, s''} \qquad (2)$$

But

$$\boxed{\mathsf{while}\ be\ \mathsf{do}\ co : C_1 \,\|\, \sigma \,\|\, s} \longmapsto^{1} \boxed{be : \mathsf{BR}((co\ ;\ \mathsf{while}\ be\ \mathsf{do}\ co), \mathsf{skip}) : C_1 \,\|\, \sigma \,\|\, s} \qquad (3)$$

and so we are done using (1) with (3), and (2). The other cases are left as exercises. □

Lemma 4.2.7 For all appropriate codes, stacks, states and natural numbers,

$$\boxed{ie \parallel \sigma \parallel s} \longmapsto^k \boxed{- \parallel \sigma' \parallel s'} \quad \Longrightarrow$$

$$s = s' \quad \text{and} \quad \sigma' = \underline{n} : \sigma \text{ some } n \in \mathbb{Z} \quad \text{and} \quad \boxed{ie \parallel - \parallel s} \longmapsto^k \boxed{- \parallel \underline{n} \parallel s}$$

and

$$\boxed{be \parallel \sigma \parallel s} \longmapsto^k \boxed{- \parallel \sigma' \parallel s'} \quad \Longrightarrow$$

$$s = s' \quad \text{and} \quad \sigma' = \underline{b} : \sigma \text{ some } b \in \mathbb{B} \quad \text{and} \quad \boxed{be \parallel - \parallel s} \longmapsto^k \boxed{- \parallel \underline{b} \parallel s}$$

Proof A lengthy, trivial Rule Induction on *IExp* and *BExp*. □

Proving Theorem 4.2.2

Let us now give the proof of the correctness theorem:

Proof ($\Longrightarrow$): We use Rule Induction for $\Downarrow_{IExp}$, $\Downarrow_{BExp}$ and $\Downarrow_{Com}$. We show property closure for just one example rule:

($Case \Downarrow \textsc{op}_1$): The inductive hypotheses are

$$\boxed{ie_1 \parallel - \parallel s} \longmapsto^t \boxed{- \parallel \underline{n_1} \parallel s} \quad \text{and} \quad \boxed{ie_2 \parallel - \parallel s} \longmapsto^t \boxed{- \parallel \underline{n_2} \parallel s}$$

Then we have

$$\begin{array}{rcl} \boxed{ie_1 \; op \; ie_2 \parallel - \parallel s} & \longmapsto & \boxed{ie_2 : ie_1 : \; op \parallel - \parallel s} \\ \text{by Lemma 4.2.5 and inductive hypotheses} & \longmapsto^t & \boxed{ie_1 : \; op \parallel \underline{n_2} \parallel s} \\ \text{by Lemma 4.2.5 and inductive hypotheses} & \longmapsto^t & \boxed{op \parallel \underline{n_1} : \underline{n_2} \parallel s} \\ & \longmapsto & \boxed{- \parallel \underline{n_1 \; op \; n_2} \parallel s} \end{array}$$

as required. We leave the reader to verify property closure of the remaining rules.

($\Longleftarrow$): We prove each of the three right to left implications separately, by Mathematical Induction. Note that the first is:

$$\text{for all } ie, n, s, \quad \boxed{ie \parallel - \parallel s} \longmapsto^t \boxed{- \parallel \underline{n} \parallel s} \quad \Longrightarrow \quad (ie \; , \; s) \Downarrow_{IExp} \underline{n}.$$

But this statement is logically equivalent to

$$\text{for all } k, \quad \boxed{\text{for all } ie, n, s, \quad \boxed{ie \parallel - \parallel s} \longmapsto^k \boxed{- \parallel \underline{n} \parallel s} \quad \Longrightarrow \quad (ie \; , \; s) \Downarrow_{IExp} \underline{n}}$$

which you should check!! We prove the latter assertion by induction on $k \in \mathbb{N}$, letting $Prop(k)$ denote the boxed proposition:

($Proof \; of \; Prop(0)$): This is trivially true (why?).

(*Proof of* for all $k_0 \in \mathbb{N}$, $Prop(k)_{k \leq k_0} \Longrightarrow Prop(k_0+1)$): Suppose that for some arbitrary k_0, ie, n and s

$$\boxed{ie}\boxed{-}\boxed{s} \longmapsto^{k_0+1} \boxed{-}\boxed{\underline{n}}\boxed{s} \qquad (*)$$

and then we prove $(ie\,,\ s) \Downarrow_{IExp} \underline{n}$ by considering cases on ie.

(*Case* ie is $\underline{m}$): If $m \neq n$ then $(*)$ is false, so the implication is true. If $m = n$, note that as $(\underline{n}\,,\ s) \Downarrow_{IExp} \underline{n}$ there is nothing to prove.

(*Case* ie is $ie_1\ op\ ie_2$): Suppose that

$$\boxed{ie_1\ op\ ie_2}\boxed{-}\boxed{s} \longmapsto^{k_0+1} \boxed{-}\boxed{\underline{n}}\boxed{s}$$

and so

$$\boxed{ie_2 : ie_1 :\ op}\boxed{-}\boxed{s} \longmapsto^{k_0} \boxed{-}\boxed{\underline{n}}\boxed{s}.$$

Using Lemmas 4.2.6 and 4.2.7 we have that

$$\begin{array}{rcl} \boxed{ie_2}\boxed{-}\boxed{s} & \longmapsto^{k_1} & \boxed{-}\boxed{\underline{n_2}}\boxed{s} \\ \boxed{ie_1 :\ op}\boxed{\underline{n_2}}\boxed{s} & \longmapsto^{k_2} & \boxed{-}\boxed{\underline{n}}\boxed{s} \end{array}$$

where $k_1 + k_2 = k_0$, and repeating for the latter transition we get

$$\begin{array}{rcl} \boxed{ie_1}\boxed{\underline{n_2}}\boxed{s} & \longmapsto^{k_{21}} & \boxed{-}\boxed{\underline{n_1} : \underline{n_2}}\boxed{s} \\ \boxed{op}\boxed{\underline{n_1} : \underline{n_2}}\boxed{s} & \longmapsto^{k_{22}} & \boxed{-}\boxed{\underline{n}}\boxed{s} \end{array} \qquad (1)$$

where $k_{21} + k_{22} = k_2$. So as $k_1 \leq k_0$, by Induction we deduce that $(ie_2\,,\ s) \Downarrow_{IExp} \underline{n_2}$, and from Lemma 4.2.7 that

$$\boxed{ie_1}\boxed{-}\boxed{s} \longmapsto^{k_{21}} \boxed{-}\boxed{\underline{n_1}}\boxed{s}.$$

Also, as $k_{21} \leq k_0$, we have Inductively that $(ie_1\,,\ s) \Downarrow_{IExp} \underline{n_1}$ and hence

$$(ie_1\ op\ ie_2\,,\ s) \Downarrow_{IExp} \underline{n_1\ op\ n_2}.$$

But from Lemma 4.2.4 and (1) we see that $\underline{n_1\ op\ n_2} = \underline{n}$ and we are done.

We omit the remaining cases.

Note that the second right to left implication (dealing with Boolean expressions) involves just the same proof technique.

The third right to left implication is (equivalent to):

for all k,

$$\boxed{\text{for all } co, s, s' \quad \boxed{co}\boxed{-}\boxed{s} \longmapsto^{k} \boxed{-}\boxed{\sigma}\boxed{s'} \quad \Longrightarrow \quad \sigma = - \quad \text{and} \quad (co\,,\ s) \Downarrow_{Com} s'}$$

which you should check!! We prove the latter assertion by induction on $k \in \mathbb{N}$, letting $Prop(k)$ denote the boxed proposition:

(*Proof of* $Prop(0)$): This is trivially true (why?).

(*Proof of* for all $k_0 \in \mathbb{N}$, $Prop(k)_{k \leq k_0} \implies Prop(k_0+1)$): Choose arbitrary $k_0 \in \mathbb{N}$. We shall show that if

$$\text{for all } co, s, s', \quad \boxed{co \,\|\, - \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, \sigma \,\|\, s'} \quad \implies \quad \sigma = - \quad \text{and} \quad (co\,,\,s) \Downarrow_{Com} s'$$

for all $k \leq k_0$, then

$$\text{for all } co, s, s', \quad \boxed{co \,\|\, - \,\|\, s} \longmapsto^{k_0+1} \boxed{- \,\|\, \sigma \,\|\, s'} \quad \implies \quad \sigma = - \text{ and } (co\,,\,s) \Downarrow_{Com} s'$$

Pick arbitrary co and σ and s, s' and suppose that

$$\boxed{co \,\|\, - \,\|\, s} \longmapsto^{k_0+1} \boxed{- \,\|\, \sigma \,\|\, s'}$$

We consider cases for co:

(*Case* co is $x := ie$): Using Lemma 4.2.4, we must have

$$\boxed{x := ie \,\|\, - \,\|\, s} \longmapsto^{1} \boxed{ie : \mathsf{STO}(x) \,\|\, - \,\|\, s} \longmapsto^{k_0} \boxed{- \,\|\, \sigma \,\|\, s'}$$

and so by Lemmas 4.2.6 and 4.2.7

$$\begin{array}{rcl} \boxed{ie \,\|\, - \,\|\, s} & \longmapsto^{k_1} & \boxed{- \,\|\, \underline{n} \,\|\, s} \\ \boxed{\mathsf{STO}(x) \,\|\, \underline{n} \,\|\, s} & \longmapsto^{k_2} & \boxed{- \,\|\, \sigma \,\|\, s'} \end{array} \qquad (1)$$

where $k_1 + k_2 = k_0$. By determinism for (1) we have $\sigma = -$ and $s\{x \mapsto n\} = s'$. By the first right to left implication for integer expressions (proved above) we have $(ie\,,\,s) \Downarrow_{IExp} \underline{n}$. Hence $(x := ie\,,\,s) \Downarrow_{Com} s\{x \mapsto n\}$, and as $s\{x \mapsto n\} = s'$ we are done. NB this case did not make use of the inductive hypotheses $Prop(k)_{k \leq k_0}$!

(*Case* co is $co\ ;\ co'$): Do this as an exercise!

The remaining cases are omitted. □

4.3 CSS Executions

Examples 4.3.1

(1) Let s be a state for which $s(x) = 6$. Then we have

$$\begin{array}{rcl} \boxed{\underline{10} - x \,\|\, - \,\|\, s} & \longmapsto & \boxed{x : \underline{10} : - \,\|\, - \,\|\, s} \\ & \longmapsto & \boxed{\underline{10} : - \,\|\, \underline{6} \,\|\, s} \\ & \longmapsto & \boxed{- \,\|\, \underline{10} : \underline{6} \,\|\, s} \\ & \longmapsto & \boxed{- \,\|\, \underline{4} \,\|\, s} \end{array}$$

where we have written $-$ for both empty list and subtraction—care!

(2) Let s be a state for which $s(x) = 1$. Then we have

$$\boxed{\textsf{if } x \geq \underline{0} \textsf{ then } x := x - \underline{1} \textsf{ else skip} \parallel - \parallel s} \longmapsto \boxed{x \geq \underline{0} : \textsf{BR}(x := x - \underline{1}, \textsf{skip}) \parallel - \parallel s}$$

$$\longmapsto \boxed{\underline{0} : x \geq : \textsf{BR}(x := x - \underline{1}, \textsf{skip}) \parallel - \parallel s}$$

$$\longmapsto \boxed{x : \geq : \textsf{BR}(x := x - \underline{1}, \textsf{skip}) \parallel \underline{0} \parallel s}$$

$$\longmapsto \boxed{\geq : \textsf{BR}(x := x - \underline{1}, \textsf{skip}) \parallel \underline{1} : \underline{0} \parallel s}$$

$$\longmapsto \boxed{\textsf{BR}(x := x - \underline{1}, \textsf{skip}) \parallel \underline{T} \parallel s}$$

$$\longmapsto \boxed{x := x - \underline{1} \parallel - \parallel s}$$

$$\longmapsto \boxed{x - \underline{1} : \textsf{STO}(x) \parallel - \parallel s}$$

$$\longmapsto \boxed{\underline{1} : x : - : \textsf{STO}(x) \parallel - \parallel s}$$

$$\longmapsto \boxed{x : - : \textsf{STO}(x) \parallel \underline{1} \parallel s}$$

$$\longmapsto \boxed{- : \textsf{STO}(x) \parallel \underline{1} : \underline{1} \parallel s}$$

$$\longmapsto \boxed{\textsf{STO}(x) \parallel \underline{0} \parallel s}$$

$$\longmapsto \boxed{- \parallel - \parallel s\{x \mapsto 0\}}$$

5
Elementary Domain Theory

5.1 Introduction

Motivation 5.1.1 We have seen how to give a denotational semantics to $\mathbb{IMP}$. This involved us dealing with partial functions between states. In particular, in order to model a while loop, we needed a fixpoint of a certain function (between $[States, States]_{par}$ and itself). To choose that fixpoint, we introduced a partial order on the set of partial functions between states. This is fine for a simple language such as $\mathbb{IMP}$, but for more complex languages we need to be able to choose fixpoints at a more abstract level, and this involves dealing with more complex partial orders (recall the partial order on $[States, States]_{par}$ is really very simple, being given simply by the subset $\subseteq$ relation).

We can achieve this abstraction by using Domain Theory. Basically, a domain is a certain kind of partially ordered set for which we can guarantee that fixpoints of certain kinds of functions always exist. These functions are called *continuous*. We will model recursive programs as least fixpoints of continuous functions. In fact, in order to ensure that whatever we are doing within our model, fixpoints always exist when we need them, all of our programming language constructs (not just recursive programs) will be modelled by continuous functions. Now, of course there are fewer continous functions than total functions, but this does not present a problem as continous functions are sufficient to model the kinds of programming languages we are interested in. Let us now introduce continuous functions:

5.2 Cpos and Continuous Functions

Definitions 5.2.1 Recall that a **partial order** $\preceq$ on a set D is a binary relation on D which is

- reflexive (for all $d \in D$, $d \preceq d$);
- transitive (for all $d, d', d'' \in D$, $d \preceq d'$ and $d' \preceq d''$ implies $d \preceq d''$); and
- anti-symmetric (for $d, d' \in D$, $d \preceq d'$ and $d' \preceq d$ implies $d = d'$).

A **partially ordered set** P is a pair $(D, \preceq)$ where D is any set and $\preceq$ is a partial order on D. We often abbreviate *partially ordered set* to **poset**. We call D the **underlying** set of the poset P. If $d, d' \in D$ are any two elements, we say that d and d' are **comparable** if either $d \preceq d'$ *or* $d' \preceq d$. The two elements are **incomparable** if they are not comparable.

We shall often refer to a poset simply by naming its underlying set (so we might say "consider the poset D") and we shall use the same symbol $\preceq$ to denote the partial orders on a variety of different posets (we might say that the symbol $\preceq$ has been **overloaded**).

Definitions 5.2.2 A **chain** in a poset D is a function $c : \mathbb{N} \to D$ for which

$$c_0 \preceq c_1 \preceq c_2 \preceq \dots$$

where we write c_n instead of the more usual notation $c(n)$ for functions. This helps us to think of the elements c_n (where $n \in \mathbb{N}$) as a "sequence" of elements of D which are indexed by the natural numbers. We call each c_n a **link** in the chain c. We shall often refer to a chain in D by explicitly naming the elements in the image of the chain $c : \mathbb{N} \to D$, that is, just naming the links of the chain. We might say "consider the chain

$$c_0 \preceq c_1 \preceq c_2 \preceq \dots$$

in D". Finally, we may also say "consider the chain $(c_n \mid n \in \mathbb{N})$ in D" where the notation indicates that we have specified a sequence of elements in D (but have not made the ordering explicit).

Note that the image of c is a subset of D, that is

$$\{ c_n \mid n \in \mathbb{N} \} \subseteq D.$$

Thus we can consider the join (least upper bound) of this subset of D, which if it exists will be denoted by $\bigvee_n^\infty c_n$. So, recall that this means

$$\text{for all } d \in D, \quad \bigvee_{n=0}^{\infty} c_n \preceq d \Longleftrightarrow (\text{ for all } n \in \mathbb{N}, \quad c_n \preceq d)$$

We refer to the element $\bigvee_{n=0}^\infty c_n$ of D as the **join of the chain** c. A **cpo** is a poset D which possesses a join for every chain c in D.

Examples 5.2.3

(1) The powerset $\mathcal{P}(A)$ of a set A which is partially ordered by $\subseteq$ is a cpo. Given a chain $X_0 \subseteq X_1 \subseteq X_2 \dots$ in A, its join is given by $\bigcup_{n=0}^\infty X_n$, that is

$$\bigvee_{n=0}^{\infty} X_n = \bigcup_{n=0}^{\infty} X_n.$$

We check that $\bigcup_{n=0}^\infty X_n$ is indeed the least upper bound of the chain $(X_n \mid n \in \mathbb{N})$. It is trivially an upper bound: $X_m \subseteq \bigcup_{n=0}^\infty X_n$ holds by definition of union for each $m \in \mathbb{N}$. So we check that $\bigcup_{n=0}^\infty X_n$ is least. If U is any element of $\mathcal{P}(A)$ for which $X_n \subseteq U$ for each $n \in \mathbb{N}$, then we need to verify that $\bigcup_{n=0}^\infty X_n \subseteq U$. This is trivial:

$$a \in \bigcup_{n=0}^{\infty} X_n \Longrightarrow a \in X_m \text{ for some } m \in \mathbb{N} \Longrightarrow a \in U.$$

(2) The set $[A, B]_{par}$ of partial functions from A to B, partially ordered by $\subseteq$, is a cpo with joins given by union, just as in (i). It is an exercise to verify this.

(3) For any set A the relation of equality is a partial order:

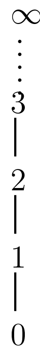
$$\text{for all } a, a' \in A, \quad a \preceq a' \iff a = a'$$

With this order, A is in fact a cpo. We call it the **discrete** cpo on A. So any chain c in A must satisfy $c_0 = c_1 = c_2 = \dots$ and so $\bigvee_{n=0}^{\infty} c_n = c_0$. Check this!

(4) Set $\mathbb{N}^\infty \stackrel{\text{def}}{=} \mathbb{N} \cup \{ \infty \}$. We can define a partial order on $\mathbb{N}^\infty$ by setting

$$\text{for all } x, x' \in \mathbb{N}^\infty, \quad x \preceq x' \iff (x, x' \in \mathbb{N} \text{ and } x \leq x') \text{ or } (x' = \infty).$$

We call $\mathbb{N}^\infty$ the **topped vertical natural numbers**. Then it is an exercise to verify that $\mathbb{N}^\infty$ is a cpo. We can draw an informal Hasse diagram of $\mathbb{N}^\infty$:



Definitions 5.2.4 Recall that a function[1] $f : D \to E$ between cpos is **monotone** if

$$\text{for all } d, d' \in D, \quad d \preceq d' \Longrightarrow f(d) \preceq f(d')$$

(where of course $f(d) \preceq f(d')$ refers to the partial order on E).

We say that $f : D \to E$ is **continuous** if it is monotone, and for all chains c in D we have

$$f(\bigvee_{n=0}^{\infty} c_n) = \bigvee_{n=0}^{\infty} f(c_n) \qquad (*)$$

Remark 5.2.5 We make two comments about the definition of continuity:

(i) Note that the composition $f \circ c : \mathbb{N} \to E$ is indeed a chain[2] since f is monotone, that is $(f(c_n) \mid n \in \mathbb{N})$ is a chain in E. Thus the join on the right hand side of $(*)$ does exist.

(ii) For any $m \in \mathbb{N}$ we have $c_m \preceq \bigvee_{n=0}^{\infty} c_n$. Thus when f is monotone, $f(c_m) \preceq f(\bigvee_{n=0}^{\infty} c_n)$, and so

$$\bigvee_{m}^{\infty} f(c_m) \preceq f(\bigvee_{n=0}^{\infty} c_n).$$

Thus $(*)$ holds if and only if $f(\bigvee_{n=0}^{\infty} c_n) \preceq \bigvee_{n=0}^{\infty} f(c_n)$.

[1] In these notes, the adjectives *monotone* and *continuous* will only be applied to total functions; and whenever we just talk of a "function" it is implicit that it is total, unless we specifically state that it is partial.

[2] thus $(f \circ c)_n \stackrel{\text{def}}{=} f(c_n)$.

Proposition 5.2.6

(i) The function $id_D \stackrel{\text{def}}{=} \lambda_{d \in D}.d : D \to D$ is continuous.

(ii) The **composition** $g \circ f \stackrel{\text{def}}{=} \lambda_{d \in D}.g(f(d)) : D \to F$ of continuous functions $f : D \to E$ and $g : E \to F$ is continuous.

(iii) If D is a discrete cpo, then any function $f : D \to E$ is continuous.

Proof An exercise. □

5.3 Constructions on Cpos

Motivation 5.3.1 You will be familiar with various procedures for building up new sets from certain given sets, for example we can construct the cartesian product $A \times B$ given the sets A and B. We can perform similar operations with cpos replacing sets, which we now describe. All of these constructions will be used in giving a denotational semantics to the languages which appear later on in the course.

Binary Product

Definitions 5.3.2 Given cpos D_1 and D_2, their **binary product** has underlying set which is the usual (cartesian) binary product $D_1 \times D_2$ with a partial order defined by

$$\text{for all } d_1, d_1' \in D_1, d_2, d_2' \in D_2, \quad (d_1, d_2) \preceq (d_1', d_2') \iff d_1 \preceq d_1' \text{ and } d_2 \preceq d_2'.$$

This poset is indeed a cpo: let us see that joins of all chains exist. Suppose that $c : \mathbb{N} \to D_1 \times D_2$ is a chain, where each c_n is an element of $D_1 \times D_2$ which we denote by (c_n^1, c_n^2). It is easy to see that each of $(c_n^1 \mid n \in \mathbb{N})$ and $(c_n^2 \mid n \in \mathbb{N})$ is a chain in D_1 and D_2 respectively. Then it is the case that

$$\bigvee_{n=0}^{\infty} c_n = (\bigvee_{n=0}^{\infty} c_n^1, \bigvee_{n=0}^{\infty} c_n^2)$$

where the joins on the right exist as each of D_1 and D_2 is a cpo. It is an exercise for you to check this.

There are continuous **projection** functions given by

$$fst : D_1 \times D_2 \to D_1 \qquad fst(d_1, d_2) \stackrel{\text{def}}{=} d_1$$

$$snd : D_1 \times D_2 \to D_2 \qquad snd(d_1, d_2) \stackrel{\text{def}}{=} d_2$$

Given continuous functions $f_1 : E \to D_1$ and $f_2 : E \to D_2$ there is a continuous function[3] denoted by $\langle f_1, f_2 \rangle : E \to D_1 \times D_2$ which is defined by $\langle f_1, f_2 \rangle(e) \stackrel{\text{def}}{=} (f_1(e), f_2(e))$.

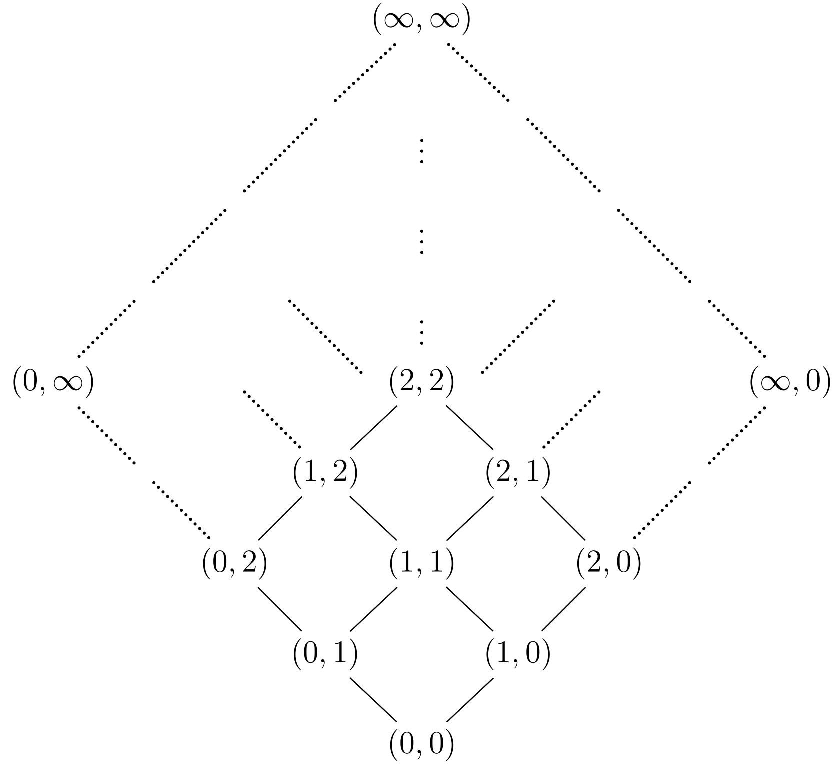
[3] Care! Note that $\langle f_1, f_2 \rangle \in [E, D_1 \times D_2]_{cts}$ and that $(f_1, f_2) \in [E, D_1]_{cts} \times [E, D_2]_{cts}$.

Given continuous functions $f_1 : E_1 \to D_1$ and $f_2 : E_2 \to D_2$ there is a continuous function denoted by $f_1 \times f_2 : E_1 \times E_2 \to D_1 \times D_2$ which is defined by

$$f_1 \times f_2 \stackrel{\text{def}}{=} \langle f_1 \circ \mathit{fst}, f_2 \circ \mathit{snd} \rangle.$$

Thus it follows that $(f_1 \times f_2)(e_1, e_2) = (f_1(e_1), f_2(e_2))$.

Examples 5.3.3 An informal Hasse diagram of $\mathbb{N}^\infty \times \mathbb{N}^\infty$:



n-ary Product

Definitions 5.3.4 Suppose that D_1, D_2, ... D_n are each cpos, where n is a finite non-zero natural number. We can extend the definition of binary product to the case of n cpos. There is a cpo denoted either by $D_1 \times D_2 \times \ldots \times D_n$ or often by $\Pi_{i=1}^{i=n} D_i$ whose underlying set consists of n-tuples of elements

$$\Pi_{i=1}^{i=n} D_i \stackrel{\text{def}}{=} \{ (d_1, \ldots, d_n) \mid d_i \in D_i \}$$

with an order defined by

$$(d_1, \ldots, d_n) \preceq (d'_1, \ldots, d'_n) \iff \text{for all } i \in \{1, \ldots, n\}, \quad d_i \preceq d'_i$$

Thus, informally, two n-tuples are related by $\preceq$ in $\Pi_{i=1}^{i=n} D_i$ if and only if each of their respective components are related in the appropriate cpo D_i. We shall write $\vec{d}$ as an abbreviation for $(d_1, \ldots, d_n)$ when no confusion can result.

There are continuous **projection** functions $proj_i : \Pi_{i=1}^{i=n} D_i \to D_i$ given by $(d_1, \ldots, d_n) \mapsto d_i$. Thus in the case that $n = 2$ we have $fst = proj_1$ and $snd = proj_2$. We sometimes write $proj_i(t)$ as t_i when $t \in \Pi_{i=1}^{i=n} D_i$.

We write D^n to denote the cpo

$$\underbrace{D \times \ldots \times D}_{n\text{-times}}$$

Finally, we let D^0 denote the one-point cpo $\{ * \}$ (where $*$ is some arbitrary element).

Cpos of Continuous Functions

Definitions 5.3.5 Suppose that D and E are cpos. Then the cpo of continuous functions denoted by $[D, E]_{cts}$ has underlying set[4]

$$\{ f \mid f \in [D, E]_{tot} \text{ and } f \text{ is continuous } \}$$

with a partial order given by

$$f \preceq f' \iff \text{for all } d \in D, \quad f(d) \preceq f'(d).$$

Joins of chains in $[D, E]_{cts}$ are given *pointwise*: Suppose that $c : \mathbb{N} \to [D, E]_{cts}$ is a chain, so that each c_n is a continuous function between D and E. Then

$$\bigvee_{n=0}^{\infty} c_n = \lambda_{d \in D}. \bigvee_{n=0}^{\infty} c_n(d),$$

that is, if we write $f \stackrel{\text{def}}{=} \bigvee_{n=0}^{\infty} c_n$, then $f(d) = \bigvee_{n=0}^{\infty} c_n(d)$ for any $d \in D$. You should check the details of these definitions and assertions very carefully!

There is a continuous function $ev : [D, E]_{cts} \times D \to E$ which is defined by $ev(f, d) \stackrel{\text{def}}{=} f(d)$. We call ev the **evaluation** function. Finally, given a continuous function $g : D' \times D \to E$ there is a continuous function $cur(g) : D' \to [D, E]_{cts}$ where for each $d' \in D'$ we have $cur(g)(d') \stackrel{\text{def}}{=} \lambda_{d \in D}.g(d', d)$. We call $cur(g)$ the **currying** of g. Note that the following equation is a simple consequence of the definitions:

$$g = ev \circ (cur(g) \times id_D).$$

Proposition 5.3.6 The function $ev : [D, E]_{cts} \times D \to E$ is continuous. Let the function $g : D' \times D \to E$ be continuous. Then $cur(g) : D' \to [D, E]_{cts}$ is also continuous.

Suppose that $h : D' \to D$ is continuous, and that E is a cpo. Then the function $h^* : [D, E]_{cts} \to [D', E]_{cts}$ defined by $h^*(f) \stackrel{\text{def}}{=} f \circ h$ is continuous.

Proof A routine exercise. □

[4] Thus $[D, E]_{cts} \subseteq [D, E]_{tot}$; any continuous (total) function is certainly a total function.

Bottom Elements

Definitions 5.3.7 Recall from MC150 that a **bottom** element of a poset D is an element $\bot \in D$ satisfying $\bot \preceq d$ for each element $d \in D$. If D has a bottom element, say $\bot$, then this is unique. For suppose that $\bot'$ were also a bottom element. Then we have $\bot' \preceq \bot$ and also $\bot \preceq \bot'$ and thus by anti-symmetry $\bot = \bot'$. We may write $\bot_D$ for the bottom element of D.

Examples 5.3.8

(1) The poset $\mathcal{P}(A)$ has a bottom, namely $\varnothing$.

(2) The set of partial functions $[A, B]_{par}$ has a bottom, namely the totally undefined function, $\varnothing \subseteq A \times B$.

(3) The discrete cpo on a set A has a bottom iff A has exactly one element.

(4) Clearly 0 is a bottom of the cpo $\mathbb{N}^{\infty}$.

(5) Suppose that D_1 and D_2 are cpos, each with bottom, in which both bottoms are denoted by $\bot$. Then $D_1 \times D_2$ is also a cpo with bottom, with $(\bot, \bot)$ as the bottom element. Check this!

(6) If D and D' are cpos, and D' is also a cpo with bottom, then the set of continuous functions $[D, D']_{cts}$ is a cpo with bottom. The bottom element is given by the function $\lambda_{d \in D}.\bot$, that is the function which maps each $d \in D$ to $\bot \in D'$.

(7) Now go back over each of these examples with pencil and paper and make sure you really understand all of the missing details!!

Lifted Cpos

Definitions 5.3.9 Suppose that D is a cpo. The **lifted** cpo $D_\bot$ has underlying set

$$D_\bot \stackrel{\text{def}}{=} \{\, [d] \mid d \in D \,\} \cup \{\, \bot \,\}$$

where we assume that $d \mapsto [d]$ is an injection, and $\bot \neq [d]$ for any $d \in D$. The partial order is given by

$$x \preceq x' \iff \begin{cases} \text{either } x = \bot \\ \text{or there exists } d, d' \in D, \quad x = [d] \text{ and } x' = [d'] \text{ and } d \preceq d' \end{cases}$$

Thus $\bot$ is a bottom of $D_\bot$, and the injection

$$\iota \stackrel{\text{def}}{=} \lambda_{d \in D}.[d] : D \to D_\bot$$

is **order reflecting** which means that for any $d, d' \in D$, whenever $[d] \preceq [d']$ then $d \preceq d'$.

If $f : D \to E$ is a continuous function where E is a cpo with bottom and D is a cpo, then there is a continuous function $f_\perp : D_\perp \to E$ given by

$$f_\perp(x) \stackrel{\text{def}}{=} \begin{cases} f(d) & \text{if } x = [d] \text{ for some } d \in D \\ \perp & \text{if } x = \perp \end{cases}$$

We call $f_\perp$ the **lift** of the function f. Note that we have $f_\perp \circ \iota = f$, which you may find helpful to picture as a diagram of functions:

$$\begin{array}{ccc} D & \xrightarrow{f} & E \\ {\scriptstyle \iota}\downarrow & \nearrow_{f_\perp} & \\ D_\perp & & \end{array}$$

Finally, note that there is a continuous function $\mathit{lift} : [D, E]_{cts} \to [D_\perp, E]_{cts}$ which is defined by $\mathit{lift}(f) \stackrel{\text{def}}{=} f_\perp$. Thus *lift* maps a continuous function f to its lift. Of course, we should check that all of our assertions about continuity are true; we summarise things in the next proposition:

Proposition 5.3.10 Let D and E be cpos, let E have a bottom, and let $f : D \to E$ be continuous. Then

(i) The injection $\iota : D \to D_\perp$ is continuous.

(ii) The function $f_\perp : D_\perp \to E$ is continuous.

(iii) The function $\mathit{lift} : [D, E]_{cts} \to [D_\perp, E]_{cts}$ is continuous.

Proof We prove (iii) and leave (i) and (ii) as exercises. Suppose that $(c_n \mid n \in \mathbb{N})$ is a chain in $[D, E]_{cts}$ (so that each c_n is a continuous function between D and E). We wish to prove that $\mathit{lift}(\bigvee_{n=0}^{\infty} c_n) = \bigvee_{n=0}^{\infty} \mathit{lift}(c_n)$, that is

$$(\bigvee_{n=0}^{\infty} (c_n))_\perp = \bigvee_{n=0}^{\infty} (c_n)_\perp : D_\perp \longrightarrow E \qquad (\dagger)$$

Let $x \in D_\perp$ be arbitrary. If $x = \perp$, note that $(c_n)_\perp(\perp) = \perp$ for each $n \in \mathbb{N}$ and so

$$(\bigvee_{n=0}^{\infty} c_n)_\perp(x) = (\bigvee_{n=0}^{\infty} c_n)_\perp(\perp) = \perp = \bigvee_{n=0}^{\infty} (c_n)_\perp(\perp) = \bigvee_{n=0}^{\infty} (c_n)_\perp(x) = (\bigvee_{n=0}^{\infty} (c_n)_\perp)(x).$$

If $x = [d]$, note that $(c_n)_\perp(x) = c_n(d)$ and so

$$(\bigvee_{n=0}^{\infty} c_n)_\perp(x) = (\bigvee_{n=0}^{\infty} c_n)(d) = \bigvee_{n=0}^{\infty} c_n(d) = \bigvee_{n=0}^{\infty} (c_n)_\perp(x) = (\bigvee_{n=0}^{\infty} (c_n)_\perp)(x).$$

Thus as x was arbitrary, $(\dagger)$ holds. $\square$

Examples 5.3.11

(1) A Hasse diagram for $\mathbb{B}_\perp = \{\, T, F \,\}_\perp$:

$$
\begin{array}{ccc}
[T] & & [F] \\
 & \diagdown \quad \diagup & \\
 & \perp &
\end{array}
$$

(2) We can regard the *set* of integers $\mathbb{Z}$ as a discrete cpo, rather than as a poset with the "usual" vertical ordering. Thus we have $n \preceq m$ if and only if $n = m$ for all $n, m \in \mathbb{Z}$. Then an informal Hasse diagram for $\mathbb{Z}$ is

$$\ldots \qquad -2 \qquad -1 \qquad 0 \qquad 1 \qquad 2 \qquad \ldots$$

and we call the cpo $\mathbb{Z}$ with the discrete partial order the **flat integers**.

An informal Hasse diagram for the lifted flat integers $\mathbb{Z}_\perp$ is

$$
\begin{array}{ccccccc}
\ldots & [-2] & [-1] & [0] & [1] & [2] & \ldots \\
 & \ldots & & & & \ldots & \\
 & & & \perp & & &
\end{array}
$$

Definitions 5.3.12 Suppose that $f : D_1 \times D_2 \times \ldots \times D_n \to E$ is a continuous function where E is a cpo with bottom, and the D_i are cpos. Then there is a continuous function

$$f_{\perp_{D_i}} : D_1 \times D_2 \times \ldots \times (D_i)_\perp \times \ldots \times D_n \longrightarrow E$$

which is defined by the mapping

$$(d_1, \ldots, x, \ldots, d_n) \mapsto \begin{cases} f(d_1, \ldots, d_i, \ldots, d_n) & \text{if } x = [d_i] \text{ for some } d_i \in D_i \\ \perp_E & \text{if } x = \perp_{D_i} \end{cases}$$

We say that f has been **lifted** on the ith component. We can extend this idea to a multiple number of arguments; we just illustrate for $n = 3$. If $f : D_1 \times D_2 \times D_3 \to E$, then $f_{\perp_{D_1}, \perp_{D_3}} : (D_1)_\perp \times D_2 \times (D_3)_\perp \to E$ is the function which maps (x_1, d_2, x_3) to $\perp_E$ if either x_1 or x_3 is bottom, or if $x_1 = [d_1]$ and $x_3 = [d_3]$ then $f_{\perp_{D_1}, \perp_{D_3}}(x_1, d_2, x_3) = f(d_1, d_2, d_3)$. If we lift f on all its components, then we write $f_{\vec{\perp}}$ for the lifted function.

Examples 5.3.13 Let $p : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_\perp$ be defined by $p(x, y, z) = [x + y + z]$ where $\mathbb{Z}_\perp$ and $\mathbb{Z}$ are the lifted flat integers, and flat integers. Convince yourself that the product cpo $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ is discrete. Note that

$$p_{\vec{\perp}} : \mathbb{Z}_\perp \times \mathbb{Z}_\perp \times \mathbb{Z}_\perp \longrightarrow \mathbb{Z}_\perp$$

and that we have for example $p_{\vec{\perp}}(\perp, \perp, \perp) = \perp$, $p_{\vec{\perp}}([2], [4], \perp) = \perp$ and $p_{\vec{\perp}}([2], [5], [8]) = [15]$. Verify that $p_{\vec{\perp}}$ is continuous!

Conditional Functions

Definitions 5.3.14 If we regard $\mathbb{B}$ as a discrete cpo, then for each cpo D there is a **conditional** function

$$cond : \mathbb{B} \times D \times D \longrightarrow D$$

whose value at any triple $(b, d, d') \in \mathbb{B} \times D \times D$ is given by

$$cond\,(b, d, d') \stackrel{\text{def}}{=} \begin{cases} d & \text{if } b = T \\ d' & \text{if } b = F \end{cases}$$

Least Fixed Points

Motivation 5.3.15 Suppose that D is a cpo with bottom and that $f : D \to D$ is continuous. Let us think about the elements $\bot$, $f(\bot)$, $f^2(\bot) \stackrel{\text{def}}{=} f(f(\bot))$, $f^3(\bot) \ldots$ and so on. Note that

$$\begin{array}{ll} \bot \preceq f(\bot) & \text{since } \bot \text{ is a bottom} \\ f(\bot) \preceq f(f(\bot)) = f^2(\bot) & \text{since } f \text{ is monotone} \\ f^2(\bot) \preceq f^3(\bot) & \text{since } f \text{ is monotone} \\ \vdots & \text{etc etc} \end{array}$$

Thus $\bot \preceq f(\bot) \preceq f^2(\bot) \preceq \ldots$ is a chain in D and so we may define

$$\mathit{fix}(f) \stackrel{\text{def}}{=} \bigvee_{n=0}^{\infty} f^n(\bot)$$

where $f^0(\bot) \stackrel{\text{def}}{=} \bot$ and $f^{n+1}(\bot) \stackrel{\text{def}}{=} f(f^n(\bot))$.

Now, as f is continuous, we have

$$f(\mathit{fix}(f)) = \bigvee_{n=0}^{\infty} f(f^n(\bot)) = \bigvee_{n=0}^{\infty} f^{n+1}(\bot).$$

But it is trivial to see that the join $\bigvee_{n=0}^{\infty} f^{n+1}(\bot)$ must be equal to the join $\bigvee_{n=0}^{\infty} f^n(\bot)$ (just think about the definition of a join) and so

$$f(\mathit{fix}(f)) = \mathit{fix}(f).$$

Thus $\mathit{fix}(f)$ is indeed a fixpoint of f. If $f(d) \preceq d$, we call d a **prefixpoint** of f. Note that for such a $d \in D$, $f^n(\bot) \preceq f^n(d) \preceq d$ and so $\bigvee_{n=0}^{\infty} f^n(\bot) \preceq d$, that is $\mathit{fix}(f) \preceq d$. In particular, if $f(d) = d$, then $f(d) \preceq d$ and so $\mathit{fix}(f) \preceq d$. Thus $\mathit{fix}(f)$ is the least element in the set of prefixpoints of f, and also the least element in the set of all fixpoints of f.

Theorem 5.3.16 Given a cpo with bottom D and a continuous function $f : D \to D$, then f has a least fixpoint which is given by

$$\mathit{fix}(f) \stackrel{\text{def}}{=} \bigvee_{n=0}^{\infty} f^n(\bot)$$

where $f^0(\bot) \stackrel{\text{def}}{=} \bot$ and $f^{n+1}(\bot) \stackrel{\text{def}}{=} f(f^n(\bot))$. Thus $f(\mathit{fix}(f)) = \mathit{fix}(f)$, and for any $d \in D$ if $f(d) = d$ then $\mathit{fix}(f) \preceq d$. Moreover, if $f(d) \preceq d$ then $\mathit{fix}(f) \preceq d$.

Proof See above. □

Motivation 5.3.17 We summarise all of the results of this section in Table 5.1.

5.4 Denotational Semantics of $\mathbb{IMP}$

Motivation 5.4.1 We can now complete the results of Chapter 3 by proving Proposition 3.2.12.

Proof The relation $\preceq$ defined on $[\mathit{States}, \mathit{States}_\bot]_{tot}$ on page 34 coincides (on restriction to the continuous functions) with the partial ordering on the set of continuous functions $[\mathit{States}, \mathit{States}_\bot]_{cts}$, where States is the discrete cpo on the *set* of states, and $\mathit{States}_\bot$ is the lifted cpo. Check the coincidence!! Note that $[\mathit{States}, \mathit{States}_\bot]_{cts}$ is a cpo with bottom, as $\mathit{States}_\bot$ (trivially) has a bottom. Hence Proposition 3.2.12 is a special case of Theorem 5.3.16, provided that

$$\Phi : [\mathit{States}, \mathit{States}_\bot]_{tot} \longrightarrow [\mathit{States}, \mathit{States}_\bot]_{tot}$$

is continuous. Recall that

$$\Phi = \lambda_{g \in [\mathit{States}, \mathit{States}_\bot]_{tot}}.\lambda_{s \in \mathit{States}}.\mathit{cond}\,([\![be]\!](s), g_\bot([\![co]\!](s)), [s]).$$

Now consider the following function Γ:

$$[\mathit{States}, \mathit{States}_\bot]_{cts} \times \mathit{States} \xrightarrow{\mathit{cond} \circ \langle [\![be]\!] \circ \mathit{snd},\, \mathit{ev} \circ (([\![co]\!]^* \circ \mathit{lift}) \times \mathit{id}_{\mathit{States}}),\, \iota \circ \mathit{snd} \rangle} \mathit{States}_\bot$$

where

$$\mathit{ev} : [\mathit{States}, \mathit{States}_\bot]_{tot} \times \mathit{States} \longrightarrow \mathit{States}_\bot$$

and

$$\mathit{cond} : \mathbb{B} \times \mathit{States}_\bot \times \mathit{States}_\bot \longrightarrow \mathit{States}_\bot.$$

If (g, s) is an arbitrary element in $[\mathit{States}, \mathit{States}_\bot]_{cts} \times \mathit{States}$ it is easy to check that $\Gamma(g, s) = \Phi(g)(s)$. Thus of course $\Phi = \mathit{cur}(\Gamma)$. It follows from Table 5.1 that Φ is indeed continuous, because the functions

$$[\![be]\!] : \mathit{States} \to \mathbb{B} \quad \text{and} \quad [\![co]\!] : \mathit{States} \to \mathit{States}_\bot$$

are continuous as States is discrete, and thus Φ is built up from continuous functions.

□

Suppose that D, E, F, D_1, D_2, $\ldots$ D_n are each cpos, that S is a set regarded as a discrete cpo, and that we have continuous functions

$l : D' \to D$	$f : D \to E$	$g : E \to F$
$k : D_1 \times D_2 \to E$	$(f_i : E \to D_i$	$1 \leq i \leq n)$
$k' : \Pi_{i=1}^{i=n} D_i \to E$		

Then the following functions (which are built out of the above functions) are also continuous:

(i) $id_D : D \to D$.

(ii) $g \circ f : D \to F$.

(iii) $h : S \to D$ where h is *any* function between S and the underlying set of D.

(iv) $h : D \to E$ where h is any *constant* function such that $h(d) \stackrel{\text{def}}{=} e_0$ for all $d \in D$ and fixed $e_0 \in E$.

(v) $l^* : [D, E]_{cts} \to [D', E]_{cts}$.

(vi) $fst : D \times E \to D$ and $snd : D \times E \to E$.

(vii) $proj_i : \Pi_{i=1}^{i=n} D_i \to D_i$.

(viii) $\langle f_1, \ldots, f_n \rangle : E \to \Pi_{i=1}^{n} D_i$.

(ix) $ev : [D, E]_{cts} \times D \to E$.

(x) $cur(k) : D_1 \to [D_2, E]_{cts}$.

(xi) $\iota : D \to D_\perp$.

(xii) $f_\perp : D_\perp \to E$ provided E has a bottom.

(xiii) $k'_{\vec{\perp}} : \Pi_{i=1}^{n} (D_i)_\perp \to E$ provided E has a bottom.

(xiv) $lift : [D, E]_{cts} \to [D_\perp, E]_{cts}$ provided E has a bottom.

(xv) $cond : \mathbb{B} \times D \times D \to D$.

Table 5.1: Properties of Continuous Functions

6
Operational Semantics of Functional Languages

6.1 Introduction

Motivation 6.1.1 In this chapter we turn our attention to functional programming languages. Such languages provide a syntax of expressions in which one can write down functions directly, much as they appear in mathematics, without having to think about how to code them as commands acting on a state. In fact the simple functional languages we meet here do not have any kind of state: a program is an expression which potentially denotes a value which can be returned to the programmer. The operational semantics gives rules for reducing programs to values. In this chapter we shall study the syntax and operational semantics of four small functional programming languages.

6.2 Types and Expressions for $\mathbb{FUN}^e$

Motivation 6.2.1 We begin by defining the types and expressions of a simple language called $\mathbb{FUN}^e$. (We shall assume that readers have some familiarity with the datatypes of functions, pairs and lists. If not, consult the course notes for MC 208.) We then briefly recall the idea of a functional programming language. Such a language provides a syntax within which one can write down functions much as we do in mathematics. The language has no concept of state. Every expression of the language can be thought of as a data-value (as against, say, a command) and the language executes by simplifying complex expressions to much simpler expressions. The simpler expressions are returned as output to the programmer.

Definitions 6.2.2 The types of the language $\mathbb{FUN}^e$ are given by the grammar

$$\sigma \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \sigma \to \sigma \mid (\sigma, \sigma) \mid [\sigma]$$

We shall write *Type* for the set of types. Thus $\mathbb{FUN}^e$ contains the types of integers, Booleans, (higher order) functions, (binary) products and lists. We shall write

$$\sigma_1 \to \sigma_2 \to \sigma_3 \to \ \ldots \to \sigma_n \to \sigma$$

for

$$\sigma_1 \to (\sigma_2 \to (\sigma_3 \to (\ \ldots \to (\sigma_n \to \sigma) \ldots))).$$

Thus for example $\sigma_1 \to \sigma_2 \to \sigma_3$ means $\sigma_1 \to (\sigma_2 \to \sigma_3)$.

Let *Var* be a fixed set of **variables**. We shall also need a fixed set of **function identifiers** *FId*, with typical elements denoted by F. These symbols will be used to define higher order functions in $\mathbb{FUN}^e$ —compare

$$\mathsf{F}xx' = x + x' \quad \text{in } \mathbb{FUN}^e \qquad \text{to} \qquad \texttt{fxy=x+y} \quad \text{in Miranda.}$$

The **expressions** of the functional language $\mathbb{FUN}^e$ are given by the grammar[1]

E	$::=$	x	variables
	$\mid$	$\underline{c}$	constant
	$\mid$	F	function identifier
	$\mid$	$E_1 \ op \ E_2$	operator
	$\mid$	$\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3$	conditional
	$\mid$	(E_1, E_2)	pairing
	$\mid$	$\mathsf{fst}(E)$	first projection
	$\mid$	$\mathsf{snd}(E)$	second projection
	$\mid$	$E_1\, E_2$	function application
	$\mid$	nil_σ	empty list
	$\mid$	$\mathsf{hd}(E)$	head of list
	$\mid$	$\mathsf{tl}(E)$	tail of list
	$\mid$	$E_1 : E_2$	cons for lists
	$\mid$	$\mathsf{elist}(E)$	test for empty list

Remark 6.2.3 We shall adopt a few conventions to make expressions more readable:

• The expressions of the language are in fact finite trees. We have not bothered to make this explicit, as we assume the reader is by now used to this fact. For example, $\mathsf{fst}(E)$ denotes a tree whose root is a constructor symbol fst and whose single immediate subtree is E. Note that the formal definition of expressions as (certain obvious) finite trees makes various desirable equations hold. Thus for example if $\mathsf{fst}(E_1) \equiv \mathsf{fst}(E_2)$, then we must have $E_1 \equiv E_2$ (make sure you understand this!).

• In general, we shall write our "formal" syntax in an informal manner, using brackets "(" and ")" to disambiguate where appropriate—recall that in Miranda one can add such brackets to structure programs. So for example, if we apply E_2 to E_3 to get $E_2\, E_3$, and then apply E_1 to the latter expression, we write this as $E_1\,(E_2\, E_3)$.

• $E_1 E_2 E_3 \ldots E_n$ is shorthand for $(\ldots((E_1 E_2) E_3)\ldots)\, E_n$. We say that application **associates** to the left. For example, $E_1 E_2 E_3$ is short for $(E_1\, E_2)\, E_3$. Note that if we made the tree structure of applications explicit, rather than using the sugared notation $E\, E'$ instead of, say, $\mathsf{ap}(E, E')$, then $(E_1\, E_2)\, E_3$ would be a shorthand notation for the tree denoted by $\mathsf{ap}(\mathsf{ap}(E_1, E_2), E_3)$.

• The integer valued integer operators also associate to the left; thus we will write (for example) $\underline{n} + \underline{m} + \underline{l}$ to mean $(\underline{n} + \underline{m}) + \underline{l}$, with the obvious extension to a finite number of integer constants.

• The cons constructor associates to the right. So, for example, we shall write $E_1 : E_2 : E_3$ for $E_1 : (E_2 : E_3)$. This is what one would expect—the "head of the list" is appended to the "tail of the list". (Recall that lists such as $[\underline{1}, \underline{4}, \underline{6}]$, which one often finds in real languages, would correspond to the $\mathbb{FUN}^e$ list $\underline{1} : \underline{4} : \underline{6} : \mathsf{nil}_{\mathsf{int}}$).

[1] As usual, $op \in \{+, -, *, \leq, <\}$. If E_1 represents a function f, and E_2 an argument a, then $E_1\, E_2$ represents $f(a)$.

• Try writing out each of the general expression forms as finite trees.

Definitions 6.2.4 For an expression E, the set $\mathit{fvar}(E)$ consists of the variables which appear in E. Being a set, a variable may appear in E many times, but is only recorded once in $\mathit{fvar}(E)$. We could give a formal recursive definition of $\mathit{fvar}(E)$ using the inductive definition of E, but we do not bother with this. We shall also talk of the function identifiers which appear in E. Again, we do not bother with a formal definition, as there are no technical difficulties such as bound function identifiers in our simple setting.

If E and $E_1, \dots, E_n$ are expressions, then $E[E_1, \dots, E_n/x_1, \dots, x_n]$ denotes the expression E with E_i replacing x_i for each $1 \leq i \leq n$. (We omit the proof that the finite tree $E[E_1, \dots, E_n/x_1, \dots, x_n]$ is indeed an expression).

Examples 6.2.5 Examples of expressions are

(1) x;

(2) $\mathsf{hd}(\underline{2} : \underline{4})$;

(3) $x\,(x\,y)$;

(4) $x : \underline{2} : \underline{3}$.

(5) $\mathsf{F}\underline{2}\,\underline{3}$.

(6) $\mathit{fvar}(x : (\underline{2} + y + y) : (z\,x)) = \{\, x, y, z \,\}$.

(7) $(xy : \underline{2} : z)[\mathsf{F}, \underline{2} + \underline{3}, z/x, y, z] = \mathsf{F}(\underline{2} + \underline{3}) : \underline{2} : z$.

Definitions 6.2.6 A **context** Γ is a finite set of (variable, type) pairs, where the type is a $\mathbb{FUN}^e$ type, and the variables are required to be *distinct* so that one does not assign two different types to the same variable. So for example $\Gamma = \{\, (x_1, \sigma_1), \dots, (x_n, \sigma_n) \,\}$. We usually write a typical pair (x, σ) as $x :: \sigma$, and a typical context as

$$\Gamma = x_1 :: \sigma_1, \dots, x_n :: \sigma_n.$$

Note that a context is by definition a set, so the order of the $x_i :: \sigma_i$ does not matter and we omit curly braces simply to cut down on notation. We write $\Gamma, \Gamma' \stackrel{\text{def}}{=} \Gamma \cup \Gamma'$ and $\Gamma, x :: \sigma \stackrel{\text{def}}{=} \Gamma \cup \{\, x :: \sigma \,\}$.

A **function arity** is a type of the form $\sigma_1 \to \sigma_2 \to \sigma_3 \to \ \dots \to \sigma_k \to \sigma$ where k is a non-zero natural number. You should think of such a function arity as typing information for a function identifier: the function takes k inputs with types σ_i and gives an output of type σ. We shall denote function arities by the Greek letter α.

A **function environment** is specified by a finite set of (function identifier, function arity) pairs, with a typical function environment being denoted by

$$\Delta = \mathsf{F}_1 :: \alpha_1, \dots, \mathsf{F}_m :: \alpha_m.$$

We say that α_i is the function arity of F_i, and that if α_i is

$$\sigma_1 \to \sigma_2 \to \sigma_3 \to \ldots \to \sigma_k \to \sigma$$

then we refer to k as the **numerical arity** of F_i.

We shall say that a variable x **appears** in a context Γ if $x :: \sigma \in \Gamma$ for some type σ. Thus z appears in $x :: \mathsf{int}, y :: [\mathsf{bool}], z :: \mathsf{int} \to \mathsf{int}$. We shall similarly say that a type *appears* in a context, or that a function identifier or arity *appears* in a function environment.

Example 6.2.7 A simple example of a function environment is

$$\Delta \stackrel{\text{def}}{=} \mathsf{map} :: (\mathsf{int} \to \mathsf{int}) \to [\mathsf{int}] \to [\mathsf{int}],\ \ \mathsf{suc} :: \mathsf{int} \to \mathsf{int}$$

which declares the arities of the map function and the successor function. Note that $(\mathsf{int} \to \mathsf{int}) \to [\mathsf{int}] \to [\mathsf{int}]$ is the function arity of map; its numerical arity is 2. Another simple example of a function environment is $\mathsf{plus} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}$.

Motivation 6.2.8 Given a context Γ of typed variables, and a function environment Δ, we can build up expressions E which use only variables and function identifiers which appear in Γ and Δ. This is how we usually write (functional) programs: we first declare variables and types, possibly also functions and types, and then write our program E which uses these data. We shall define judgements of the form $\Delta \mid \Gamma \vdash E :: \sigma$ which should be understood as follows: given the function environment Δ, and the context Γ of variable typings, then the expression E is well formed and has type σ. Given Δ and Γ, we say that E is **assigned** the type σ. We call $\Delta \mid \Gamma \vdash E :: \sigma$ a **type assignment** relation.

Definitions 6.2.9 We shall inductively define a type assignment (ternary) relation which takes the form $\Delta \mid \Gamma \vdash E :: \sigma$ using the rules in Table 6.1.

Proposition 6.2.10 If $\Delta \mid \Gamma \vdash E :: \sigma$, then the function identifiers which appear in E appear in Δ, and the variables which appear in E appear in Γ.

Suppose that Δ, Δ' is a function environment, that Γ, Γ' is a context, and that $\Delta \mid \Gamma \vdash E :: \sigma$. Then in fact $\Delta, \Delta' \mid \Gamma, \Gamma' \vdash E :: \sigma$ is also a derivable type assignment.

Proof Follows by a simple Rule Induction. Exercise! □

Remark 6.2.11 Note that if $\Delta \mid \Gamma \vdash E :: \sigma$, then Δ and Γ may contain function identifiers and variables which do not (necessarily) appear in E. For example

$$\mathsf{F} :: \mathsf{int} \to \mathsf{int} \mid x :: \mathsf{int}, y :: \mathsf{int}, z :: \mathsf{int} \vdash \mathsf{F}x :: \mathsf{int}$$

is a valid type assignment. The motto is "just because we declared some variables or function identifiers, does not mean we need to program with them".

Note that the second part of the previous proposition says that "given any type assignment, we can add function identifiers to the function environment, and variables to the context, without changing the type of E." Sometimes this is called **weakening**.

$$\frac{}{\Delta \mid \Gamma \vdash x :: \sigma} \text{(where } x :: \sigma \in \Gamma) \quad \vdash \text{VAR} \qquad \frac{}{\Delta \mid \Gamma \vdash \underline{n} :: \mathsf{int}} \vdash \text{INT}$$

$$\frac{}{\Delta \mid \Gamma \vdash \underline{T} :: \mathsf{bool}} \vdash \text{TRUE} \qquad \frac{}{\Delta \mid \Gamma \vdash \underline{F} :: \mathsf{bool}} \vdash \text{FALSE}$$

$$\frac{\Delta \mid \Gamma \vdash E_1 :: \mathsf{int} \quad \Delta \mid \Gamma \vdash E_2 :: \mathsf{int}}{\Delta \mid \Gamma \vdash E_1 \; op \; E_2 :: \mathsf{int}} \text{(where } \mathsf{op} \text{ is integer valued)} \quad \vdash \text{OP}_1$$

$$\frac{\Delta \mid \Gamma \vdash E_1 :: \mathsf{int} \quad \Delta \mid \Gamma \vdash E_2 :: \mathsf{int}}{\Delta \mid \Gamma \vdash E_1 \; op \; E_2 :: \mathsf{bool}} \text{(where } \mathsf{op} \text{ is Boolean valued)} \quad \vdash \text{OP}_2$$

$$\frac{\Delta \mid \Gamma \vdash E_1 :: \mathsf{bool} \quad \Delta \mid \Gamma \vdash E_2 :: \sigma \quad \Delta \mid \Gamma \vdash E_3 :: \sigma}{\Delta \mid \Gamma \vdash \mathsf{if} \; E_1 \; \mathsf{then} \; E_2 \; \mathsf{else} \; E_3 :: \sigma} \vdash \text{COND}$$

$$\frac{\Delta \mid \Gamma \vdash E_1 :: \sigma_2 \to \sigma_1 \quad \Delta \mid \Gamma \vdash E_2 :: \sigma_2}{\Delta \mid \Gamma \vdash E_1 \; E_2 :: \sigma_1} \vdash \text{AP}$$

$$\frac{\Delta \mid \Gamma \vdash E_1 :: \sigma_1 \quad \Delta \mid \Gamma \vdash E_2 :: \sigma_2}{\Delta \mid \Gamma \vdash (E_1, E_2) :: (\sigma_1, \sigma_2)} \vdash \text{PAIR}$$

$$\frac{\Delta \mid \Gamma \vdash E :: (\sigma_1, \sigma_2)}{\Delta \mid \Gamma \vdash \mathsf{fst}(E) :: \sigma_1} \vdash \text{FST} \qquad \frac{\Delta \mid \Gamma \vdash E :: (\sigma_1, \sigma_2)}{\Delta \mid \Gamma \vdash \mathsf{snd}(E) :: \sigma_2} \vdash \text{SND}$$

$$\frac{}{\Delta \mid \Gamma \vdash \mathsf{F} :: \alpha} \text{(where } \mathsf{F} :: \alpha \in \Delta) \quad \vdash \text{FAP}$$

$$\frac{}{\Delta \mid \Gamma \vdash \mathsf{nil}_\sigma :: [\sigma]} \vdash \text{NIL} \qquad \frac{\Delta \mid \Gamma \vdash E_1 :: \sigma \quad \Delta \mid \Gamma \vdash E_2 :: [\sigma]}{\Delta \mid \Gamma \vdash E_1 : E_2 :: [\sigma]} \vdash \text{CONS}$$

$$\frac{\Delta \mid \Gamma \vdash E :: [\sigma]}{\Delta \mid \Gamma \vdash \mathsf{hd}(E) :: \sigma} \vdash \text{HD} \qquad \frac{\Delta \mid \Gamma \vdash E :: [\sigma]}{\Delta \mid \Gamma \vdash \mathsf{tl}(E) :: [\sigma]} \vdash \text{TL} \qquad \frac{\Delta \mid \Gamma \vdash E :: [\sigma]}{\Delta \mid \Gamma \vdash \mathsf{elist}(E) :: \mathsf{bool}} \vdash \text{ELIST}$$

Table 6.1: Type Assignment Relation $\Delta \mid \Gamma \vdash E :: \sigma$ in $\mathbb{FUN}^e$

Proposition 6.2.12 Given a function environment Δ, a context Γ and an expression E, if there is a type σ for which $\Delta \mid \Gamma \vdash E :: \sigma$, then such a type is unique. Thus $\mathbb{FUN}^e$ is **monomorphic**.

Proof We can prove this using Rule Induction. In fact we verify that

$$\text{for all } \Delta \mid \Gamma \vdash E :: \sigma_1, \quad \text{for all } \sigma_2, \quad (\Delta \mid \Gamma \vdash E :: \sigma_2 \Longrightarrow \sigma_1 = \sigma_2).$$

We check property closure for the rule HD: The inductive hypothesis is that

$$\text{for all } \sigma_2, \quad (\Delta \mid \Gamma \vdash E :: \sigma_2 \Longrightarrow [\sigma] = \sigma_2)$$

where $\Delta \mid \Gamma \vdash E :: [\sigma]$. We wish to prove that

$$\text{for all } \sigma_2, \quad (\Delta \mid \Gamma \vdash \mathsf{hd}(E) :: \sigma_2 \Longrightarrow \sigma = \sigma_2)$$

where $\Delta \mid \Gamma \vdash \mathsf{hd}(E) :: \sigma$.

Let σ_2 be arbitrary, where $\Delta \mid \Gamma \vdash \mathsf{hd}(E) :: \sigma_2$. Then we must have $\Delta \mid \Gamma \vdash E :: [\sigma_2]$. From the inductive hypothesis we see that $[\sigma] = [\sigma_2]$. It follows that $\sigma = \sigma_2$ as required.

Property closure of the remaining rules is left as an exercise. □

Examples 6.2.13

(1) With Δ as in Example 6.2.7, we have

$$\Delta \mid x :: \mathsf{int}, y :: \mathsf{int}, z :: \mathsf{int} \vdash \mathsf{map}\,\mathsf{suc}\,(x : y : z : \mathsf{nil}_{\mathsf{int}}) :: [\mathsf{int}]$$

(2) We have

$$\mathsf{twicehead} :: [\mathsf{int}] \to \mathsf{int} \to (\mathsf{int}, \mathsf{int}) \mid y :: [\mathsf{int}], x :: \mathsf{int} \vdash \mathsf{twicehead}\, y\, x :: (\mathsf{int}, \mathsf{int})$$

(3) We have

$$\varnothing \mid \varnothing \vdash \mathsf{if}\ \underline{T}\ \mathsf{then}\ \mathsf{fst}((\underline{2} : \mathsf{nil}_{\mathsf{int}}, \mathsf{nil}_{\mathsf{int}}))\ \mathsf{else}\ (\underline{2} : \underline{6} : \mathsf{nil}_{\mathsf{int}}) :: [\mathsf{int}]$$

6.3 Function Declarations and Programs for $\mathbb{FUN}^e$

Motivation 6.3.1 A *function declaration* is a method for declaring that certain function identifiers have certain meanings. We look at two examples:

We begin by specifying a function environment, such as $\mathsf{plus} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}$ or $\mathsf{fac} :: \mathsf{int} \to \mathsf{int}$. Then to declare that plus is a function which takes a pair of integers and adds them, we write $\mathsf{plus}\, x = \mathsf{fst}(x) + \mathsf{snd}(x)$. To declare that fac denotes the factorial function, we would like

$$\mathsf{fac}\, x = \mathsf{if}\ x < \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x * \mathsf{fac}(x - \underline{1})$$

Thus in general, if F is a function identifier, we might write $\mathsf{F}x = E$ where E is an expression which denotes "the result" of applying F to x. In $\mathbb{FUN}^e$, we are able to specify statements such as $\mathsf{F}x = E$ which are regarded as preliminary data to writing a program—we *declare* the definitions of certain functions. The language is then able to provide the user with functions F whose action is specified by the expression E. Each occurrence of F in program code executes using its declared definition. This is exactly like Miranda.

In general, a function declaration will specify the action of a finite number of function identifiers, and moreover the definitions can be mutually recursive—each function may be defined in terms of the others. Note that the factorial function given above is defined recursively: the identifier fac actually appears in the expression giving "the result" of the function.

A *program* in $\mathbb{FUN}^e$ is an expression in which there are no variables and each of the function identifiers appearing in the expression have been declared. The idea is that a program is an expression in which there is no "missing data" and thus the expression can be "evaluated" as it stands. A *value* is an "evaluated program". It is an expression which has a particularly simple form, such as an integer, or a list of integers, and thus is a sensible item of data to return to a user. We now make all of these ideas precise.

Definitions 6.3.2 A **function declaration** dec_Δ, where $\Delta = \mathsf{F}_1 :: \alpha_1, \ldots, \mathsf{F}_m :: \alpha_m$ is a given, fixed, function environment for which

$$\alpha_j = \sigma_{j1} \to \sigma_{j2} \to \sigma_{j3} \to \ \ldots \to \sigma_{jk_j} \to \sigma_j$$

consists of the following data:

$$\begin{array}{lclll}
\mathsf{F}_1\, x_{11} \ldots x_{1k_1} & = & E_{\mathsf{F}_1} & \text{where} & \Delta \mid x_{11} :: \sigma_{11}, \ldots, x_{1k_1} :: \sigma_{1k_1} \vdash E_{\mathsf{F}_1} :: \sigma_1 \\
\mathsf{F}_2\, x_{21} \ldots x_{2k_2} & = & E_{\mathsf{F}_2} & \text{where} & \Delta \mid x_{21} :: \sigma_{21}, \ldots, x_{2k_2} :: \sigma_{2k_2} \vdash E_{\mathsf{F}_2} :: \sigma_2 \\
 & \vdots & & & \\
\mathsf{F}_j\, x_{j1} \ldots x_{jk_j} & = & E_{\mathsf{F}_j} & \text{where} & \Delta \mid x_{j1} :: \sigma_{j1}, \ldots, x_{jk_j} :: \sigma_{jk_j} \vdash E_{\mathsf{F}_j} :: \sigma_j \\
 & \vdots & & & \\
\mathsf{F}_m\, x_{m1} \ldots x_{mk_m} & = & E_{\mathsf{F}_m} & \text{where} & \Delta \mid x_{m1} :: \sigma_{m1}, \ldots, x_{mk_m} :: \sigma_{mk_m} \vdash E_{\mathsf{F}_m} :: \sigma_m
\end{array}$$

Note that the data which are *specified* in dec_Δ just consist of the declarations $\mathsf{F}_j\vec{x} = E_{\mathsf{F}_j}$; the type assignments just need to hold of the specified E_{F_j}. We shall sometimes abbreviate the jth type assignment to $\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E_{\mathsf{F}_j} :: \sigma_j$. We call the expression E_{F_j} the **definitional body** of F_j. Note that the type assignments force each of the variables in $\{ x_{j1}, \ldots, x_{jk_j} \}$ to be *distinct* (for each $j \in \{ 1, \ldots, m \}$).

We define a **program expression** P to be any expression for which $fvar(P) = \varnothing$, that is, no variables occur in P. A **program** in $\mathbb{FUN}^e$ is a judgement of the form

$$dec_\Delta \quad in \ \ P$$

where dec_Δ is a given function declaration and the program expression P satisfies a type assignment of the form

$$\Delta \mid \varnothing \vdash P :: \sigma.$$

We may sometimes simply refer to P as a program, when no confusion can arise from this. We call σ the type of the program $dec_\Delta \quad in \quad P$ (and sometimes just say σ is the type of P).

Examples 6.3.3

(1) Let $\Delta = \mathsf{F}_1 :: [\mathsf{int}] \to \mathsf{int} \to \mathsf{int}, \mathsf{F}_2 :: \mathsf{int} \to \mathsf{int}$. Then an example of a function declaration dec_Δ is

$$\begin{aligned} \mathsf{F}_1 x_{11} x_{12} &= \mathsf{hd}(\mathsf{tl}(\mathsf{tl}(x_{11}))) + \mathsf{F}_2 x_{12} \\ \mathsf{F}_2 x_{21} &= x_{21} * x_{21} \end{aligned}$$

Note that here we labelled the variables with subscripts to match the general definition of function declaration—in future we will not bother to do this. It is easy to see that the declaration is well defined: for example $\Delta \mid x_{21} :: \mathsf{int} \vdash x_{21} * x_{21} :: \mathsf{int}$.

(2) Let Δ be $\mathsf{F} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$. Then we have a declaration dec_Δ

$$\mathsf{F}\, x\, y\, z = x + y + z$$

where of course $\Delta \mid x :: \mathsf{int}, y :: \mathsf{int}, z :: \mathsf{int} \vdash x + y + z :: \mathsf{int}$.

(3) The next few examples are all programs

$$\mathsf{F}\, x = \mathsf{if}\ x \leq \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x * \mathsf{F}\,(x - \underline{1}) \quad in \quad \mathsf{F}\, \underline{4}$$

(4)

$$\left.\begin{aligned} \mathsf{F}_1\, x\, y\, z &= \mathsf{if}\ x \leq \underline{1}\ \mathsf{then}\ y\ \mathsf{else}\ z \\ \mathsf{F}_2\, x &= \mathsf{F}_1\, x\, \underline{1}\, (x * \mathsf{F}_2\,(x - \underline{1})) \end{aligned}\right\} \quad in \quad \mathsf{F}_2\, \underline{4}$$

(5)

$$\left.\begin{aligned} \mathsf{Ev}\, x &= \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{T}\ \mathsf{else}\ \mathsf{Od}\,(x - \underline{1}) \\ \mathsf{Od}\, x &= \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{F}\ \mathsf{else}\ \mathsf{Ev}\,(x - \underline{1}) \end{aligned}\right\} \quad in \quad \mathsf{Ev}\, \underline{12}$$

Note that Ev and Od are defined by *mutual recursion*, and that they only correctly determine the evenness or oddness of non-negative integers. How would you correct this deficiency?

(6) $\mathsf{F}\, x = \mathsf{F}\, x \quad in \quad \mathsf{F}\,(\underline{3} : \mathsf{nil}_{\mathsf{int}})$ is a program which does not evaluate to a value; the program *loops*.

6.4 Operational Semantics for $\mathbb{FUN}^e$

Definitions 6.4.1 Let dec_Δ be a function declaration. A **value expression** is any expression V which can be produced by the following grammar

$$V ::= \underline{c} \mid \mathsf{nil}_\sigma \mid (V, V') \mid \mathsf{F}_j\, \vec{V} \mid V : V'$$

where c is any Boolean or integer, σ is any type, and $\vec{V}$ is an application of the form $V_1\ V_2 \ldots V_{l-1}\ V_l$ where $0 \leq l < k_j$, and k_j is the numerical arity of F_j. *Note* that l is *strictly* less than k_j. See Remark 6.4.4. A **value** is any value expression for which $dec_\Delta \quad in \quad V$ is a valid $\mathbb{FUN}^e$ program.

Motivation 6.4.2 The operational semantics of $\mathbb{FUN}^e$ gives rules for proving that a program P evaluates to a value V within a given function declaration dec_Δ. For any given function declaration, we write this as $dec_\Delta \vdash P \Downarrow^e V$, and a trivial example is, say, $dec_\Delta \vdash \underline{3} + \underline{4} + \underline{10} \Downarrow^e \underline{17}$.

This is an **eager** or **call-by-value** language. This means that when expressions are evaluated, their arguments (or sub-expressions) are fully evaluated before the whole expression is evaluated. We give a couple of examples:

In evaluating a function application $\mathsf{F} P_1 P_2$ we first compute values for P_1 and P_2, say V_1 and V_2, and then evaluate $\mathsf{F}\, V_1\, V_2$. In evaluating a pair (P_1, P_2), we compute values for P_1 and P_2, say V_1 and V_2, giving a final value of (V_1, V_2).

Definitions 6.4.3 We shall define an **evaluation relation** whose judgements will take the form

$$dec_\Delta \vdash P \Downarrow^e V$$

where P and V are respectively a program expression and value expression whose function identifiers appear in the function declaration dec_Δ. The rules for inductively generating these judgements are given by the rules in Table 6.2.

Remark 6.4.4 You may find the definition of $\mathsf{F}\,\vec{V}$ as a value expression rather odd. In fact, there is good reason for the definition. The basic idea behind the definition of a value is that "values are those expressions which are as fully evaluated as possible, according to the call-by-value execution strategy". This explains why $\mathsf{F}\,\vec{V}$ is indeed a value expression; a small example will clarify:

Suppose that

$$\mathsf{F} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int},$$

and that P_1 and P_2 are integer programs, which compute to the values $\underline{n_1}$ and $\underline{n_2}$. Then $\mathsf{F}\, P_1$ is not a value, because the language is eager. It will evaluate to $\mathsf{F}\, \underline{n_1}$. But this latter expression cannot be evaluated any further—informally, the function F cannot itself be called until it is applied to three integer arguments. Thus $\mathsf{F}\, \underline{n_1}$ is a value. Giving it the argument P_2, we have a program $\mathsf{F}\, \underline{n_1}\, P_2$ which evaluates to the value $\mathsf{F}\, \underline{n_1}\, \underline{n_2}$. Again, we have a value, as the expression cannot be computed any further. Finally, however, we can supply a third argument to $\mathsf{F}\, \underline{n_1}\, \underline{n_2}$ giving $\mathsf{F}\, \underline{n_1}\, \underline{n_2}\, P_3$. This evaluates to $\mathsf{F}\, \underline{n_1}\, \underline{n_2}\, \underline{n_3}$, and at last F has its full quota of three arguments—thus the latter expression is not a value as we can now compute the function F using rule $\Downarrow^e$ FAP.

Examples 6.4.5

$$\frac{}{dec_\Delta \vdash V \Downarrow^e V} \Downarrow^e\text{VAL} \qquad \frac{dec_\Delta \vdash P_1 \Downarrow^e \underline{m} \quad dec_\Delta \vdash P_2 \Downarrow^e \underline{n}}{dec_\Delta \vdash P_1 \ op \ P_2 \Downarrow^e \underline{m \ op \ n}} \Downarrow^e\text{OP}$$

$$\frac{dec_\Delta \vdash P_1 \Downarrow^e \underline{T} \quad dec_\Delta \vdash P_2 \Downarrow^e V}{dec_\Delta \vdash \mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V} \Downarrow^e\text{COND}_1$$

$$\frac{dec_\Delta \vdash P_1 \Downarrow^e \underline{F} \quad dec_\Delta \vdash P_3 \Downarrow^e V}{dec_\Delta \vdash \mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V} \Downarrow^e\text{COND}_2$$

$$\frac{dec_\Delta \vdash P_1 \Downarrow^e V_1 \quad dec_\Delta \vdash P_2 \Downarrow^e V_2}{dec_\Delta \vdash (P_1, P_2) \Downarrow^e (V_1, V_2)} \Downarrow^e\text{PAIR}$$

$$\frac{dec_\Delta \vdash P \Downarrow^e (V_1, V_2)}{dec_\Delta \vdash \mathsf{fst}(P) \Downarrow^e V_1} \Downarrow^e\text{FST} \qquad \frac{dec_\Delta \vdash P \Downarrow^e (V_1, V_2)}{dec_\Delta \vdash \mathsf{snd}(P) \Downarrow^e V_2} \Downarrow^e\text{SND}$$

$$\frac{\begin{cases} dec_\Delta \vdash P_1 \Downarrow^e \mathsf{F}\ \vec{V} \quad dec_\Delta \vdash P_2 \Downarrow^e V_2 \quad dec_\Delta \vdash \mathsf{F}\ \vec{V}\ V_2 \Downarrow^e V \\ \text{where either } P_1 \text{ or } P_2 \text{ is not a value} \end{cases}}{dec_\Delta \vdash P_1\, P_2 \Downarrow^e V} \Downarrow^e\text{AP}$$

$$\frac{dec_\Delta \vdash E_{\mathsf{F}_j}[V_1, \ldots, V_{k_j}/x_1, \ldots, x_{k_j}] \Downarrow^e V}{dec_\Delta \vdash \mathsf{F}_j\, V_1 \ldots V_{k_j} \Downarrow^e V} [\mathsf{F}_j \vec{x} = E_{\mathsf{F}_j} \text{ declared in } dec_\Delta] \Downarrow^e\text{FAP}$$

$$\frac{dec_\Delta \vdash P \Downarrow^e \mathsf{nil}_\sigma}{dec_\Delta \vdash \mathsf{tl}(P) \Downarrow^e \mathsf{nil}_\sigma} \Downarrow^e\text{NIL} \qquad \frac{dec_\Delta \vdash P \Downarrow^e V : V'}{dec_\Delta \vdash \mathsf{hd}(P) \Downarrow^e V} \Downarrow^e\text{HD} \qquad \frac{dec_\Delta \vdash P \Downarrow^e V : V'}{dec_\Delta \vdash \mathsf{tl}(P) \Downarrow^e V'} \Downarrow^e\text{TL}$$

$$\frac{dec_\Delta \vdash P_1 \Downarrow^e V \quad dec_\Delta \vdash P_2 \Downarrow^e V'}{dec_\Delta \vdash P_1 : P_2 \Downarrow^e V : V'} \Downarrow^e\text{CONS}$$

$$\frac{dec_\Delta \vdash P \Downarrow^e \mathsf{nil}_\sigma}{dec_\Delta \vdash \mathsf{elist}(P) \Downarrow^e \underline{T}} \Downarrow^e\text{ELIST}_1 \qquad \frac{dec_\Delta \vdash P \Downarrow^e V : V'}{dec_\Delta \vdash \mathsf{elist}(P) \Downarrow^e \underline{F}} \Downarrow^e\text{ELIST}_2$$

Table 6.2: Evaluation Relation $dec_\Delta \vdash P \Downarrow^e V$ in $\mathbb{FUN}^e$

(1) With F as in the last remark, the expressions $\mathsf{F}\,\underline{2}$ and $\mathsf{F}\,\underline{2}\,\underline{3}$ are (programs and) values. $\mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1})$ is a program, but not a value: the function F has numerical arity three, and can now be evaluated.

(2) Let dec_Δ be that of Examples 6.3.3 part (2). Then we can prove that

$$dec_\Delta \vdash \mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1}) \Downarrow^e \underline{10}$$

as follows:

$$\frac{\dfrac{}{\mathsf{F}\,\underline{2}\,\underline{3} \Downarrow^e \mathsf{F}\,\underline{2}\,\underline{3}} \qquad \dfrac{\dfrac{}{\underline{4} \Downarrow^e \underline{4}} \qquad \dfrac{}{\underline{1} \Downarrow^e \underline{1}}}{\underline{4}+\underline{1} \Downarrow^e \underline{5}} \qquad T}{\mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1}) \Downarrow^e \underline{10}} \Downarrow^e \mathrm{AP}$$

where T is the tree

$$\frac{\begin{array}{c}\dfrac{\dfrac{\dfrac{}{\underline{2} \Downarrow^e \underline{2}} \qquad \dfrac{}{\underline{3} \Downarrow^e \underline{3}}}{\underline{2}+\underline{3} \Downarrow^e \underline{5}} \qquad \dfrac{}{\underline{5} \Downarrow^e \underline{5}}}{\underline{2}+\underline{3}+\underline{5} \Downarrow^e \underline{10}} \\ \hline\hline (x+y+z)[\underline{2},\underline{3},\underline{5}/x,y,z] \Downarrow^e \underline{10}\end{array}}{\mathsf{F}\,\underline{2}\,\underline{3}\,\underline{5} \Downarrow^e \underline{10}} \Downarrow^e \mathrm{FAP}$$

Note that we have omitted all occurrences of $dec_\Delta \vdash$ in the tree above to save space—please feel free to do the same when you write down such deductions of evaluations. It is an exercise to fill in the missing labels on the rules.

(3) Consider the program $\mathsf{large}\, x = \underline{1} + \mathsf{large}\, x \quad in \;\; \mathsf{fst}((\underline{3}, \mathsf{large}\,\underline{0}))$. If we attempt to prove that this program evaluates to a value V, the deduction tree takes the form:

$$\frac{\dfrac{\dfrac{}{\underline{3} \Downarrow^e V} \qquad \dfrac{\dfrac{\vdots}{\underline{1} + \mathsf{large}\,\underline{0} \Downarrow^e V'}}{\mathsf{large}\,\underline{0} \Downarrow^e V'}}{(\underline{3}, \mathsf{large}\,\underline{0}) \Downarrow^e (V, V')}}{\mathsf{fst}((\underline{3}, \mathsf{large}\,\underline{0})) \Downarrow^e V}$$

for some value V'. It is easy to see that no finite deduction exists, and so there is no value V for which $dec_\Delta \vdash \mathsf{fst}((\underline{3}, \mathsf{large}\,\underline{0})) \Downarrow^e V$. Informally, we cannot take the first component of the pair without first evaluating its sub-expressions, as $\mathbb{FUN}^e$ is eager. Compare this evaluation to the execution of the same program in the lazy $\mathbb{FUN}^l$ on page 73.

Theorem 6.4.6 Let dec_Δ be a function declaration. The evaluation relation for $\mathbb{FUN}^e$ is **deterministic** in the sense that if a program evaluates to a value, that value is unique. More precisely, for all P, V_1 and V_2, if

$$dec_\Delta \vdash P \Downarrow^e V_1 \text{ and } dec_\Delta \vdash P \Downarrow^e V_2$$

then $V_1 = V_2$.

Proof We prove by Rule Induction that

$$\text{for all } dec_\Delta \vdash P \Downarrow^e V_1, \quad \text{for all } V_2, \quad (dec_\Delta \vdash P \Downarrow^e V_2 \Longrightarrow V_1 = V_2)$$

The routine details are omitted. □

Theorem 6.4.7 Evaluating a program dec_Δ *in* P does not alter its type. More precisely,

$$(\Delta \mid \varnothing \vdash P :: \sigma \text{ and } dec_\Delta \vdash P \Downarrow^e V) \quad \Longrightarrow \quad \Delta \mid \varnothing \vdash V :: \sigma$$

for any P, V, σ and Δ. The conservation of type during program evaluation is called **subject reduction**.

Proof We prove by Rule Induction that

$$\text{for all } dec_\Delta \vdash P \Downarrow^e V \quad \forall \sigma(\Delta \mid \varnothing \vdash P :: \sigma \quad \Longrightarrow \quad \Delta \mid \varnothing \vdash V :: \sigma).$$

The proof is omitted □

6.5 The Language $\mathbb{FUN}^l$

Motivation 6.5.1 The language $\mathbb{FUN}^l$ is identical to $\mathbb{FUN}^e$, except that it has a *lazy* operational semantics. We explain below exactly what this means. The expressions, contexts, function environments, function declarations and type assignments of $\mathbb{FUN}^l$ are exactly the same as for $\mathbb{FUN}^e$. In a nutshell, we can say that the definition of the relationships $\Delta \mid \Gamma \vdash E :: \sigma$ for $\mathbb{FUN}^l$ is specified by the rules in Table 6.1.

6.6 Operational Semantics for $\mathbb{FUN}^l$

Motivation 6.6.1 The operational semantics of $\mathbb{FUN}^l$ is **lazy** or **call-by-name**. This means that certain expressions can be evaluated *before* their subexpressions are computed. This method of computation applies to functions, pairs and lists. This has the advantage that if any of the subexpressions are not required in the computation of the expression, then no time is lost evaluating the subexpression. *Lazy* refers to the fact that the language does not bother to compute subexpressions if it does not need to. The definition of *program* is the same as before. We shall need a different notion of value in $\mathbb{FUN}^l$. We give the new definition of value, and then give the lazy operational semantics of $\mathbb{FUN}^l$.

Definitions 6.6.2 Let dec_Δ be a function declaration. A **value expression** is any expression V which can be produced by the following grammar

$$V ::= \underline{c} \mid \mathsf{nil}_\sigma \mid (P, P') \mid \mathsf{F}_j\, \vec{P} \mid P : P'$$

where c is any Boolean or integer, σ is any type, P and P' are any program expressions, and $\vec{P}$ is an application of the form $P P_1\, P_2\, \ldots\, P_{l-1}\, P_l$ where $0 \leq l < k_j$, and k_j is the numerical arity of F_j. *Note* that l is *strictly* less than k_j.

A **value** is any value expression for which dec_Δ *in* V is a valid $\mathbb{FUN}^l$ program.

Definitions 6.6.3 We shall define an **evaluation relation** whose judgements will take the form

$$dec_\Delta \vdash P \Downarrow^l V$$

where P and V are respectively a program expression and value expression whose function identifiers appear in the function declaration dec_Δ. The rules for inductively generating these judgements are given by the rules in Table 6.3.

Examples 6.6.4

(1) Recall Examples 6.4.5. Consider the program $\mathsf{large}\, x = \underline{1} + \mathsf{large}\, x \quad in \ \mathsf{fst}((\underline{3}, \mathsf{large}\, \underline{0}))$ once again. Let us try to see if this program evaluates to a value, say V. Working the rules in Table 6.3 backwards, there must be P_1 and P_2 and rules R and R' such that we have

$$\dfrac{\dfrac{}{(\underline{3}, \mathsf{large}\, \underline{0}) \Downarrow^l (P_1, P_2)}\,R \qquad \dfrac{}{P_1 \Downarrow^l V}\,R'}{\mathsf{fst}((\underline{3}, \mathsf{large}\, \underline{0})) \Downarrow^l V} \Downarrow^l \text{FST}$$

and clearly we have a valid (finite) deduction tree when P_1 is $\underline{3}$, P_2 is $\mathsf{large}\, \underline{0}$, V is $\underline{3}$ and R and R' are both instances of $\Downarrow^l$ VAL. In the lazy language, we can extract the first component of a pair without having first to compute the second component.

(2) Let Δ be $\mathsf{F} :: \mathsf{int} \to [\mathsf{int}]$, and dec_Δ be $\mathsf{F}x = x : \mathsf{F}\,(x + \underline{2})$. Then there is a program $dec_\Delta \quad in \ \mathsf{hd}(\mathsf{tl}(\mathsf{F}\,\underline{1}))$. We prove that $dec_\Delta \vdash \mathsf{hd}(\mathsf{tl}(\mathsf{F}\,\underline{1})) \Downarrow^l \underline{3}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{1} : \mathsf{F}\,(\underline{1}+\underline{2}) \Downarrow^l \underline{1} : \mathsf{F}\,(\underline{1}+\underline{2})}}{\mathsf{F}\,\underline{1} \Downarrow^l \underline{1} : \mathsf{F}\,(\underline{1}+\underline{2})} \Downarrow^l \text{FAP} \qquad T}{\mathsf{tl}(\mathsf{F}\,\underline{1}) \Downarrow^l (\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2})} \qquad \dfrac{\dfrac{}{\underline{1} \Downarrow^l \underline{1}} \qquad \dfrac{}{\underline{2} \Downarrow^l \underline{2}}}{\underline{1}+\underline{2} \Downarrow^l \underline{3}}}{\mathsf{hd}(\mathsf{tl}(\mathsf{F}\,\underline{1})) \Downarrow^l \underline{3}}$$

where T is the tree

$$\dfrac{\dfrac{}{(\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2}) \Downarrow^l (\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2})} \Downarrow^l \text{VAL}}{\mathsf{F}\,(\underline{1}+\underline{2}) \Downarrow^l (\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2})}$$

It is an exercise to check this deduction tree is correct, adding in the labels for the rules. Why might we call $\mathsf{F}\,\underline{1}$ the *lazy list of odd numbers*? Try evaluating $\mathsf{F}\,\underline{1}$ using the eager semantics. What happens?

Theorem 6.6.5 The language $\mathbb{FUN}^l$ is monomorphic, deterministic, and satisfies subject reduction.

Proof The proofs of these facts are basically the same as for $\mathbb{FUN}^e$ and are omitted. □

$$\frac{}{dec_\Delta \vdash V \Downarrow^l V} \Downarrow^l\text{VAL} \qquad \frac{dec_\Delta \vdash P_1 \Downarrow^l \underline{m} \quad dec_\Delta \vdash P_2 \Downarrow^l \underline{n}}{dec_\Delta \vdash P_1 \; op \; P_2 \Downarrow^l \underline{m \; op \; n}} \Downarrow^l\text{OP}$$

$$\frac{dec_\Delta \vdash P_1 \Downarrow^l \underline{T} \quad dec_\Delta \vdash P_2 \Downarrow^l V}{dec_\Delta \vdash \mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^l V} \Downarrow^l\text{COND}_1 \qquad \frac{dec_\Delta \vdash P_1 \Downarrow^l \underline{F} \quad dec_\Delta \vdash P_3 \Downarrow^l V}{dec_\Delta \vdash \mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^l V} \Downarrow^l\text{COND}_2$$

$$\frac{dec_\Delta \vdash P \Downarrow^l (P_1, P_2) \quad dec_\Delta \vdash P_1 \Downarrow^l V}{dec_\Delta \vdash \mathsf{fst}(P) \Downarrow^l V} \Downarrow^l\text{FST}$$

$$\frac{dec_\Delta \vdash P \Downarrow^l (P_1, P_2) \quad dec_\Delta \vdash P_2 \Downarrow^l V}{dec_\Delta \vdash \mathsf{snd}(P) \Downarrow^l V} \Downarrow^l\text{SND}$$

$$\frac{\begin{cases} dec_\Delta \vdash P_1 \Downarrow^l \mathsf{F}\,\vec{P} \quad dec_\Delta \vdash \mathsf{F}\,\vec{P}\,P_2 \Downarrow^l V \\ \text{where either } P_1 \text{ or } P_2 \text{ is not a value} \end{cases}}{dec_\Delta \vdash P_1\,P_2 \Downarrow^l V} \Downarrow^l\text{AP}$$

$$\frac{dec_\Delta \vdash E_{\mathsf{F}_j}[P_1, \ldots, P_{k_j}/x_1, \ldots, x_{k_j}] \Downarrow^l V}{dec_\Delta \vdash \mathsf{F}_j P_1 \ldots P_{k_j} \Downarrow^l V} \left[\mathsf{F}_j \vec{x} = E_{\mathsf{F}_j} \text{ declared in } dec_\Delta\right] \Downarrow^l\text{FAP}$$

$$\frac{dec_\Delta \vdash P \Downarrow^l \mathsf{nil}_\sigma}{dec_\Delta \vdash \mathsf{tl}(P) \Downarrow^l \mathsf{nil}_\sigma} \Downarrow^l\text{NIL}$$

$$\frac{dec_\Delta \vdash P_1 \Downarrow^l P_2 : P_3 \quad dec_\Delta \vdash P_2 \Downarrow^l V}{dec_\Delta \vdash \mathsf{hd}(P_1) \Downarrow^l V} \Downarrow^l\text{HD}$$

$$\frac{dec_\Delta \vdash P_1 \Downarrow^l P_2 : P_3 \quad dec_\Delta \vdash P_3 \Downarrow^l V}{dec_\Delta \vdash \mathsf{tl}(P_1) \Downarrow^l V} \Downarrow^l\text{TL}$$

$$\frac{dec_\Delta \vdash P \Downarrow^l \mathsf{nil}_\sigma}{dec_\Delta \vdash \mathsf{elist}(P) \Downarrow^l \underline{T}} \Downarrow^l\text{ELIST}_1 \qquad \frac{dec_\Delta \vdash P_1 \Downarrow^l P_2 : P_3}{dec_\Delta \vdash \mathsf{elist}(P_1) \Downarrow^l \underline{F}} \Downarrow^l\text{ELIST}_2$$

Table 6.3: Evaluation Relation $dec_\Delta \vdash P \Downarrow^l V$ in $\mathbb{FUN}^l$

6.7 The Language $\mathbb{TUR}^e$ and its Operational Semantics

Motivation 6.7.1 The language $\mathbb{TUR}^e$ is basically a simplified version of $\mathbb{FUN}^e$. We introduce it because giving a denotational semantics to $\mathbb{FUN}^e$ is rather too tricky for an undergraduate course. We shall study a denotational semantics of $\mathbb{TUR}^e$ in the next chapter.

Definitions 6.7.2 The types of the language $\mathbb{TUR}^e$ are given by the grammar

$$\sigma \quad ::= \quad \mathsf{int} \mid \mathsf{bool}.$$

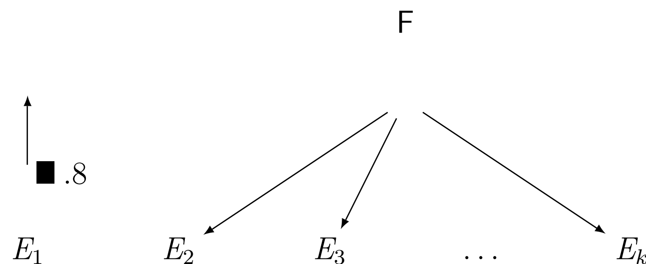
(So all expressions in $\mathbb{TUR}^e$ will be either integers or Booleans).

Let *Var* be a fixed set of **variables**. The **expressions** of the functional language $\mathbb{TUR}^e$ are given by the grammar

$$\begin{array}{lcll}
E & ::= & x & \text{variables} \\
 & \mid & \underline{c} & \text{constant} \\
 & \mid & E\ op\ E' & \text{operator} \\
 & \mid & \mathsf{if}\ E\ \mathsf{then}\ E'\ \mathsf{else}\ E'' & \text{conditional} \\
 & \mid & \mathsf{F}(E_1, \ldots, E_k) & k\text{-ary functions}
\end{array}$$

where k ranges over the non-zero natural numbers.

Remark 6.7.3 In this language, an expression such as $\mathsf{F}(E_1, E_2, E_3)$ is not "an application of F to a triple". The language $\mathbb{TUR}^e$ does not have application expressions, or pairing, or indeed tuples of any kind. All it has is expressions of the form $\mathsf{F}(E_1, \ldots, E_k)$ which are intended to be the result of a function identifier F acting on k arguments. Such an expression denotes a tree of the form



We also intend such function identifiers to act only on integer expressions and to only return integer expressions. The reason for this rather gross simplification is simply to make the presentation of material in the next chapter as simple as possible, while still retaining the essential ideas.

Definitions 6.7.4 A **function arity** is now a natural number $k \in \mathbb{N}$. This reflects the intended meaning of function identifiers—we only need to specify the number of their arguments, and not the types (which are assumed to be of type int).

$$\frac{}{\Delta \mid \Gamma \vdash x :: \mathsf{int}} \text{(where } x :: \mathsf{int} \in \Gamma) \quad \vdash \text{VAR} \qquad \frac{}{\Delta \mid \Gamma \vdash \underline{n} :: \mathsf{int}} \vdash \text{INT}$$

$$\frac{}{\Delta \mid \Gamma \vdash \underline{T} :: \mathsf{bool}} \vdash \text{TRUE} \qquad \frac{}{\Delta \mid \Gamma \vdash \underline{F} :: \mathsf{bool}} \vdash \text{FALSE}$$

$$\frac{\Delta \mid \Gamma \vdash E_1 :: \mathsf{int} \quad \Delta \mid \Gamma \vdash E_2 :: \mathsf{int}}{\Delta \mid \Gamma \vdash E_1 \ op \ E_2 :: \mathsf{int}} \text{(where } op \text{ is integer valued)} \quad \vdash \text{OP}_1$$

$$\frac{\Delta \mid \Gamma \vdash E_1 :: \mathsf{int} \quad \Delta \mid \Gamma \vdash E_2 :: \mathsf{int}}{\Delta \mid \Gamma \vdash E_1 \ op \ E_2 :: \mathsf{bool}} \text{(where } op \text{ is Boolean valued)} \quad \vdash \text{OP}_2$$

$$\frac{\Delta \mid \Gamma \vdash E_1 :: \mathsf{bool} \quad \Delta \mid \Gamma \vdash E_2 :: \sigma \quad \Delta \mid \Gamma \vdash E_3 :: \sigma}{\Delta \mid \Gamma \vdash \mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3 :: \sigma} \vdash \text{COND}$$

$$\frac{\Delta \mid \Gamma \vdash E_1 :: \mathsf{int} \quad \dots \quad \Delta \mid \Gamma \vdash E_k :: \mathsf{int}}{\Delta, \mathsf{F} :: k \mid \Gamma \vdash \mathsf{F}(E_1, \dots, E_k) :: \mathsf{int}} \vdash \text{FAP}$$

Table 6.4: Type Assignment Relation $\Delta \mid \Gamma \vdash E :: \sigma$ in $\mathbb{TUR}^e$

A **function environment** is specified by a finite set of (function identifier, function arity) pairs, with a typical function environment being denoted by

$$\Delta = \mathsf{F}_1 :: k_1, \dots, \mathsf{F}_m :: k_m.$$

We shall inductively define a type assignment (ternary) relation which takes the form $\Delta \mid \Gamma \vdash E :: \sigma$ using the rules in Table 6.4.

Definitions 6.7.5 A **function declaration** dec_Δ, where $\Delta = \mathsf{F}_1 :: k_1, \dots, \mathsf{F}_m :: k_m$ is a given, fixed, function environment consists of the following data:

$$\begin{array}{lcllr}
\mathsf{F}_1(x_{11}, \dots, x_{1k_1}) & = & E_{\mathsf{F}_1} & \text{where} & \Delta \mid x_{11} :: \mathsf{int}, \dots, x_{1k_1} :: \mathsf{int} \vdash E_{\mathsf{F}_1} :: \mathsf{int} \\
\mathsf{F}_2(x_{21}, \dots, x_{2k_2}) & = & E_{\mathsf{F}_2} & \text{where} & \Delta \mid x_{21} :: \mathsf{int}, \dots, x_{2k_2} :: \mathsf{int} \vdash E_{\mathsf{F}_2} :: \mathsf{int} \\
 & \vdots & & & \\
\mathsf{F}_j(x_{j1}, \dots, x_{jk_j}) & = & E_{\mathsf{F}_j} & \text{where} & \Delta \mid x_{j1} :: \mathsf{int}, \dots, x_{jk_j} :: \mathsf{int} \vdash E_{\mathsf{F}_j} :: \mathsf{int} \\
 & \vdots & & & \\
\mathsf{F}_m(x_{m1}, \dots, x_{mk_m}) & = & E_{\mathsf{F}_m} & \text{where} & \Delta \mid x_{m1} :: \mathsf{int}, \dots, x_{mk_m} :: \mathsf{int} \vdash E_{\mathsf{F}_m} :: \mathsf{int}
\end{array}$$

We define a **program expression** P in $\mathbb{TUR}^e$ to be any expression for which $fvar(P) = \varnothing$, that is, no variables occur in P. A **program** in $\mathbb{TUR}^e$ is a judgement of the form

$$dec_\Delta \quad in \ \ P$$

$$\frac{}{dec_\Delta \vdash \underline{c} \Downarrow^e \underline{c}} \Downarrow^e\text{VAL} \qquad \frac{dec_\Delta \vdash P_1 \Downarrow^e \underline{m} \quad dec_\Delta \vdash P_2 \Downarrow^e \underline{n}}{dec_\Delta \vdash P_1 \; op \; P_2 \Downarrow^e \underline{m \; op \; n}} \Downarrow^e\text{OP}$$

$$\frac{dec_\Delta \vdash P_1 \Downarrow^e \underline{T} \quad dec_\Delta \vdash P_2 \Downarrow^e \underline{c}}{dec_\Delta \vdash \mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e \underline{c}} \Downarrow^e\text{COND}_1 \qquad \frac{dec_\Delta \vdash P_1 \Downarrow^e \underline{F} \quad dec_\Delta \vdash P_3 \Downarrow^e \underline{c}}{dec_\Delta \vdash \mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e \underline{c}} \Downarrow^e\text{COND}_2$$

$$\frac{\begin{cases} dec_\Delta \vdash P_1 \Downarrow^e \underline{n_1} \quad \ldots \quad dec_\Delta \vdash P_{k_j} \Downarrow^e \underline{n_{k_j}} \\ dec_\Delta \vdash E_{\mathsf{F}_j}[\underline{n_1}, \ldots, \underline{n_{k_j}}/x_1, \ldots, x_{k_j}] \Downarrow^e \underline{n} \end{cases}}{dec_\Delta \vdash \mathsf{F}_j(P_1, \ldots, P_{k_j}) \Downarrow^e \underline{n}} \; [\mathsf{F}_j(\vec{x}) = E_{\mathsf{F}_j} \text{ declared in } dec_\Delta] \; \Downarrow^e\text{FAP}$$

Table 6.5: Evaluation Relation $dec_\Delta \vdash P \Downarrow^e \underline{c}$ in $\mathbb{TUR}^e$

where dec_Δ is a given function declaration and the program expression P satisfies a type assignment of the form

$$\Delta \mid \varnothing \vdash P :: \sigma$$

where σ is of course either int or bool. We may sometimes simply refer to P as a program, when no confusion can arise from this. We call σ the type of the program dec_Δ *in* P (and sometimes just say σ is the type of P).

A **value** is any $\underline{c}$ where c is any Boolean or integer.

Definitions 6.7.6 We shall define an **evaluation relation** whose judgements will take the form

$$dec_\Delta \vdash P \Downarrow^e \underline{c}$$

where P and $\underline{c}$ are respectively a program expression and value. The rules for inductively generating these judgements are given by the rules in Table 6.5. Note that the operational semantics is eager.

Theorem 6.7.7 The language $\mathbb{TUR}^e$ is monomorphic, deterministic with respect to the evaluation relation, and subject reduction holds.

Proof The proof method is identical to that for $\mathbb{FUN}^e$, and indeed the details are pretty much the same. It is an exercise to provide some of the details of the appropriate Rule Inductions. □

6.8 The Language $\mathbb{TUR}^l$ and its Operational Semantics

Definitions 6.8.1 The expressions, contexts, function environments, function declarations and type assignments of $\mathbb{TUR}^l$ are exactly the same as for $\mathbb{TUR}^e$. In a nutshell, we

$$\frac{}{dec_\Delta \vdash \underline{c} \Downarrow^l \underline{c}} \Downarrow^l \text{VAL} \qquad \frac{dec_\Delta \vdash P_1 \Downarrow^l \underline{m} \quad dec_\Delta \vdash P_2 \Downarrow^l \underline{n}}{dec_\Delta \vdash P_1 \; op \; P_2 \Downarrow^l \underline{m \; op \; n}} \Downarrow^l \text{OP}$$

$$\frac{dec_\Delta \vdash P_1 \Downarrow^l \underline{T} \quad dec_\Delta \vdash P_2 \Downarrow^l \underline{c}}{dec_\Delta \vdash \mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^l \underline{c}} \Downarrow^l \text{COND}_1 \qquad \frac{dec_\Delta \vdash P_1 \Downarrow^l \underline{F} \quad dec_\Delta \vdash P_3 \Downarrow^l \underline{c}}{dec_\Delta \vdash \mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^l \underline{c}} \Downarrow^l \text{COND}_2$$

$$\frac{dec_\Delta \vdash E_{\mathsf{F}_j}[\underline{P_1}, \dots, \underline{P_{k_j}}/x_1, \dots, x_{k_j}] \Downarrow^l \underline{n}}{dec_\Delta \vdash \mathsf{F}_j(P_1, \dots, P_{k_j}) \Downarrow^l \underline{n}} [\mathsf{F}_j(\vec{x}) = E_{\mathsf{F}_j} \text{ declared in } dec_\Delta] \Downarrow^l \text{FAP}$$

Table 6.6: Evaluation Relation $dec_\Delta \vdash P \Downarrow^l \underline{c}$ in $\mathbb{TUR}^l$

can say that the definition of the relationships $\Delta \mid \Gamma \vdash E :: \sigma$ for $\mathbb{TUR}^l$ are specified by the rules in Table 6.4.

The operational semantics is (slightly) different. We shall define an evaluation relation, whose relationships will be written $dec_\Delta \vdash P \Downarrow^l V$, via the rules given in Table 6.6. Note that this is a lazy language.

Theorem 6.8.2 The language $\mathbb{TUR}^l$ is monomorphic, deterministic with respect to the evaluation relation, and subject reduction holds.

Proof The proof method is as usual. □

Examples 6.8.3 Show that the program

$$\left.\begin{array}{lcl} \mathsf{large}(x) & = & \underline{1} + \mathsf{large}(x) \\ \mathsf{const}(x) & = & \underline{4} \end{array}\right\} \quad in \;\; \mathsf{const}(\mathsf{large}(\underline{0}))$$

evaluates to $\underline{4}$ in $\mathbb{TUR}^l$, but does not evaluate to a value in $\mathbb{TUR}^e$.

In $\mathbb{TUR}^l$ we have

$$\cfrac{\cfrac{\cfrac{}{\underline{4} \Downarrow^l \underline{4}} \Downarrow^l \text{VAL}}{\underline{4}[\mathsf{large}(\underline{0})/x] \Downarrow^l \underline{4}}}{\mathsf{const}(\mathsf{large}(\underline{0})) \Downarrow^l \underline{4}} \Downarrow^l \text{FAP}$$

The case of $\mathbb{TUR}^e$ is an exercise.

7
The Denotational Semantics of Functional Languages

7.1 Introduction

Motivation 7.1.1 We shall now present a denotational semantics for $\mathbb{TUR}^e$. The basic idea behind this kind of semantics is just the same as for $\mathbb{IMP}$. Such a semantics can be thought of as a mathematical model, in which we attempt to abstract away from the operational details of the programming language, seeing the crucial concepts of the language appearing in a very general setting. We shall see that the method of modelling the recursion present in the operational semantics of a while *be* do *co* loop in $\mathbb{IMP}$ is just the same as the method of modelling recursive function declarations in $\mathbb{TUR}^e$—by fixpoints of continuous functions. Thus we have a framework in which two apparently different languages, with quite distinct operational semantics, can in fact be seen to be quite similar.

7.2 Denotations for Type Assignments in $\mathbb{TUR}^e$

Motivation 7.2.1 Let us think about how we might model the type assignment

$$\mathsf{F} :: 1 \mid x :: \mathsf{int} \vdash E :: \mathsf{int}.$$

The type assignment indicates that E is an expression containing the identifier F and the variable x. Thus E "depends" on F and x. Now, x is of type int so we could model x by specifying an integer e in $\mathbb{Z}$. Also, F is of type int $\rightarrow$ int and we could model this by giving a function between $\mathbb{Z}$ and $\mathbb{Z}$, that is a function f in $[\mathbb{Z},\mathbb{Z}]_{tot}$. We can then regard the meaning of E as a function (call it h) which acts on the meanings of F and x, namely f and e, and returns an integer (as E has type int). So we could require h to take the pair (f,e) as input and return an integer $h(f,e)$ as output. Thus

$$h : ([\mathbb{Z},\mathbb{Z}]_{tot} \times \mathbb{Z}) \longrightarrow \mathbb{Z} \qquad (*)$$

is a function which we can regard as the *meaning* of $\mathsf{F} :: 1 \mid x :: \mathsf{int} \vdash E :: \mathsf{int}$.

These ideas need refining. First, F may "represent" a recursive program which can loop. Thus we really need to model it as a partial function on integers. As we have seen, it is easier to think about functions between $\mathbb{Z}$ and $\mathbb{Z}_\perp$. Similarly, E is an expression whose operational behaviour may constitute a loop, so its meaning ought also to be a partial function. Moreover, E may contain recursively defined functions. As we have seen, we can model such recursion using fixpoints, and in order to ensure that these exist we have to deal with *continuous* functions. We can regard $\mathbb{Z}$ as a discrete cpo, and then the set

of total functions $[\mathbb{Z}, \mathbb{Z}_\perp]_{tot}$ coincides with the set of continuous functions $[\mathbb{Z}, \mathbb{Z}_\perp]_{cts}$. We shall partially order this latter set of continuous functions just as we did in on page 54, and moreover we have a cpo

$$[\mathbb{Z}, \mathbb{Z}_\perp]_{cts} \times \mathbb{Z}$$

defined using the definitions in Section 5.3. So in fact we should ask that

$$h : [\mathbb{Z}, \mathbb{Z}_\perp]_{cts} \times \mathbb{Z} \longrightarrow \mathbb{Z}_\perp$$

be a continuous function (which is a refinement of $(*)$).

If in fact $\mathsf{F} :: k$, then we can model F as a function which acts on a k-tuple of integers, and returns an integer. The meaning of F is thus a continuous function from $\mathbb{Z}^k$ to $\mathbb{Z}_\perp$, that is an element of $[\mathbb{Z}^k, \mathbb{Z}_\perp]_{cts}$. The latter is of course also a cpo—see Section 5.3.

What about modelling $\Delta = \mathsf{F}_1 :: k_1, \ldots, \mathsf{F}_m :: k_m$? It seems sensible to model this as an m-tuple of functions, say $(f_1, \ldots, f_m)$ where each f_j models F_j. Thus $f_j \in [\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$ and

$$\vec{f} = (f_1, \ldots, f_m) \in \Pi_{j=1}^{j=m} [\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$$

Similarly, we can model $\Gamma = x_1 :: \mathsf{int}, \ldots, x_n :: \mathsf{int}$ as an n-tuple of integers, say

$$\vec{e} = (e_1, \ldots, e_n) \in \mathbb{Z}^n$$

Putting this all together,

we can regard the meaning of $\Delta \mid \Gamma \vdash E :: \mathsf{int}$ as a function (say h) which takes $\vec{f}$ and $\vec{e}$ as inputs, and either returns an integer as output, or $\perp$ if "E loops." Thus

$$h : (\Pi_{j=1}^{j=m} [\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}) \times \mathbb{Z}^n \longrightarrow \mathbb{Z}_\perp.$$

Remark 7.2.2

(i) In fact $\Pi_{j=1}^{j=m} [\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$ is a cpo with bottom. It has a bottom element given by

$$(\lambda_{\vec{n} \in \mathbb{Z}^{k_1}}.\perp, \ldots, \lambda_{\vec{n} \in \mathbb{Z}^{k_m}}.\perp)$$

where $\lambda_{\vec{n} \in \mathbb{Z}^{k_j}}.\perp : \mathbb{Z}^{k_j} \to \mathbb{Z}_\perp$.

(ii) It may be helpful to spell out the definition of the partial order on $\Pi_{j=1}^{j=m} [\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$ explicitly. If $\vec{f}, \vec{f'} \in \Pi_{j=1}^{j=m} [\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$, then $\vec{f} \preceq \vec{f'}$ if and only if

$$\text{for all } j = 1, \ldots, m, \quad \text{for all } (n_1, \ldots, n_{k_j}) \in \mathbb{Z}^{k_j}, \quad f_j(n_1, \ldots, n_{k_j}) \preceq f'_j(n_1, \ldots, n_{k_j}) \text{ in } \mathbb{Z}_\perp$$

if and only if

$$\text{for all } j = 1, \ldots, m, \quad \text{for all } (n_1, \ldots, n_{k_j}) \in \mathbb{Z}^{k_j}, \quad \text{for all } n \in \mathbb{Z} \quad f_j(n_1, \ldots, n_{k_j}) = [n] \Longrightarrow f'_j(n_1, \ldots, n_{k_j}) = [n]$$

$$
\begin{array}{rcl}
[\![\Delta \mid \Gamma \vdash x_i]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & [e_i] \\
[\![\Delta \mid \Gamma \vdash \underline{c}]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & [c] \\
[\![\Delta \mid \Gamma \vdash E_1 \ op \ E_2]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & (\iota \circ op)_{\vec{\bot}} \, ([\![\Delta \mid \Gamma \vdash E_1]\!](\vec{f},\vec{e}) \, , \, [\![\Delta \mid \Gamma \vdash E_2]\!](\vec{f},\vec{e})) \\
[\![\Delta \mid \Gamma \vdash \textsf{if } E_1 \textsf{ then } E_2 \textsf{ else } E_3]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & \\
 & & cond_{\bot_{\mathbb{B}}}([\![\Delta \mid \Gamma \vdash E_1]\!](\vec{f},\vec{e}), [\![\Delta \mid \Gamma \vdash E_2]\!](\vec{f},\vec{e}), [\![\Delta \mid \Gamma \vdash E_3]\!](\vec{f},\vec{e})) \\
[\![\Delta \mid \Gamma \vdash \mathsf{F}_j(E_1,\ldots,E_{k_j})]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & (f_j)_{\vec{\bot}}([\![\Delta \mid \Gamma \vdash E_1]\!](\vec{f},\vec{e}),\ldots,[\![\Delta \mid \Gamma \vdash E_{k_j}]\!](\vec{f},\vec{e})) \\
op & : & \mathbb{Z} \times \mathbb{Z} \longrightarrow X \\
\iota \circ op & : & \mathbb{Z} \times \mathbb{Z} \longrightarrow X_\bot \\
(\iota \circ op)_{\vec{\bot}} & : & \mathbb{Z}_\bot \times \mathbb{Z}_\bot \longrightarrow X_\bot \\
cond_{\bot_{\mathbb{B}}} & : & \mathbb{B}_\bot \times X_\bot \times X_\bot \longrightarrow X_\bot \\
f_j & : & \mathbb{Z}^{k_j} \longrightarrow \mathbb{Z}_\bot \\
(f_j)_{\vec{\bot}} & : & (\mathbb{Z}_\bot)^{k_j} \longrightarrow \mathbb{Z}_\bot \\
X & = & \mathbb{Z} \text{ or } \mathbb{B}
\end{array}
$$

Table 7.1: Denotational Semantics $[\![\Delta \mid \Gamma \vdash E]\!]$ in $\mathbb{TUR}^e$

(iii) If $\Gamma = \varnothing$ we take $n = 0$ and if $\Delta = \varnothing$ we take $m = 0$.

Definitions 7.2.3 Recall that in general, if *exp* is a syntactical expression in a programming language, and we are intending to model *exp* as some mathematical entity, then we shall write $[\![exp]\!]$ for the object modelling *exp*. We can think of $[\![exp]\!]$ as a semantical explanation of *exp*, and we refer to $[\![exp]\!]$ as the **denotational semantics** of *exp*.

Thus for $\mathbb{TUR}^e$, for each type assignment $\Delta \mid \Gamma \vdash E :: \sigma$ (where σ is either int or bool) we shall define a (continuous) function of the form

$$[\![\Delta \mid \Gamma \vdash E :: \textsf{int}]\!] : (\Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts} \times \mathbb{Z}^n) \longrightarrow \mathbb{Z}_\bot$$

or

$$[\![\Delta \mid \Gamma \vdash E :: \textsf{bool}]\!] : (\Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts} \times \mathbb{Z}^n) \longrightarrow \mathbb{B}_\bot.$$

We do this recursively, using the rules in Table 7.1, where we denote a typical element of $(\Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts}) \times \mathbb{Z}^n$ by $(\vec{f},\vec{e})$. Note that because $\mathbb{TUR}^e$ enjoys subject reduction, we omit the type of E in each of the judgements defining the denotational semantics. You may like to recall the notation of Section 5.3. Note that in this chapter we sometimes use prefix notation for operators *op*. Thus, for example, $\leq(6,8) = T$.

Example 7.2.4 Set $\Delta \stackrel{\text{def}}{=} \mathsf{suc} :: 1, \mathsf{F}' :: 4$. So

$$[\![\Delta \mid x :: \mathsf{int}, y :: \mathsf{int} \vdash x \leq y]\!] : ([\mathbb{Z}, \mathbb{Z}_\perp]_{cts} \times [\mathbb{Z}^4, \mathbb{Z}_\perp]_{cts}) \times \mathbb{Z}^2 \longrightarrow \mathbb{B}_\perp$$

and we have $m = 2$, $k_1 = 1$, $k_2 = 4$ and $n = 2$. Then

$$\begin{aligned}
&[\![\Delta \mid x :: \mathsf{int}, y :: \mathsf{int} \vdash x \leq y]\!]((f, f'), (4, 3)) \\
&= (\iota\circ \leq)_{\perp}^{\to} ([\![\Delta \mid x :: \mathsf{int}, y :: \mathsf{int} \vdash x]\!]((f, f'), (4, 3)) \ , \ [\![\Delta \mid x :: \mathsf{int}, y :: \mathsf{int} \vdash y]\!]((f, f'), (4, 3))) \\
&= (\iota\circ \leq)_{\perp}^{\to} ([4] \ , \ [3]) \\
&= (\iota\circ \leq) (4 \ , \ 3) \\
&= [4 \leq 3] \\
&= [F] \in \mathbb{B}_\perp.
\end{aligned}$$

What role, if any, do the function identifiers play?

Proposition 7.2.5 Each function $[\![\Delta \mid \Gamma \vdash E]\!]$ is indeed continuous.

Proof (Please see page 60). This is proved by induction on the derivation of $\Delta \mid \Gamma \vdash E :: \sigma$. The base cases are when E is a variable or constant. In the first case, the denotational function is a projection, hence continuous, and in the second case, the denotational function is constant, hence continuous. In the inductive cases, the denotational functions are built up from continuous functions, using continuity preserving operations. □

Remark 7.2.6 Recall that programs P satisfy type assignments of the form $\Delta \mid \varnothing \vdash P :: \sigma$. Thus their denotations are functions of the form

$$[\![\Delta \mid \varnothing \vdash P]\!] : (\Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}) \times \mathbb{Z}^0 \longrightarrow X_\perp$$

where X is $\mathbb{Z}$ or $\mathbb{B}$ depending on σ. Now $\mathbb{Z}^0$ is the cpo with just one element, say $\{*\}$. Thus the previous function essentially determines a function of the form

$$g : \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts} \longrightarrow X_\perp$$

where $g(\vec{f}) \stackrel{\text{def}}{=} [\![\Delta \mid \varnothing \vdash P]\!](\vec{f}, *)$. From now on we shall abuse notation, and consider program denotations as functions of the form

$$[\![\Delta \mid \varnothing \vdash P]\!] : \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts} \longrightarrow X_\perp.$$

Similarly, if Γ has n variables, we shall take

$$[\![\varnothing \mid \Gamma \vdash E]\!] : \mathbb{Z}^n \longrightarrow X_\perp$$

with X as above.

7.3 Denotations of Function Declarations and Programs in $\mathbb{TUR}^e$

Motivation 7.3.1 We aim to give a denotational model of programs $dec_\Delta \quad in \quad P$. First we need to give a model of a function declaration dec_Δ. Roughly, the idea behind modelling dec_Δ is this: Each function in the declaration is determined by its definitional body. Further, each function may be recursive. We can give a denotation to the function bodies using Table 7.1. We can then form the tuple of functions modelling the declaration bodies. Then the denotation of the function declaration is given by the fixpoint of the tuple function.

Definitions 7.3.2 Suppose that dec_Δ is a typical function declaration for $\mathbb{TUR}^e$

$$\begin{array}{rcl} \mathsf{F}_1(x_{11},\ldots,x_{1k_1}) & = & E_{\mathsf{F}_1} \\ \mathsf{F}_2(x_{21},\ldots,x_{2k_2}) & = & E_{\mathsf{F}_2} \\ & \vdots & \\ \mathsf{F}_j(x_{j1},\ldots,x_{jk_j}) & = & E_{\mathsf{F}_j} \\ & \vdots & \\ \mathsf{F}_m(x_{m1},\ldots,x_{mk_m}) & = & E_{\mathsf{F}_m} \end{array}$$

where $\Delta = \mathsf{F}_1 :: k_1, \ldots, \mathsf{F}_m :: k_m$, and we let j run between 1 and m. We write Γ_{F_j} for $x_{j1} :: \mathsf{int}, \ldots, x_{jk_j} :: \mathsf{int}$.

By Proposition 7.2.5 we know that each definitional body E_{F_j} determines a continuous function

$$[\![\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E_{\mathsf{F}_j}]\!] : \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts} \times \mathbb{Z}^{k_j} \to \mathbb{Z}_\bot$$

and hence

$$\Phi_j \stackrel{\text{def}}{=} cur([\![\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E_{\mathsf{F}_j}]\!]) : \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts} \to [\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts}.$$

Thus there is a continuous function

$$\Phi \stackrel{\text{def}}{=} \langle \Phi_1, \ldots, \Phi_m \rangle : \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts} \longrightarrow \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts}.$$

We can apply Theorem 5.3.16 to see that Φ has a least fixpoint

$$\mathit{fix}(\Phi) \in \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts}.$$

With this, we define

$$[\![dec_\Delta]\!] \stackrel{\text{def}}{=} \mathit{fix}(\Phi) = \mathit{fix}(\langle cur([\![\Delta \mid \Gamma_{\mathsf{F}_1} \vdash E_{\mathsf{F}_1}]\!]), \ldots, cur([\![\Delta \mid \Gamma_{\mathsf{F}_m} \vdash E_{\mathsf{F}_m}]\!])\rangle).$$

Finally, we define the denotation of a program $dec_\Delta \quad in \;\; P$ to be

$$[\![dec_\Delta \quad in \;\; P]\!] \stackrel{\text{def}}{=} [\![\Delta \mid \varnothing \vdash P]\!]([\![dec_\Delta]\!]).$$

Examples 7.3.3

(1) Let Δ be $\mathsf{P} :: 2$, and let dec_Δ be $\mathsf{P}\,(x,y) = x+y$. Writing Γ for $x :: \mathsf{int}, y :: \mathsf{int}$, clearly $\Delta \mid \Gamma \vdash x+y :: \mathsf{int}$. Note that, with respect to our general notation, we have $m = 1$, $1 \le j \le m$, $k_j = k_1 = 2$, and $n = 2$. So

$$p \stackrel{\text{def}}{=} [\![\Delta \mid \Gamma \vdash x+y :: \mathsf{int}]\!] : [\mathbb{Z}^2, \mathbb{Z}_\perp]_{cts} \times \mathbb{Z}^2 \longrightarrow \mathbb{Z}_\perp.$$

If $(f,(e,e')) \in [\mathbb{Z}^2, \mathbb{Z}_\perp]_{cts} \times \mathbb{Z}^2$, then

$$\begin{aligned} p(f,(e,e')) &= (\iota \circ +)_{\bar{\perp}}\,([\![\Delta \mid \Gamma \vdash x]\!](f,(e,e'))\ ,\ [\![\Delta \mid \Gamma \vdash y]\!](f,(e,e'))) \\ &= (\iota \circ +)_{\bar{\perp}}\,([e]\ ,\ [e']) \\ &= [e+e']. \qquad\qquad (*) \end{aligned}$$

Now, $[\![dec_\Delta]\!]$ is defined to be $\mathit{fix}(\Phi)$ where

$$\Phi \stackrel{\text{def}}{=} cur(p) : [\mathbb{Z}^2, \mathbb{Z}_\perp]_{cts} \longrightarrow [\mathbb{Z}^2, \mathbb{Z}_\perp]_{cts}.$$

But for any $f \in [\mathbb{Z}^2, \mathbb{Z}_\perp]_{cts}$ we have $cur(p)(f) = \lambda_{(e,e') \in \mathbb{Z}^2}.[e+e']$ using $(*)$. Thus Φ is a constant function. Using this, it is trivial that for each $n \ge 0 \in \mathbb{N}$ we have

$$\Phi^{n+1}(\perp_{[\mathbb{Z}^2,\mathbb{Z}_\perp]_{cts}}) \stackrel{\text{def}}{=} \Phi(\Phi^n(\perp_{[\mathbb{Z}^2,\mathbb{Z}_\perp]_{cts}})) = \lambda_{(e,e') \in \mathbb{Z}^2}.[e+e']$$

and thus for all $(e,e') \in \mathbb{Z}^2$

$$[\![dec_\Delta]\!](e,e') = \bigvee_{n=0}^{\infty} \Phi^n(\perp_{[\mathbb{Z}^2,\mathbb{Z}_\perp]_{cts}}) = [e+e'].$$

Did you expect this?!? Note that the link $\Phi^0(\perp_{[\mathbb{Z}^2,\mathbb{Z}_\perp]_{cts}})$ is $\perp_{[\mathbb{Z}^2,\mathbb{Z}_\perp]_{cts}}$ and every other link in the chain $(\Phi^n(\perp_{[\mathbb{Z}^2,\mathbb{Z}_\perp]_{cts}}) \mid n \in \mathbb{N})$ is the function $\lambda_{(e,e') \in \mathbb{Z}^2}.[e+e']$.

(2) Let Δ be $\mathsf{F} :: 1$ and let dec_Δ be

$$\mathsf{F}(x) = \mathsf{if}\ x \le \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x * \mathsf{F}(x - \underline{1}).$$

Writing E_F for the body of the function declaration, and Γ for $x :: \mathsf{int}$ we have

$$h \stackrel{\text{def}}{=} [\![\Delta \mid \Gamma \vdash E_\mathsf{F}]\!] : [\mathbb{Z}, \mathbb{Z}_\perp]_{cts} \times \mathbb{Z} \longrightarrow \mathbb{Z}_\perp.$$

If $(f,e) \in [\mathbb{Z}, \mathbb{Z}_\perp]_{cts} \times \mathbb{Z}$, then

$$\begin{aligned} h(f,e) &= cond_{\perp_\mathbb{B}}([\![\Delta \mid \Gamma \vdash x \le \underline{1}]\!](f,e)\ ,\ [\![\Delta \mid \Gamma \vdash \underline{1}]\!](f,e)\ ,\ [\![\Delta \mid \Gamma \vdash x * \mathsf{F}\,(x-\underline{1})]\!](f,e)) \\ &= cond_{\perp_\mathbb{B}}((\iota \circ \le)_{\bar{\perp}}\,([\![\Delta \mid \Gamma \vdash x]\!](f,e)\ ,\ [\![\Delta \mid \Gamma \vdash \underline{1}]\!](f,e))\ ,\ [1]\ , \\ &\qquad\qquad (\iota \circ *)_{\bar{\perp}}\,([\![\Delta \mid \Gamma \vdash x]\!](f,e)\ ,\ [\![\Delta \mid \Gamma \vdash \mathsf{F}\,(x-\underline{1})]\!](f,e))) \\ &= cond_{\perp_\mathbb{B}}((\iota \circ \le)_{\bar{\perp}}\,([e]\ ,\ [1])\ ,\ [1]\ ,\ (\iota \circ *)_{\bar{\perp}}\,([e]\ ,\ f_{\bar{\perp}}([\![\Delta \mid \Gamma \vdash x - \underline{1}]\!])(f,e))) \\ &\ \vdots \\ &= cond_{\perp_\mathbb{B}}([e \le 1]\ ,\ [1]\ ,\ (\iota \circ *)_{\bar{\perp}}\,([e]\ ,\ f_{\bar{\perp}}((\iota \circ -)_{\bar{\perp}}\,([e]\ ,\ [1])))) \\ &= cond_{\perp_\mathbb{B}}([e \le 1]\ ,\ [1]\ ,\ (\iota \circ *)_{\bar{\perp}}\,([e]\ ,\ f_{\bar{\perp}}([e-1]))) \\ &= cond_{\perp_\mathbb{B}}([e \le 1]\ ,\ [1]\ ,\ (\iota \circ *)_{\bar{\perp}}\,([e]\ ,\ f(e-1))) \end{aligned}$$

Thus we have

$$h(f,e) = \begin{cases} [1] & \text{if } e \leq 1 \\ [e * e'] & \text{if } e > 1 \text{ and } f(e-1) = [e'] \text{ for some } e' \in \mathbb{Z} \\ \bot & \text{if } e > 1 \text{ and } f(e-1) = \bot \end{cases} \quad (1)$$

We now claim that for each $e \in \mathbb{Z}$, $[\![dec_\Delta]\!] : \mathbb{Z} \to \mathbb{Z}_\bot$ where

$$[\![dec_\Delta]\!](e) = \xi \stackrel{\text{def}}{=} \begin{cases} [1] & \text{if } e \leq 1 \\ [e * (e-1) * \ldots * 1] & \text{if } e > 1 \end{cases}$$

Recall that $[\![dec_\Delta]\!] \stackrel{\text{def}}{=} \mathit{fix}(\Phi)$ where $\Phi = \mathit{cur}(h)$. Now if we write $\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}}$ for the bottom element of $[\mathbb{Z},\mathbb{Z}_\bot]_{cts}$, we have $[\![dec_\Delta]\!] = \bigvee_{n=1}^{\infty} \Phi^n(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})$ and so we will prove that for any $e \in \mathbb{Z}$,

$$\xi = \bigvee_{n=1}^{\infty} \Phi^n(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e) \quad (2)$$

We claim that for each $n \geq 1$,

$$\Phi^n(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e) = \begin{cases} [1] & \text{if } e \leq 1 \\ [e * (e-1) * (e-2) * \ldots * 1] & \text{if } 1 < e \leq n \\ \bot & \text{if } n < e \end{cases} \quad \mathit{Prop}(n)$$

If $\mathit{Prop}(n)$ holds for all n, then it is an exercise to verify that (2) follows. So we shall now prove by induction that for all $n \geq 1$, $\mathit{Prop}(n)$. First note that $\mathit{Prop}(1)$ holds, for $\Phi(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e) = \mathit{cur}(h)(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e) = h(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}}, e)$ and thus using (1) we have

$$\Phi(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e) = \begin{cases} [1] & \text{if } e \leq 1 \\ \bot & \text{if } e > 1 \end{cases}$$

Now suppose inductively that $\mathit{Prop}(n)$ holds for an arbitrary n. As $\Phi^{n+1} \stackrel{\text{def}}{=} \Phi \circ \Phi^n$ we have $\Phi^{n+1}(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e) = \Phi(\Phi^n(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}}))(e) = h(\Phi^n(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}}), e)$, and so

$$\Phi^{n+1}(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e) = \begin{cases} [1] & \text{if } e \leq 1 \\ [e * e'] & \text{if } e > 1 \text{ and } \Phi^n(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e-1) = [e'] \text{ some } e' \in \mathbb{Z} \\ \bot & \text{if } e > 1 \text{ and } \Phi^n(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e-1) = \bot \end{cases}$$

But whenever $e > 1$, then[1] by $\mathit{Prop}(n)$

$$\Phi^n(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e-1) = [e'] \text{ some } e' \in \mathbb{Z} \iff 1 < e-1 \leq n \text{ or } e = 2$$

(and $e' = (e-1) * (e-2) * \ldots * 1$) and

$$\Phi^n(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e-1) = \bot \iff n < e-1$$

[1] from where does $e = 2$ arise?

Thus

$$\Phi^{n+1}(\bot_{[\mathbb{Z},\mathbb{Z}_\bot]_{cts}})(e) = \begin{cases} [1] & \text{if } e \leq 1 \\ [e*(e-1)*(e-2)*\ldots*1] & \text{if } 1 < e \leq n+1 \\ \bot & \text{if } n+1 < e \end{cases}$$

Thus $Prop(n+1)$ holds and the induction is complete.

Let us finish by computing the denotation of $dec_\Delta \quad in \;\; \mathsf{F}(\underline{3})$. We have

$$\begin{aligned} [\![dec_\Delta \quad in \;\; \mathsf{F}(\underline{3})]\!] &= [\![\Delta \mid \varnothing \vdash \mathsf{F}(\underline{3})]\!]([\![dec_\Delta]\!]) \\ &= [\![dec_\Delta]\!]_\bot([\![\Delta \mid \varnothing \vdash \underline{3}]\!]([\![dec_\Delta]\!])) \\ &= [\![dec_\Delta]\!]_\bot([3]) \\ &= [\![dec_\Delta]\!](3) \\ &= [3*2*1] \\ &= [6] \end{aligned}$$

Did you expect this ?!?

7.4 Equivalence of Operational and Denotational Semantics

Motivation 7.4.1 We would like to prove a correspondence between the operational and denotational semantics of $\mathbb{TUR}^e$, which mimics correspondences such as

$$\mathcal{I}(ie)(s) = n \qquad \Longleftrightarrow \qquad (ie\,,\, s) \Downarrow_{IExp} \underline{n} \qquad \Longleftrightarrow \qquad [\![ie]\!](s) = n.$$

To this end, we set up some suitable notation, and then prove a correspondence theorem, the latter making use of the next lemma.

Definitions 7.4.2 Let us define $\mathcal{E}(P)(dec_\Delta)$ by

$$\mathcal{E}(P)(dec_\Delta) \stackrel{\text{def}}{=} \begin{cases} [c] & \text{if } dec_\Delta \vdash P \Downarrow^e \underline{c} \text{ for some } c \in \mathbb{Z} \cup \mathbb{B} \\ \bot & \text{otherwise} \end{cases}$$

This gives us some notation for program evaluation. What about function declarations? In view of Theorem 6.4.6, we can see that for any function declaration dec_Δ and F_j appearing in Δ, then

$$\{\ ((n_1 \ldots, n_{k_j}), n) \ \mid \ dec_\Delta \vdash \mathsf{F}_j(\underline{n_1}, \ldots, \underline{n_{k_j}}) \Downarrow^e \underline{n}\ \}$$

defines a partial function in $[\mathbb{Z}^{k_j}, \mathbb{Z}]_{par}$ which we denote by $\mathcal{D}(\mathsf{F}_j)(dec_\Delta)$. Equivalently, we have defined a function $\mathcal{D}(\mathsf{F}_j)(dec_\Delta) : \mathbb{Z}^{k_j} \to \mathbb{Z}_\bot$ for each $\mathsf{F}_j \in \Delta$ given by

$$\mathcal{D}(\mathsf{F}_j)(dec_\Delta)(n_1 \ldots, n_{k_j}) = \begin{cases} [n] \text{ if } dec_\Delta \vdash \mathsf{F}_j(\underline{n_1}, \ldots, \underline{n_{k_j}}) \Downarrow^e \underline{n} \text{ for some } n \in \mathbb{Z} \\ \bot \text{ otherwise} \end{cases}$$

Lemma 7.4.3 Suppose we have a type assignment of the form $\Delta \mid \Gamma, \Gamma' \vdash E :: \mathsf{int}$, where $\Gamma' = y_{n+1} :: \mathsf{int}, \ldots, y_{n'} :: \mathsf{int}$. Then for any $(e_1, \ldots, e_n, e'_{n+1}, \ldots, e'_{n'}) \in \mathbb{Z}^{n+n'}$ and $\vec{f} \in \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$ we have

$$[\![\Delta \mid \Gamma \vdash E[\underline{e'_{n+1}}, \ldots, \underline{e'_{n'}}/y_{n+1}, \ldots, y_{n'}]]\!](\vec{f}, (e_1, \ldots, e_n)) = [\![\Delta \mid \Gamma, \Gamma' \vdash E]\!](\vec{f}, (e_1, \ldots, e_n, e'_{n+1}, \ldots, e'_{n'})).$$

Proof We use Rule Induction on the expression E. The details are omitted. □

Theorem 7.4.4 Let dec_Δ be a given function declaration with $\Delta = \mathsf{F}_1 :: k_1, \ldots, \mathsf{F}_m :: k_m$. We define

$$\mathcal{D}_{dec_\Delta} \stackrel{\text{def}}{=} (\mathcal{D}(\mathsf{F}_1)(dec_\Delta), \ldots, \mathcal{D}(\mathsf{F}_m)(dec_\Delta))$$

and so (by definition) $(\mathcal{D}_{dec_\Delta})_j = \mathcal{D}(\mathsf{F}_j)(dec_\Delta)$.

(i) We have

$$[\![dec_\Delta]\!] = \mathcal{D}_{dec_\Delta} \in \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$$

that is,

$$[\![dec_\Delta]\!]_j(n_1 \ldots, n_{k_j}) = [n] \iff \mathcal{D}(\mathsf{F}_j)(dec_\Delta)(n_1 \ldots, n_{k_j}) = [n].$$

(ii) We have

$$[\![dec_\Delta \quad in \quad P]\!] = \mathcal{E}(P)(dec_\Delta)$$

that is

$$[\![\Delta \mid \varnothing \vdash P]\!]([\![dec_\Delta]\!]) = [c] \iff \mathcal{E}(P)(dec_\Delta) = [c].$$

Proof We shall prove the following statements:

(a) For all P and c,

$$\mathcal{E}(P)(dec_\Delta) = [c] \implies [\![\Delta \mid \varnothing \vdash P]\!]([\![dec_\Delta]\!]) = [c].$$

(b) $\mathcal{D}_{dec_\Delta} \preceq [\![dec_\Delta]\!]$ in $\Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$.

(c) $[\![dec_\Delta]\!] \preceq \mathcal{D}_{dec_\Delta}$ in $\Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$.

(d) For all P and c,

$$[\![\Delta \mid \varnothing \vdash P]\!]([\![dec_\Delta]\!]) = [c] \implies \mathcal{E}(P)(dec_\Delta) = [c].$$

The theorem follows trivially from the latter statements.

(a) This part follows by an essentially routine Rule Induction; in particular we prove

$$\text{for all } dec_\Delta \vdash P \Downarrow^e \underline{c}, \quad [\![\Delta \mid \varnothing \vdash P]\!]([\![dec_\Delta]\!]) = [c].$$

Let us consider property closure of the rule $\Downarrow^e$ FAP for some F_j. The inductive hypotheses are

$$[\![\Delta \mid \varnothing \vdash P_r]\!]([\![dec_\Delta]\!]) = [n_r]$$

where $1 \leq r \leq k_j$ and

$$[\![\Delta \mid \varnothing \vdash E_{\mathsf{F}_j}[\underline{n_1}, \dots, \underline{n_{k_j}}/x_1, \dots, x_{k_j}]]\!]([\![dec_\Delta]\!]) = [n].$$

Then we have

$$\begin{aligned}
&[\![\Delta \mid \varnothing \vdash \mathsf{F}_j(P_1, \dots, P_{k_j})]\!]([\![dec_\Delta]\!]) \\
&\quad = \quad ([\![dec_\Delta]\!]_j)_{\vec{\bot}}([\![\Delta \mid \varnothing \vdash P_1]\!]([\![dec_\Delta]\!]), \dots, [\![\Delta \mid \varnothing \vdash P_{k_j}]\!]([\![dec_\Delta]\!])) \\
&\quad = \quad ([\![dec_\Delta]\!]_j)_{\vec{\bot}}([n_1], \dots, [n_{k_j}]) \\
&\quad = \quad [\![dec_\Delta]\!]_j(n_1, \dots, n_{k_j}) \\
&\quad =_* \quad cur([\![\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E_{\mathsf{F}_j}]\!])([\![dec_\Delta]\!])(n_1, \dots, n_{k_j}) \\
&\quad = \quad [\![\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E_{\mathsf{F}_j}]\!]([\![dec_\Delta]\!], (n_1, \dots, n_{k_j})) \\
&\quad = \quad [\![\Delta \mid \varnothing \vdash E_{\mathsf{F}_j}[\underline{n_1}, \dots, \underline{n_{k_j}}/x_1, \dots, x_{k_j}]]\!]([\![dec_\Delta]\!]) \\
&\quad = \quad [n]
\end{aligned}$$

where step $*$ uses the fixpoint property of $[\![dec_\Delta]\!]$ and the penultimate equality follows from Lemma 7.4.3 (with $n = 0$ and $n' = k_j$).

(b) By definition of the partial order on the product cpo $\Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts}$ and the partial order on each cpo $[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts}$, we can see that $\mathcal{D}_{dec_\Delta} \preceq [\![dec_\Delta]\!]$ in $\Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts}$ iff for each j between 1 and m we have

$$\mathcal{D}(\mathsf{F}_j)(dec_\Delta) \preceq [\![dec_\Delta]\!]_j \qquad \text{in} \qquad [\mathbb{Z}^{k_j}, \mathbb{Z}_\bot]_{cts},$$

if and only if for all $(n_1 \dots, n_{k_j}) \in \mathbb{Z}^{k_j}$ and $n \in \mathbb{Z}$

$$dec_\Delta \vdash \mathsf{F}_j(\underline{n_1} \dots, \underline{n_{k_j}}) \Downarrow^e \underline{n} \Longrightarrow [\![dec_\Delta]\!]_j(n_1 \dots, n_{k_j}) = [n].$$

So suppose that $dec_\Delta \vdash \mathsf{F}_j(\underline{n_1} \dots, \underline{n_{k_j}}) \Downarrow^e \underline{n}$. Then it follows by part (a) that

$$[\![\Delta \mid \varnothing \vdash \mathsf{F}_j(\underline{n_1} \dots, \underline{n_{k_j}})]\!]([\![dec_\Delta]\!]) = [n].$$

But by definition of the denotational semantics we have

$$[\![\Delta \mid \varnothing \vdash \mathsf{F}_j(\underline{n_1} \dots, \underline{n_{k_j}})]\!]([\![dec_\Delta]\!]) = [\![dec_\Delta]\!]_j(n_1 \dots, n_{k_j})$$

as so part (b) follows.

(c) Because $[\![dec_\Delta]\!]$ is defined as the least fixpoint of the function $\Phi : \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts} \to \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$, to prove (c) it suffices to show that $\mathcal{D}_{dec_\Delta}$ is a prefixed point of Φ,

$$\Phi(\mathcal{D}_{dec_\Delta}) \preceq \mathcal{D}_{dec_\Delta} \text{ in } \Pi_{j=1}^{j=m}[\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}$$

that is, for each j between 1 and m we have

$$\Phi_j(\mathcal{D}_{dec_\Delta}) \preceq \mathcal{D}(\mathsf{F}_j)(dec_\Delta) \text{ in } [\mathbb{Z}^{k_j}, \mathbb{Z}_\perp]_{cts}. \tag{1}$$

Using the definition of Φ_j on page 83 and of $\mathcal{D}(\mathsf{F}_j)(dec_\Delta)$ we see that (1) holds if for all $(n_1 \ldots, n_{k_j}) \in \mathbb{Z}^{k_j}$ and $n \in \mathbb{Z}$

$$[\![\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E_{\mathsf{F}_j}]\!](\mathcal{D}_{dec_\Delta}, (n_1 \ldots, n_{k_j})) = [n] \quad \Longrightarrow \quad dec_\Delta \vdash \mathsf{F}_j(\underline{n_1} \ldots, \underline{n_{k_j}}) \Downarrow^e \underline{n} \tag{2}$$

and so by ($\Downarrow^e$ FAP), we see that (2) holds if

$$[\![\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E_{\mathsf{F}_j}]\!](\mathcal{D}_{dec_\Delta}, (n_1 \ldots, n_{k_j})) = [n] \quad \Longrightarrow \quad dec_\Delta \vdash E_{\mathsf{F}_j}[\underline{n_1} \ldots, \underline{n_{k_j}}/\vec{x}] \Downarrow^e \underline{n} \tag{3}$$

Instead of proving the previous statement we prove the following stronger statement (4) and then deduce (3) from (4): For all E, $(n_1 \ldots, n_{k_j})$ and c,

$$[\![\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E]\!](\mathcal{D}_{dec_\Delta}, (n_1 \ldots, n_{k_j})) = [c] \quad \Longrightarrow \quad dec_\Delta \vdash E[\underline{n_1} \ldots, \underline{n_{k_j}}/\vec{x}] \Downarrow^e \underline{c} \tag{4}$$

We prove (4) by Rule Induction on E. We shall look at just one case, when E is of the form $\mathsf{F}'_{j'}(E'_1, \ldots, E'_{k_{j'}})$. In this case we see that $[\![\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E]\!](\mathcal{D}_{dec_\Delta}, (n_1 \ldots, n_{k_j})) = [c]$ means that there are $m_r \in \mathbb{Z}$ where $1 \leq r \leq k_{j'}$ for which

$$[\![\Delta \mid \Gamma_{\mathsf{F}_j} \vdash E'_r]\!](\mathcal{D}_{dec_\Delta}, (n_1 \ldots, n_{k_j})) = [m_r] \text{ and } (\mathcal{D}_{dec_\Delta})_{j'}(m_1, \ldots, m_{k_{j'}}) = [c]$$

Using the latter assertions we see that by induction we have for each r

$$dec_\Delta \vdash E'_r[\underline{n_1} \ldots, \underline{n_{k_j}}/\vec{x}] \Downarrow^e \underline{m_r} \tag{5}$$

and that by definition of $\mathcal{D}_{dec_\Delta}$,

$$dec_\Delta \vdash \mathsf{F}_{j'}(\underline{m_1}, \ldots, \underline{m_{k_{j'}}}) \Downarrow^e \underline{c} \tag{6}$$

Applying ($\Downarrow^e$ FAP) to (5) and (6) we obtain

$$dec_\Delta \vdash \underbrace{\mathsf{F}_{j'}(E'_1[\underline{n_1}, \ldots, \underline{n_{k_j}}/\vec{x}], \ldots, E'_{k_{j'}}[\underline{n_1}, \ldots, \underline{n_{k_j}}/\vec{x}])}_{E[\underline{n_1}, \ldots, \underline{n_{k_j}}/\vec{x}]} \Downarrow^e \underline{c}$$

as required. We omit the other inductive cases. Therefore (4) holds and hence so does (3).

(d) This is proved by Rule Induction on P. We shall just consider the induction step when P is $\mathsf{F}_j(P'_1, \ldots, P'_{k_j})$. In this case, using the definition of the semantics, if

$$[\![\Delta \mid \varnothing \vdash P]\!]([\![dec_\Delta]\!]) = [c]$$

then there must be m_r where $1 \leq r \leq m_{k_j}$ for which

$$[\![\Delta \mid \varnothing \vdash P'_r]\!]([\![dec_\Delta]\!]) = [m_r] \tag{7}$$

and

$$[\![dec_\Delta]\!]_j(m_1, \ldots, m_{k_j}) = [c] \tag{8}$$

By induction, (7) gives

$$dec_\Delta \vdash P'_r \Downarrow^e \underline{m_r} \tag{9}$$

and by parts (b) and (c) we have $[\![dec_\Delta]\!] = \mathcal{D}_{dec_\Delta}$, so (8) gives

$$dec_\Delta \vdash \mathsf{F}_j(\underline{m_1}, \ldots, \underline{m_{k_j}}) \Downarrow^e \underline{c} \tag{10}$$

Applying ($\Downarrow^e$ FAP) to (9) and (10) we obtain $dec_\Delta \vdash P \Downarrow^e \underline{c}$ as required, that is $\mathcal{E}(P)(dec_\Delta) = [c]$.

□

7.5 Further Denotational Semantics

Motivation 7.5.1 Consider a variable $x :: \mathsf{int}$. In the lazy or call-by-name language, a program, rather than a value, may be passed to a variable. This program may loop. Thus we model a variable of type int as an element of $\mathbb{Z}_\perp$, rather than $\mathbb{Z}$, with $\perp$ modelling a looping program.

Consider a function identifier $\mathsf{F} :: 1$. In the language $\mathbb{TUR}^e$, which is eager (call-by-value) we thought of each F as acting on an integer value, and returning a possibly looping program. Thus we modelled F as a function $f : \mathbb{Z} \to \mathbb{Z}_\perp$. In $\mathbb{TUR}^l$, we think of F as acting on an integer *program* and returning a possibly looping program. Thus there is the possiblilty that the program which F acts on may itself loop and so we shall model F by a function of the form $\mathbb{Z}_\perp \to \mathbb{Z}_\perp$.

In general, we model an identifier $\mathsf{F} :: k$ by a function of the form $(\mathbb{Z}_\perp)^k \to \mathbb{Z}_\perp$, and model a type assignment of the form $\Delta \mid \Gamma \vdash E :: \sigma$ by a continuous function of the form

$$[\![\Delta \mid \Gamma \vdash E :: \sigma]\!] : (\Pi_{j=1}^{j=m}[(\mathbb{Z}_\perp)^{k_j}, \mathbb{Z}_\perp]_{cts}) \times (\mathbb{Z}_\perp)^n \longrightarrow X_\perp$$

where X is $\mathbb{Z}$ if σ is int and $\mathbb{B}$ if σ is bool.

Definitions 7.5.2 We define the functions $[\![\Delta \mid \Gamma \vdash E]\!]$ by the rules in Table 7.2. Note that as $\mathbb{TUR}^l$ satisfies subject reduction, we omit the types from the expressions to save space.

Theorem 7.5.3 For all $c \in \mathbb{Z} \cup \mathbb{B}$ we have

$$[\![\Delta \mid \varnothing \vdash P]\!]([\![dec_\Delta]\!]) = [c] \iff dec_\Delta \vdash P \Downarrow^l \underline{c}.$$

Proof The proof is identical in principle to the proof of Theorem 7.4.4. Try looking at the details of induction steps involving expressions of the form $\mathsf{F}_j(E_1, \ldots, E_{k_j})$. □

$$
\begin{array}{rcl}
[\![\Delta \mid \Gamma \vdash x_i]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & e_i \\
[\![\Delta \mid \Gamma \vdash \underline{c}]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & [c] \\
[\![\Delta \mid \Gamma \vdash E_1\ op\ E_2]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & (\iota \circ op)_{\vec{\bot}}\,([\![\Delta \mid \Gamma \vdash E_1]\!](\vec{f},\vec{e})\ ,\ [\![\Delta \mid \Gamma \vdash E_2]\!](\vec{f},\vec{e})) \\
[\![\Delta \mid \Gamma \vdash \mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{cond_{\bot_{\mathbb{B}}}([\![\Delta \mid \Gamma \vdash E_1]\!](\vec{f},\vec{e}), [\![\Delta \mid \Gamma \vdash E_2]\!](\vec{f},\vec{e}), [\![\Delta \mid \Gamma \vdash E_3]\!](\vec{f},\vec{e}))} \\
[\![\Delta \mid \Gamma \vdash \mathsf{F}_j(E_1,\ldots,E_{k_j})]\!](\vec{f},\vec{e}) & \stackrel{\text{def}}{=} & \\
& & f_j([\![\Delta \mid \Gamma \vdash E_1]\!](\vec{f},\vec{e}),\ldots,[\![\Delta \mid \Gamma \vdash E_{k_j}]\!](\vec{f},\vec{e})) \\
op & : & \mathbb{Z} \times \mathbb{Z} \longrightarrow X \\
\iota \circ op & : & \mathbb{Z} \times \mathbb{Z} \longrightarrow X_\bot \\
(\iota \circ op)_{\vec{\bot}} & : & \mathbb{Z}_\bot \times \mathbb{Z}_\bot \longrightarrow X_\bot \\
cond_{\bot_{\mathbb{B}}} & : & \mathbb{B}_\bot \times X_\bot \times X_\bot \longrightarrow X_\bot \\
f_j & : & (\mathbb{Z}_\bot)^{k_j} \longrightarrow \mathbb{Z}_\bot
\end{array}
$$

Table 7.2: Denotational Semantics $[\![\Delta \mid \Gamma \vdash E]\!]$ in $\mathbb{TUR}^l$