Midlands Graduate School
in the
Foundations of Computer Science

Operational Semantics, Abstract Machines, and Correctness

Roy L. Crole

University of Leicester, 8th to 12th April 2006

University of Nottingham, 16th to 20th April 2007

University of Birmingham, 14th to 18th April 2008

Preface

These notes are to support five lectures given at Midlands Graduate Schools in the Foundations of Computer Science.

In outline, two languages are presented, a simple imperative language with state, and a simple pure (that is, stateless) eager functional language. For each language we present an evaluation (natural) semantics, an abstract machine, and we prove that the machine and evaluation semantics are mutually correct. This forms a nice story for a course of five (rapid) lectures.

We do not cover types in great detail (polymorphism and type inference are not covered). Also, we do not cover transition (Plotkin) semantics, nor any form of lazy evaluation for the functional language.

The MGS notes are based on previous courses given by the author which do cover these topics, and also denotational semantics. The notes for the previous courses can be obtained from the author, as well as an implementation (and descriptive report) of an operational semantics which is a considerable superset of the languages presented in the MGS course.

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.

First edition 2006 (revised 2007,2008). Based upon author's notes from 1999 (revised 2000, 2001, 2003).

Contents

List of Tables

List of Figures

1

Operational Semantics for an Imperative Language $\mathbb{IMP}$

1.1 Introduction

Motivation 1.1.1 We shall look at a formal definition of a simple imperative language which we call $\mathbb{IMP}$. Here we define the *syntax* of this language and its *type system*—in a real programming language, it is the job of the compiler to do type checking. In Chapter 1 we then describe how programs in the language execute—the so called *operational semantics* of $\mathbb{IMP}$. This corresponds to the run-time of a real language.

The program expressions of the $\mathbb{IMP}$ 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 (some of) the locations. The commands of the language are "instructions" for changing the state—just as in any real imperative language.

A *configuration* in $\mathbb{IMP}$ consists of a program expression together with a specified state—in a real language, this would correspond to a real program and a given machine (memory) state. If the program expression happens to be a command, the configuration executes (or runs) by using the information coded by the command to change the state. The final result of the execution is given by the state at the end of execution—the details are in Chapter 1.

1.2 Expressions

Definitions 1.2.1 We begin to describe formally the language $\mathbb{IMP}$. The first step is to give a definition of the syntax of the language. In this course, syntax will in fact be abstract syntax—every syntactic object will be a finitely branching tree—see page 62. The syntax of $\mathbb{IMP}$ will be built out of various sets of symbols. These are

$\mathbb{Z}$	$\stackrel{\text{def}}{=}$	$\{\ldots,-1,0,1,\ldots\}$	the set of **integers**;
$\mathbb{B}$	$\stackrel{\text{def}}{=}$	$\{T,F\}$	the set of **Booleans**;
Loc	$\stackrel{\text{def}}{=}$	$\{\mathsf{L}_1,\mathsf{L}_2,\ldots\}$	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 $\mathbb{Z} \cup \mathbb{B}$, and l over Loc. 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 also that we write $\underline{c}$ to indicate that the constant c is an $\mathbb{IMP}$ program expression. Given (for example) $\underline{2}$ and $\underline{3}$ we cannot add these "numbers" until our programming language $\mathbb{IMP}$ instructs that this may happen: the syntax tree $\underline{2} + \underline{3}$ is not the same thing as the tree $\underline{5}$! 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 program expressions of the language $\mathbb{IMP}$. The set Exp of **program expressions** of the language is inductively defined by the grammar

P	$::=$	$\underline{c}$	constant
	$\mid$	l	memory location
	$\mid$	$iop(P, P')$	integer operator
	$\mid$	$bop(P, P')$	boolean operator
	$\mid$	skip	do nothing
	$\mid$	$\mathsf{assign}(l, P')$	assignment
	$\mid$	$\mathsf{sequence}(P, P')$	conditional
	$\mid$	$\mathsf{cond}(P, P', P'')$	while loop
	$\mid$	$\mathsf{while}(P, P')$	sequencing

where each program expression is a finite tree, whose nodes are labelled with constructors. Note that *iop* ranges over *IOpr* and *bop* ranges over *BOpr*. Also, *op* ranges over $IOpr \cup BOpr$. We immediately adopt the following abbreviations (known as syntactic **sugar**):

- We write $P\ iop\ P'$ for the finite tree $iop(P, P')$;
- $P\ bop\ P'$ for $bop(P, P')$;
- $l := P'$ for $\mathsf{assign}(l, P')$;
- $P\ ;\ P'$ for $\mathsf{sequence}(P, P')$;
- $\mathsf{if}\ P\ \mathsf{then}\ P'\ \mathsf{else}\ P''$ for $\mathsf{cond}(P, P', P'')$; and

• while P do P' for while(P,P').

We shall also adopt the following bracketing and scoping conventions:

• Arithmetic operators (generally denoted *op*) group to the left. Thus $P_1\ op\ P_2\ op\ P_3$ abbreviates $(P_1\ op\ P_2)\ op\ P_3$ with the expected extension to any finite number of integer expressions.

• Sequencing associates to the right.

Remark 1.2.2 We will usually denote elements of any given set of syntactic objects by one or two fixed symbols. So for example, P is always used to denote program expressions, that is, elements of *Exp*. We shall occasionally also use Q to denote a program expression.

We shall use brackets as informal punctuation when writing expressions. As an exercise, draw the syntax trees for

$$\text{if } P \text{ then } P_1 \text{ else } (P_2\ ;P_3) \qquad\qquad (\text{if } P \text{ then } P_1 \text{ else } P_2)\ ;P_3.$$

1.3 Types

Motivation 1.3.1 We shall asssume that you have some basic familiarity with types, such as that gained from programming at a typical undergraduate level. In this course, we shall only ever consider compile time checks.

If a program expression P can be assigned a type σ we write this as $P :: \sigma$ and call the statement a **type assignment**. A programming language will algorithmically encode[1] certain rules for deriving type assignments. In fact, such type assignments are often inductively defined. Given P and σ, **type checking** is the process of checking that $P :: \sigma$ is valid.

Definitions 1.3.2 The types of the language $\mathbb{IMP}$ are given by the grammar

$$\sigma \quad ::= \quad \text{int} \mid \text{bool} \mid \text{cmd}$$

We shall define a **location environment** $\mathcal{L}$ to be a finite set of (location, type) pairs. A pair (l,σ) will be written $l :: \sigma$. The *types in a location environment are either* int *or* bool *and the locations are all required to be different*. We write a typical location environment as

$$\mathcal{L} = l_1 :: \text{int}, \ldots, l_n :: \text{int},\ l_{n+1} :: \text{bool}, \ldots, l_m :: \text{bool}$$

[1]This statement is perhaps overselling the truth with regard to certain languages. The lectures will give further details.

$$\frac{}{\underline{n} :: \mathsf{int}}\ [\text{any } n \in \mathbb{Z}] \quad :: \text{INT} \qquad \frac{}{\underline{T} :: \mathsf{bool}} :: \text{TRUE} \qquad \frac{}{\underline{F} :: \mathsf{bool}} :: \text{FALSE}$$

$$\frac{}{l :: \mathsf{int}}\ [l :: \mathsf{int} \in \mathcal{L}] \quad :: \text{INTLOC} \qquad \frac{}{l :: \mathsf{bool}}\ [l :: \mathsf{bool} \in \mathcal{L}] \quad :: \text{BOOLLOC}$$

$$\frac{P_1 :: \mathsf{int} \quad P_2 :: \mathsf{int}}{P_1\ iop\ P_2 :: \mathsf{int}}\ [\ iop \in IOpr] \quad :: \text{IOP}$$

$$\frac{P_1 :: \mathsf{int} \quad P_2 :: \mathsf{int}}{P_1\ bop\ P_2 :: \mathsf{bool}}\ [\ bop \in BOpr] \quad :: \text{BOP}$$

$$\frac{}{\mathsf{skip} :: \mathsf{cmd}} :: \text{SKIP} \qquad \frac{l :: \sigma \quad P :: \sigma}{l := P :: \mathsf{cmd}}\ [\sigma \text{ is int or bool}] \quad :: \text{ASS}$$

$$\frac{P_1 :: \mathsf{cmd} \quad P_2 :: \mathsf{cmd}}{P_1 ; P_2 :: \mathsf{cmd}} :: \text{SEQ}$$

$$\frac{P_1 :: \mathsf{bool} \quad P_2 :: \mathsf{cmd} \quad P_3 :: \mathsf{cmd}}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 :: \mathsf{cmd}} :: \text{COND} \qquad \frac{P_1 :: \mathsf{bool} \quad P_2 :: \mathsf{cmd}}{\mathsf{while}\ P_1\ \mathsf{do}\ P_2 :: \mathsf{cmd}} :: \text{LOOP}$$

Table 1.1: $\mathbb{IMP}$ type assignments $P :: \sigma$

and we leave out the set braces { and }. Given a location environment $\mathcal{L}$, then a program expression P built up using only locations which appear in $\mathcal{L}$ can (sometimes) be assigned a type; we write $P :: \sigma$ to indicate this, and $P :: \sigma$ is called a **type assignment**. Such type assignments are defined inductively using the rules in Table 1.1.

Example 1.3.3 We give an example of a deduction of a type assignment, given the (concrete) location environment $\mathsf{L}_1 :: \mathsf{int}, \mathsf{L}_2 :: \mathsf{int}$ of the first two locations of memory.

$$\frac{\dfrac{\dfrac{}{\mathsf{L}_1 :: \mathsf{int}} \qquad \dfrac{}{\underline{1} :: \mathsf{int}}}{\mathsf{L}_1 \leq \underline{1} :: \mathsf{bool}} \qquad \dfrac{\mathcal{D}1 \qquad \mathcal{D}2 \qquad \dfrac{\mathcal{D}3 \qquad \mathcal{D}4}{\mathsf{L}_1 := \mathsf{L}_1 - 1 ; \mathsf{L}_2 := \mathsf{L}_2 * \mathsf{L}_1 :: \mathsf{cmd}}\,(\dagger)}{\mathsf{if}\ \mathsf{L}_1 = 1\ \mathsf{then}\ \mathsf{L}_2 := \underline{1}\ \mathsf{else}\ (\mathsf{L}_1 := \mathsf{L}_1 - 1 ; \mathsf{L}_2 := \mathsf{L}_2 * \mathsf{L}_1) :: \mathsf{cmd}}}{\mathsf{while}\ \mathsf{L}_1 \leq \underline{1}\ \mathsf{do}\ (\mathsf{if}\ \mathsf{L}_1 = 1\ \mathsf{then}\ \mathsf{L}_2 := \underline{1}\ \mathsf{else}\ (\mathsf{L}_1 := \mathsf{L}_1 - 1 ; \mathsf{L}_2 := \mathsf{L}_2 * \mathsf{L}_1)) :: \mathsf{cmd}}$$

It is an exercise to write down the missing deductions. See the note below, and the example after it.

$\ggg$ **NOTE 1.3.4** ***In order to produce the missing deductions, consider the following situation***

$$\frac{?}{P :: \sigma}\, R$$

in which $P :: \sigma$ ***is given. To see which rule*** R ***should be, look at the outermost constructor of*** P***. This will determine the rule*** R***. For example, look at*** $(\dagger)$ ***in Example 1.3.3. We have***

$$\frac{?}{\mathsf{L}_1 := \mathsf{L}_1 - 1\, ;\, \mathsf{L}_2 := \mathsf{L}_2 * \mathsf{L}_1 :: \mathsf{cmd}}\, R$$

and the outermost constructor is **sequence*****. Thus rule*** R ***is*** $::$ ***SEQ.***

Example 1.3.5 Perform type checking for $\mathsf{L}_1 := \mathsf{L}_1 + \underline{4} :: \mathsf{cmd}$ given $\mathcal{L} \stackrel{\text{def}}{=} \mathsf{L}_1 :: \mathsf{int}$. We do this by giving a deduction. The outermost constructor is assign, thus the final rule used must be $::$ ASS. Working backwards we have

$$\frac{\dfrac{?}{\mathsf{L}_1 :: \sigma}\, ?? \qquad \dfrac{???}{\mathsf{L}_1 + \underline{4} :: \sigma}\, ????}{\mathsf{L}_1 := \mathsf{L}_1 + \underline{4} :: \mathsf{cmd}}\, :: \textsc{ass}$$

Given $\mathcal{L}$, we must have $\sigma = \mathsf{int}$, ?? $=::$ INTLOC and ? is blank! Further, ???? must be $::$ IOP. You can fill in ??? for yourself.

1.4 An Evaluation Relation

Motivation 1.4.1 We shall now describe an operational semantics for $\mathbb{IMP}$ which, in the case of integer expressions, specifies how such program expressions can compute *directly* to integers. The operational semantics has assertions which look like $(P, s) \Downarrow (\underline{n}, s)$, meaning that the expression P evaluates to an integer with no state changes, and $(P, s) \Downarrow (\mathsf{skip}, s')$, meaning that the expression P evaluates to yield a change of state from s to s'. Types play a role here; in practice we can only derive such "evaluations" if P is of type int or cmd respectively.

Definitions 1.4.2 The set *States* of **states** is given by the subset of $[Loc, \mathbb{Z} \cup \mathbb{B}]_{par}$ consisting of those partial functions s with a *finite* domain of definition $dom(s)$. If $s \in States$ and $l \in Loc$ and $s(l)$ is defined, we refer to $s(l)$ as "the datum held in l at state s" or just "the contents of location l". Typical examples of states are $\langle \mathsf{L}_1 \mapsto 4, \mathsf{L}_2 \mapsto 5 \rangle$, $\langle \mathsf{L}_1 \mapsto 45, \mathsf{L}_2 \mapsto T, \mathsf{L}_3 \mapsto 2 \rangle$, and a general (finite) state will look like

$$s = \langle l_1 \mapsto c_1, \ldots, l_n \mapsto c_n \rangle$$

If $s \in States$, $l \in Loc$ and $c \in \mathbb{Z} \cup \mathbb{B}$, then there is a state denoted by $s\{l \mapsto c\} : Loc \to \mathbb{Z} \cup \mathbb{B}$ which is the partial function defined by

$$(s\{l \mapsto c\})(l') \stackrel{\text{def}}{=} \begin{cases} c & \text{if } l' = l \\ s(l') & \text{otherwise} \end{cases}$$

for each $l' \in Loc$. We say that state s is **updated** at l to c. As a simple exercise, if $\langle l_1 \mapsto c_1, \ldots, l_n \mapsto c_n \rangle$ is a general finite state, simplify the updated state

$$\langle l_1 \mapsto c_1, \ldots, l_n \mapsto c_n \rangle\{l \mapsto c\}$$

Definitions 1.4.3 We shall inductively define the set $\Downarrow$ of $\mathbb{IMP}$ **configurations**, where

$$\Downarrow \subseteq (Exp \times States) \times (Exp \times States)$$

by the rules in Table 1.2.

Examples 1.4.4

(1) Let us write P for $\mathsf{while}\ l > \underline{0}\ \mathsf{do}\ P'$ where P' is the command $l' := l' + \underline{2}\, ;\, l := l - \underline{1}$. Suppose that s is a state for which $s(l) = 1$ and $s(l') = 0$. A proof of $(P\,,s) \Downarrow (\mathsf{skip}\,, s\{l' \mapsto 2\}\{l \mapsto 0\})$ is given in Figure 1.1. It is an exercise to add in the appropriate labels to the deduction tree, and to fill in T.

(2) Show, by carefully examining deduction trees, that for any commands P_1, P_2 and P_3, and any states s and s', that

$$((P_1\,;P_2)\,;P_3\,,s) \Downarrow (\mathsf{skip}\,,s') \qquad \textit{implies} \qquad (P_1\,;(P_2\,;P_3)\,,s) \Downarrow (\mathsf{skip}\,,s')$$

(Thus the execution behaviour of (finite) sequences of commands is unchanged by re-arranging the sequence tree associatively.)

For any C_i, and s and s', suppose that $((P_1\,;P_2)\,;P_3\,,s) \Downarrow (\mathsf{skip}\,,s')$. Then the deduction tree must take the form

$$\dfrac{\dfrac{(P_1\,,s) \Downarrow (\mathsf{skip}\,,s_2) \quad (P_2\,,s_2) \Downarrow (\mathsf{skip}\,,s_3)}{(P_1\,;P_2\,,s) \Downarrow (\mathsf{skip}\,,s_3)} \qquad (P_3\,,s_3) \Downarrow (\mathsf{skip}\,,s')}{((P_1\,;P_2)\,;P_3\,,s) \Downarrow (\mathsf{skip}\,,s')}$$

Then, using $\Downarrow_{\text{SEQ}}$ twice with the evaluations at the leaves of the tree above, we can deduce that $(P_1\,;(P_2\,;P_3)\,,s) \Downarrow (\mathsf{skip}\,,s')$. The converse direction is similar.

(3) A student tries to do the previous problem using Rule Induction. She formulates the following proposition

$$\boxed{\forall x.\forall y.\forall z\ (\ P = (x\,;y)\,;z \ \textit{and}\ V = \mathsf{skip} \qquad \textit{implies} \qquad (x\,;(y\,;z)\,,s) \Downarrow (\mathsf{skip}\,,s')\ )}$$

about the evaluation relation, which *if* proved (by Rule Induction) for all $(P\,,s) \Downarrow (V\,,s')$ implies the result of the previous problem. This is correct—can you explain why?

$$\frac{}{(l\,,s)\Downarrow(\underline{s(l)}\,,s)}\;[\text{ provided } l\in\text{ domain of } s]\Downarrow\text{LOC} \qquad \frac{}{(\underline{c}\,,s)\Downarrow(\underline{c}\,,s)}\Downarrow\text{CONST}$$

$$\frac{(P_1\,,s)\Downarrow(\underline{n_1}\,,s)\quad (P_2\,,s)\Downarrow(\underline{n_2}\,,s)}{(P_1\;op\;P_2\,,s)\Downarrow(\underline{n_1\;op\;n_2}\,,s)}\Downarrow\text{OP}$$

$$\frac{}{(\mathsf{skip}\,,s)\Downarrow(\mathsf{skip}\,,s)}\Downarrow\text{SKIP}$$

$$\frac{(P\,,s)\Downarrow(\underline{c}\,,s)}{(l:=P\,,s)\Downarrow(\mathsf{skip}\,,s\{l\mapsto c\})}\Downarrow\text{ASS}$$

$$\frac{(P_1\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)\quad (P_2\,,s_2)\Downarrow(\mathsf{skip}\,,s_3)}{(P_1\,;P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_3)}\Downarrow\text{SEQ}$$

$$\frac{(P\,,s_1)\Downarrow(\underline{T}\,,s_1)\quad (P_1\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)}{(\mathsf{if}\;P\;\mathsf{then}\;P_1\;\mathsf{else}\;P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)}\Downarrow\text{COND}_1$$

$$\frac{(P\,,s_1)\Downarrow(\underline{F}\,,s_1)\quad (P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)}{(\mathsf{if}\;P\;\mathsf{then}\;P_1\;\mathsf{else}\;P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)}\Downarrow\text{COND}_2$$

$$\frac{(P_1\,,s_1)\Downarrow(\underline{T}\,,s_1)\quad (P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)\quad (\mathsf{while}\;P_1\;\mathsf{do}\;P_2\,,s_2)\Downarrow(\mathsf{skip}\,,s_3)}{(\mathsf{while}\;P_1\;\mathsf{do}\;P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_3)}\Downarrow\text{LOOP}_1$$

$$\frac{(P_1\,,s)\Downarrow(\underline{F}\,,s)}{(\mathsf{while}\;P_1\;\mathsf{do}\;P_2\,,s)\Downarrow(\mathsf{skip}\,,s)}\Downarrow\text{LOOP}_2$$

Table 1.2: Evaluation Relation $(P\,,s)\Downarrow(P\,,s')$ in $\mathbb{IMP}$

$$\frac{\mathcal{D}_1 \qquad \mathcal{D}_2 \qquad \mathcal{D}_3}{(P\,,\, s) \Downarrow (\mathsf{skip}\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\})}$$

where $\mathcal{D}_1$ is

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

and $\mathcal{D}_2$ is

$$\frac{\dfrac{\dfrac{\dfrac{}{(l'\,,\, s) \Downarrow (\underline{0}\,,\, s)} \qquad \dfrac{}{(\underline{2}\,,\, s) \Downarrow (\underline{2}\,,\, s)}}{(l' + \underline{2}\,,\, s) \Downarrow (\underline{2}\,,\, s)}}{(l' := l' + \underline{2}\,,\, s) \Downarrow (\mathsf{skip}\,,\, s\{l'\mapsto 2\})} \qquad \dfrac{T}{(l := l - \underline{1}\,,\, s\{l'\mapsto 2\}) \Downarrow (\mathsf{skip}\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\})}}{(l' := l' + \underline{2}\,;\, l := l - \underline{1}\,,\, s) \Downarrow (\mathsf{skip}\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\})}$$

and $\mathcal{D}_3$ is

$$\frac{\dfrac{\dfrac{}{(l\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\}) \Downarrow (\underline{0}\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\})} \qquad \dfrac{}{(\underline{0}\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\}) \Downarrow (\underline{0}\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\})}}{(l > \underline{0}\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\}) \Downarrow (\underline{F}\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\})}}{(P\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\}) \Downarrow (\mathsf{skip}\,,\, s\{l'\mapsto 2\}\{l\mapsto 0\})}$$

Figure 1.1: An Example Deduction of an Evaluation

2
A Compiled CSS Machine

2.1 Architecture of the Machine

Motivation 2.1.1 We have seen that an operational semantics gives a useful 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 evaluating configurations. We provide just that in this chapter, by defining an abstract machine which executes via single step re-write rules.

Definitions 2.1.2 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:

$$\begin{aligned} ins &::= \mathsf{PUSH}(\underline{c}) \mid \mathsf{FETCH}(l) \mid \mathsf{OP}(op) \mid \mathsf{SKIP} \mid \mathsf{STO}(l) \mid \mathsf{BR}(C,C) \mid \mathsf{LOOP}(C,C) \\ C &::= - \mid ins : C \end{aligned}$$

where op is any operator, l is any location and c is any constant. The objects ins are CSS **instructions**. A **stack** S is produced by the grammar

$$S ::= - \mid \underline{c} : S$$

where c is any integer or Boolean. A **state** s is indeed an $\mathbb{IMP}$ state. We shall write $-$ to indicate an empty code or stack. We shall also abbreviate $C : -$ to C and $S : -$ to S.

A CSS **configuration** is a triple (C, S, s) whose components are defined as above. A CSS **re-write** takes the form

$$(C_1, S_1, s_1) \longmapsto (C_2, S_2, s_2)$$

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

$$\frac{}{(C_1, S_1, s_1) \longmapsto (C_2, S_2, s_2)}\ R$$

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 2.1, where each rule R is written

$$\boxed{C_1} \| \boxed{S_1} \| \boxed{s_1} \longmapsto \boxed{C_2} \| \boxed{S_2} \| \boxed{s_2}$$

$\boxed{\mathsf{PUSH}(\underline{c}):C}\boxed{S}\boxed{s}$	$\longmapsto$	$\boxed{C}\boxed{\underline{c}:S}\boxed{s}$
$\boxed{\mathsf{FETCH}(l):C}\boxed{S}\boxed{s}$	$\longmapsto$	$\boxed{C}\boxed{\underline{s(l)}:S}\boxed{s}$
$\boxed{\mathsf{OP}(\,op\,):C}\boxed{\underline{n_1}:\underline{n_2}:S}\boxed{s}$	$\longmapsto$	$\boxed{C}\boxed{\underline{n_1\;op\;n_2}:S}\boxed{s}$
$\boxed{\mathsf{SKIP}:C}\boxed{S}\boxed{s}$	$\longmapsto$	$\boxed{C}\boxed{S}\boxed{s}$
$\boxed{\mathsf{STO}(l):C}\boxed{\underline{c}:S}\boxed{s}$	$\longmapsto$	$\boxed{C}\boxed{S}\boxed{s\{l\mapsto c\}}$
$\boxed{\mathsf{BR}(C_1,C_2):C}\boxed{\underline{T}:S}\boxed{s}$	$\longmapsto$	$\boxed{P_1:C}\boxed{S}\boxed{s}$
$\boxed{\mathsf{BR}(C_1,C_2):C}\boxed{\underline{F}:S}\boxed{s}$	$\longmapsto$	$\boxed{P_2:C}\boxed{S}\boxed{s}$
$\boxed{\mathsf{LOOP}(C_1,C_2):C}\boxed{S}\boxed{s}$	$\longmapsto$	$\boxed{C_1:\mathsf{BR}(C_2:\mathsf{LOOP}(C_1,C_2),\mathsf{SKIP}):C}\boxed{S}\boxed{s}$

Table 2.1: The CSS Re-Writes

Motivation 2.1.3 We shall now compile $\mathbb{IMP}$ program expressions into CSS codes. We shall assume that any given program expression has already been through the type checking phase of compilation. We shall define a function $[\![-]\!]\colon Exp \to CSScodes$ which takes a CSS program expression and turns it into CSS code.

Definitions 2.1.4 The function $[\![-]\!]\colon Exp \to CSScodes$ is specified by the clauses in Table 2.2.

2.2 Correctness of the Machine

Motivation 2.2.1 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 1, the result matches that of the CSS machine, and vice versa. We make this precise in the following theorem:

Theorem 2.2.2 For all $n \in \mathbb{Z}$, $b \in \mathbb{B}$, $P_1 :: \mathsf{int}$, $P_2 :: \mathsf{bool}$, $P_3 :: \mathsf{cmd}$ and $s, s_1, s_2 \in States$ we have

$$\begin{array}{rcl}
(P_1\,,s) \Downarrow (\underline{n}\,,s) & \textit{iff} & \boxed{[\![P_1]\!]}\boxed{-}\boxed{s} \longmapsto^t \boxed{-}\boxed{\underline{n}}\boxed{s} \\
(P_2\,,s) \Downarrow (\underline{b}\,,s) & \textit{iff} & \boxed{[\![P_2]\!]}\boxed{-}\boxed{s} \longmapsto^t \boxed{-}\boxed{\underline{b}}\boxed{s} \\
(P_3\,,s_1) \Downarrow (\mathsf{skip}\,,s_2) & \textit{iff} & \boxed{[\![P_3]\!]}\boxed{-}\boxed{s_1} \longmapsto^t \boxed{-}\boxed{-}\boxed{s_2}
\end{array}$$

where $\longmapsto^t$ denotes the transitive closure of $\longmapsto$.

$$
\begin{aligned}
[\![\underline{c}]\!] &\stackrel{\text{def}}{=} \mathsf{PUSH}(\underline{c})\\
[\![l]\!] &\stackrel{\text{def}}{=} \mathsf{FETCH}(l)\\
[\![P_1\ op\ P_2]\!] &\stackrel{\text{def}}{=} [\![P_2]\!] : [\![P_1]\!] : \mathsf{OP}(op)\\
[\![l := P]\!] &\stackrel{\text{def}}{=} [\![P]\!] : \mathsf{STO}(l)\\
[\![\mathsf{skip}]\!] &\stackrel{\text{def}}{=} \mathsf{SKIP}\\
[\![P_1\,;P_2]\!] &\stackrel{\text{def}}{=} [\![P_1]\!] : [\![P_2]\!]\\
[\![\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2]\!] &\stackrel{\text{def}}{=} [\![P]\!] : \mathsf{BR}([\![P_1]\!], [\![P_2]\!])\\
[\![\mathsf{while}\ P_1\ \mathsf{do}\ P_2]\!] &\stackrel{\text{def}}{=} \mathsf{LOOP}([\![P_1]\!], [\![P_2]\!])
\end{aligned}
$$

Table 2.2: Compiling $\mathbb{IMP}$ into CSS Code

We shall not prove this theorem, although in Chapter 3 we shall prove a correctness theorem for an *interpreted* CSS machine. Once you are familiar with the proof, try to prove Theorem 2.2.2.

2.3 Executions (of Compiled code)

Examples 2.3.1

(1) Let s be a state for which $s(l) = 6$. Execute $\underline{10} - l$ on the CSS machine.

First, compile the program.

$$[\![\underline{10} - l]\!] \quad = \quad \mathsf{FETCH}(l) : \mathsf{PUSH}(\underline{10}) : \mathsf{OP}(-)$$

Then

$$
\begin{aligned}
\boxed{\mathsf{FETCH}(l) : \mathsf{PUSH}(\underline{10}) : \mathsf{OP}(-) \;\|\; - \;\|\; s} \quad &\longmapsto \quad \boxed{\mathsf{PUSH}(\underline{10}) : \mathsf{OP}(-) \;\|\; \underline{6} \;\|\; s}\\
&\longmapsto \quad \boxed{\mathsf{OP}(-) \;\|\; \underline{10} : \underline{6} \;\|\; s}\\
&\longmapsto \quad \boxed{- \;\|\; \underline{4} \;\|\; s}
\end{aligned}
$$

(2) Let s be a state for which $s(l) = 1$. Run the program $\mathsf{if}\ l \geq \underline{0}\ \mathsf{then}\ l := l - \underline{1}\ \mathsf{else\ skip}$. First compile

$$
\begin{aligned}
&[\![\mathsf{if}\ l \geq \underline{0}\ \mathsf{then}\ l := l - \underline{1}\ \mathsf{else\ skip}]\!]\\
&\qquad = \quad [\![l \geq \underline{0}]\!] : \mathsf{BR}([\![l := l - \underline{1}]\!], [\![\mathsf{skip}]\!])\\
&\qquad = \quad \mathsf{PUSH}(\underline{0}) : \mathsf{FETCH}(l) : \geq : \mathsf{BR}(\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l), \mathsf{SKIP})
\end{aligned}
$$

Then

$$
\begin{aligned}
&\boxed{\mathsf{PUSH}(\underline{0}) : \mathsf{FETCH}(l) :\geq: \mathsf{BR}(\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l), \mathsf{SKIP}) \parallel - \parallel s} \\
&\qquad \longmapsto \boxed{\mathsf{FETCH}(l) :\geq: \mathsf{BR}(\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l), \mathsf{SKIP}) \parallel \underline{0} \parallel s} \\
&\qquad \longmapsto \boxed{\geq: \mathsf{BR}(\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l), \mathsf{SKIP}) \parallel \underline{1} : \underline{0} \parallel s} \\
&\qquad \longmapsto \boxed{\mathsf{BR}(\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l), \mathsf{SKIP}) \parallel \underline{T} \parallel s} \\
&\qquad \longmapsto \boxed{\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l) \parallel - \parallel s} \\
&\qquad \longmapsto \boxed{\mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l) \parallel \underline{1} \parallel s} \\
&\qquad \longmapsto \boxed{\mathsf{OP}(-) : \mathsf{STO}(l) \parallel \underline{1} : \underline{1} \parallel s} \\
&\qquad \longmapsto \boxed{\mathsf{STO}(l) \parallel \underline{0} \parallel s} \\
&\qquad \longmapsto \boxed{- \parallel - \parallel s\{l \mapsto 0\}}
\end{aligned}
$$

3

Correctness of an Interpreted CSS Machine

3.1 Architecture of the Machine

Definitions 3.1.1 We make slight changes to the previous definitions. A CSS **code** C is a list which is produced by the following grammars:

$$ins ::= P \mid op \mid \mathsf{STO}(l) \mid \mathsf{BR}(P_1, P_2) \qquad\qquad C ::= \mathsf{nil} \mid ins : C$$

where P is any $\mathbb{IMP}$ expression, op is any operator, l is any location and P_1 and P_2 are any two commands. The objects ins are CSS **instructions**. A **stack** S is produced by the grammar

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

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.

The CSS re-writes are defined in Table 3.1, where each rule R is written

$$\boxed{C_1}\!\boxed{S_1}\!\boxed{s_1} \longmapsto \boxed{C_2}\!\boxed{S_2}\!\boxed{s_2}$$

3.2 A Correctness Theorem

Motivation 3.2.1 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 1, the result matches that of the CSS machine, and vice versa. We make this precise in the following theorem:

Theorem 3.2.2 For all $n \in \mathbb{Z}$, $b \in \mathbb{B}$, $P_1 :: \mathsf{int}$, $P_2 :: \mathsf{bool}$, $P_3 :: \mathsf{cmd}$ and $s, s_1, s_2 \in States$ we have

$$\begin{array}{rcl}
(P_1, s) \Downarrow (\underline{n}, s) & \textit{iff} & \boxed{P_1}\!\boxed{-}\!\boxed{s} \longmapsto^t \boxed{-}\!\boxed{\underline{n}}\!\boxed{s} \\
(P_2, s) \Downarrow (\underline{b}, s) & \textit{iff} & \boxed{P_2}\!\boxed{-}\!\boxed{s} \longmapsto^t \boxed{-}\!\boxed{\underline{b}}\!\boxed{s} \\
(P_3, s_1) \Downarrow (\mathsf{skip}, s_2) & \textit{iff} & \boxed{P_3}\!\boxed{-}\!\boxed{s_1} \longmapsto^t \boxed{-}\!\boxed{-}\!\boxed{s_2}
\end{array}$$

where $\longmapsto^t$ denotes the transitive closure of $\longmapsto$.

$$
\begin{array}{rcl}
\boxed{\underline{n} : C \parallel S \parallel s} & \longmapsto & \boxed{C \parallel \underline{n} : S \parallel s} \\
\boxed{P_1\ op\ P_2 : C \parallel S \parallel s} & \longmapsto & \boxed{P_2 : P_1 : \ op\ : C \parallel S \parallel s} \\
\boxed{l : C \parallel S \parallel s} & \longmapsto & \boxed{C \parallel \underline{s(l)} : S \parallel s} \\
\boxed{\ op\ : C \parallel \underline{n_1} : \underline{n_2} : S \parallel s} & \longmapsto & \boxed{C \parallel \underline{n_1\ op\ n_2} : S \parallel s} \\
\boxed{\underline{T} : C \parallel S \parallel s} & \longmapsto & \boxed{C \parallel \underline{T} : S \parallel s} \\
\boxed{\underline{F} : C \parallel S \parallel s} & \longmapsto & \boxed{C \parallel \underline{F} : S \parallel s} \\
\boxed{\mathsf{skip} : C \parallel S \parallel s} & \longmapsto & \boxed{C \parallel S \parallel s} \\
\boxed{l := P : C \parallel S \parallel s} & \longmapsto & \boxed{P : \mathsf{STO}(l) : C \parallel S \parallel s} \\
\boxed{\mathsf{STO}(l) : C \parallel \underline{n} : S \parallel s} & \longmapsto & \boxed{C \parallel S \parallel s\{l \mapsto n\}} \\
\boxed{(P_1 \,;\, P_2) : C \parallel S \parallel s} & \longmapsto & \boxed{P_1 : P_2 : C \parallel S \parallel s} \\
\boxed{\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : C \parallel S \parallel s} & \longmapsto & \boxed{P : \mathsf{BR}(P_1, P_2) : C \parallel S \parallel s} \\
\boxed{\mathsf{BR}(P_1, P_2) : C \parallel \underline{T} : S \parallel s} & \longmapsto & \boxed{P_1 : C \parallel S \parallel s} \\
\boxed{\mathsf{BR}(P_1, P_2) : C \parallel \underline{F} : S \parallel s} & \longmapsto & \boxed{P_2 : C \parallel S \parallel s} \\
\boxed{\mathsf{while}\ P_1\ \mathsf{do}\ P_2 : C \parallel S \parallel s} & \longmapsto & \boxed{P_1 : \mathsf{BR}((P_2 \,;\, \mathsf{while}\ P_1\ \mathsf{do}\ P_2), \mathsf{skip}) : C \parallel S \parallel s}
\end{array}
$$

Table 3.1: The CSS Re-Writes

3.3 Preliminary Results

Motivation 3.3.1 The proof method for Theorem 3.2.2 is as follows: For the $\Longrightarrow$ implication(s) we use Rule Induction for $\Downarrow$. For the $\Longleftarrow$ implication(s) we use Mathematical Induction on k, where of course $\kappa \longmapsto^t \kappa'$ iff for some $k \geq 1 \in \mathbb{N}$ and $\kappa_1, \ldots, \kappa_k$

$$\kappa = \longmapsto \kappa_1 \longmapsto \ldots \longmapsto \kappa_k = \kappa' \qquad k \text{ re-writes}$$

If it is not immediately clear to you how Mathematical Induction will be used, then look ahead to page 19. We shall need a few preliminary results before we can prove the theorem.

Lemma 3.3.2 The CSS machine re-writes are deterministic, that is each CSS configuration re-writes to a unique CSS configuration:

More precisely, if

$$\boxed{C \parallel S \parallel s} \longmapsto \boxed{C_1 \parallel S_1 \parallel s_1} \quad \textit{and} \quad \boxed{C \parallel S \parallel s} \longmapsto \boxed{C_2 \parallel S_2 \parallel s_2}$$

then $C_1 = C_2$, $S_1 = S_2$ and $s_1 = s_2$.

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

Lemma 3.3.3 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 S_i, states s_i and $k \in \mathbb{N}$,

$$\boxed{C_1 \parallel S_1 \parallel s_1} \longmapsto^k \boxed{C_2 \parallel S_2 \parallel s_2}$$

implies

$$\boxed{C_1 : C_3 \parallel S_1 : S_3 \parallel s_1} \longmapsto^k \boxed{C_2 : C_3 \parallel S_2 : S_3 \parallel s_2}$$

where we define $\longmapsto^0$ to be the identity binary relation on the set of all CSS configurations, and of course $\longmapsto^k$ means there are k re-writes when $k \geq 1$. 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 $\phi(k)$ holds for all $k \in \mathbb{N}$ where $\phi(k)$ is the assertion that for all appropriate codes, stacks and states

$$\boxed{C_1 \parallel S_1 \parallel s_1} \longmapsto^{k} \boxed{C_2 \parallel S_2 \parallel s_2}$$

implies

$$\boxed{C_1 : C_3 \parallel S_1 : S_3 \parallel s_1} \longmapsto^{k} \boxed{C_2 : C_3 \parallel S_2 : S_3 \parallel s_2}.$$

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

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

for all codes, stacks and states,

$$\boxed{C_1 \parallel S_1 \parallel s_1} \longmapsto^{k} \boxed{C_2 \parallel S_2 \parallel s_2}$$

implies

$$\boxed{C_1 : C_3 \parallel S_1 : S_3 \parallel s_1} \longmapsto^{k} \boxed{C_2 : C_3 \parallel S_2 : S_3 \parallel s_2}$$

holds for each $k \leq k_0$, then

for all codes, stacks and states,

$$\boxed{C_1 \parallel S_1 \parallel s_1} \longmapsto^{k_0+1} \boxed{C_2 \parallel S_2 \parallel s_2}$$

implies

$$\boxed{C_1 : C_3 \parallel S_1 : S_3 \parallel s_1} \longmapsto^{k_0+1} \boxed{C_2 : C_3 \parallel S_2 : S_3 \parallel s_2}.$$

Let us choose arbitrary codes, stacks and states for which

$$\boxed{C_1 \parallel S_1 \parallel s_1} \longmapsto^{k_0+1} \boxed{C_2 \parallel S_2 \parallel s_2}$$

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

$$\boxed{- \parallel S_1 \parallel s_1} \longmapsto^{k_0+1} \boxed{C_2 \parallel S_2 \parallel s_2}$$

implies

$$\boxed{- : C_3 \parallel S_1 : S_3 \parallel s_1} \longmapsto^{k_0+1} \boxed{C_2 : C_3 \parallel S_2 : S_3 \parallel s_2}$$

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

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

$$\boxed{\underline{n} : C_1 \parallel S_1 \parallel s_1} \longmapsto^{k_0+1} \boxed{C_2 \parallel S_2 \parallel s_2}$$

We need to prove that

$$\boxed{\underline{n}:C_1:C_3 \parallel S_1:S_3 \parallel s_1} \longmapsto^{k_0+1} \boxed{C_2:C_3 \parallel S_2:S_3 \parallel s_2} \tag{1}$$

By Lemma 3.3.2 we must have[1]

$$\boxed{\underline{n}:C_1 \parallel S_1 \parallel s_1} \longmapsto^{1} \boxed{C_1 \parallel \underline{n}:S_1 \parallel s_1} \longmapsto^{k_0} \boxed{C_2 \parallel S_2 \parallel s_2}$$

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

$$\boxed{C_1:C_3 \parallel \underline{n}:S_1:S_3 \parallel s_1} \longmapsto^{k_0} \boxed{C_2:C_3 \parallel S_2:S_3 \parallel s_2} \tag{2}$$

But

$$\boxed{\underline{n}:C_1:C_3 \parallel S_1:S_3 \parallel s_1} \longmapsto^{1} \boxed{C_1:C_3 \parallel \underline{n}:S_1:S_3 \parallel s_1} \tag{3}$$

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

(*Case* C_1 *is* $\mathsf{BR}(P_1,P_2):C_1$): Assume that[2]

$$\boxed{\mathsf{BR}(P_1,P_2):C_1 \parallel \underline{T}:S_1 \parallel s_1} \longmapsto^{k_0+1} \boxed{C_2 \parallel S_2 \parallel s_2}$$

We need to prove that

$$\boxed{\mathsf{BR}(P_1,P_2):C_1:C_3 \parallel \underline{T}:S_1:S_3 \parallel s_1} \longmapsto^{k_0+1} \boxed{C_2:C_3 \parallel S_2:S_3 \parallel s_2} \tag{4}$$

Now

$$\boxed{\mathsf{BR}(P_1,P_2):C_1 \parallel \underline{T}:S_1 \parallel s_1} \longmapsto^{1} \boxed{P_1:C_1 \parallel S_1 \parallel s_1}$$

and so by induction we have

$$\boxed{P_1:C_1:C_3 \parallel S_1:S_3 \parallel s_1} \longmapsto^{k_0} \boxed{C_2:C_3 \parallel S_2:S_3 \parallel s_2} \tag{5}$$

But

$$\boxed{\mathsf{BR}(P_1,P_2):C_1:C_3 \parallel \underline{T}:S_1:S_3 \parallel s_1} \longmapsto^{1} \boxed{P_1:C_1:C_3 \parallel S_1:S_3 \parallel s_1} \tag{6}$$

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

Lemma 3.3.4 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]We often use determinism of $\longmapsto$ in the next few pages, without always quoting Lemma 3.3.2. Informally, any configuration has a unique sequence of re-writes (if there are any).

[2]Given that the code begins with the instruction $\mathsf{BR}(P_1,P_2)$ 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

$$\boxed{C_1 : C_2 \parallel S \parallel s} \longmapsto^{k} \boxed{- \parallel S'' \parallel s''}$$

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

$$\begin{aligned} \boxed{C_1 \parallel S \parallel s} &\longmapsto^{k_1} \boxed{- \parallel S' \parallel s'} \\ \boxed{C_2 \parallel S' \parallel s'} &\longmapsto^{k_2} \boxed{- \parallel S'' \parallel s''} \end{aligned}$$

where $k_1 + k_2 = k$.

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

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

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

$$\boxed{C_1 : C_2 \parallel S \parallel s} \longmapsto^{k_0+1} \boxed{- \parallel S'' \parallel s''}$$

and then consider the possible forms that C_1 can take.

(*Case C_1 is* $\mathsf{while}\, P_1 \,\mathsf{do}\, P_2 : C_1$):

We suppose that

$$\boxed{\mathsf{while}\, P_1 \,\mathsf{do}\, P_2 : C_1 : C_2 \parallel S \parallel s} \longmapsto^{k_0+1} \boxed{- \parallel S'' \parallel s''}$$

and hence by Lemma 3.3.2

$$\begin{aligned} &\boxed{\mathsf{while}\, P_1 \,\mathsf{do}\, P_2 : C_1 : C_2 \parallel S \parallel s} \\ &\qquad \longmapsto^{1} \boxed{P_1 : \mathsf{BR}((P_2 \,;\, \mathsf{while}\, P_1 \,\mathsf{do}\, P_2), \mathsf{skip}) : C_1 : C_2 \parallel S \parallel s} \\ &\qquad \longmapsto^{k_0} \boxed{- \parallel S'' \parallel s''} \end{aligned}$$

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

$$\boxed{P_1 : \mathsf{BR}((P_2 \,;\, \mathsf{while}\, P_1 \,\mathsf{do}\, P_2), \mathsf{skip}) : C_1 \parallel S \parallel s} \longmapsto^{k_1} \boxed{- \parallel S' \parallel s'} \qquad (1)$$

and

$$\boxed{C_2 \parallel S' \parallel s'} \longmapsto^{k_2} \boxed{- \parallel S'' \parallel s''} \qquad (2)$$

But

$$\boxed{\mathsf{while}\, P_1 \,\mathsf{do}\, P_2 : C_1 \parallel S \parallel s} \longmapsto^{1} \boxed{P_1 : \mathsf{BR}((P_2 \,;\, \mathsf{while}\, P_1 \,\mathsf{do}\, P_2), \mathsf{skip}) : C_1 \parallel S \parallel s} \qquad (3)$$

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

Lemma 3.3.5 For all appropriate codes, stacks, states and natural numbers,

$$P :: \mathsf{int}\ \textit{and}\ \boxed{P \,\|\, S \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, S' \,\|\, s'} \quad \textit{implies}$$

$$s = s' \quad \textit{and} \quad S' = \underline{n} : S \text{ some } n \in \mathbb{Z} \quad \textit{and} \quad \boxed{P \,\|\, - \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, \underline{n} \,\|\, s}$$

and

$$P :: \mathsf{bool}\ \textit{and}\ \boxed{P \,\|\, S \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, S' \,\|\, s'} \quad \textit{implies}$$

$$s = s' \quad \textit{and} \quad S' = \underline{b} : S \text{ some } b \in \mathbb{B} \quad \textit{and} \quad \boxed{P \,\|\, - \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, \underline{b} \,\|\, s}$$

Proof A lengthy induction; as an exercise, experiment with structural induction on P and mathematical induction on the number of transitions. Which works? Both? Just one? □

3.4 Proving Theorem 3.2.2

Let us now give the proof of the correctness theorem:

Proof ($\Longrightarrow$): We use Rule Induction for $\Downarrow$, together with a case analysis on the types. If the type is int, only the rules for operators can be used in the deduction of the evaluation. We show property closure for just one example rule:

(*Case* $\Downarrow$ OP$_1$): The inductive hypotheses (where $P_i :: \mathsf{int}$) are

$$\boxed{P_1 \,\|\, - \,\|\, s} \longmapsto^{t} \boxed{- \,\|\, \underline{n_1} \,\|\, s} \quad \textit{and} \quad \boxed{P_2 \,\|\, - \,\|\, s} \longmapsto^{t} \boxed{- \,\|\, \underline{n_2} \,\|\, s}$$

Then we have

$$\begin{array}{rcl} \boxed{P_1\ op\ P_2 \,\|\, - \,\|\, s} & \longmapsto & \boxed{P_2 : P_1 :\ op \,\|\, - \,\|\, s} \\ \text{by Lemma 3.3.3 and inductive hypotheses} & \longmapsto^{t} & \boxed{P_1 :\ op \,\|\, \underline{n_2} \,\|\, s} \\ \text{by Lemma 3.3.3 and inductive hypotheses} & \longmapsto^{t} & \boxed{op \,\|\, \underline{n_1} : \underline{n_2} \,\|\, s} \\ & \longmapsto & \boxed{- \,\|\, \underline{n_1\ op\ n_2} \,\|\, s} \end{array}$$

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:

$$\text{for all } P :: \mathsf{int}, n, s, \quad \boxed{P \,\|\, - \,\|\, s} \longmapsto^{t} \boxed{- \,\|\, \underline{n} \,\|\, s} \quad \textit{implies} \quad (P, s) \Downarrow (\underline{n}, s).$$

But this statement is logically equivalent to

$$\text{for all } k, \quad \boxed{\text{for all } P :: \mathsf{int}, n, s, \quad \boxed{P \,\|\, - \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, \underline{n} \,\|\, s} \quad \textit{implies} \quad (P, s) \Downarrow (\underline{n}, s)}$$

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

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

(*Proof of* for all $k_0 \in \mathbb{N}$, $\phi(k)_{k \le k_0}$ *implies* $\phi(k_0+1)$): Suppose that for some arbitrary k_0, $P :: \mathsf{int}$, n and s

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

and then we prove $(P\,,s) \Downarrow (\underline{n}\,,s)$ by considering cases on P.

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

(*Case* P is $P_1\ op\ P_2$): Suppose that

$$\boxed{P_1\ op\ P_2 \,\|\, - \,\|\, s} \longmapsto^{k_0+1} \boxed{- \,\|\, \underline{n} \,\|\, s}$$

and so

$$\boxed{P_2 : P_1 :\ op \,\|\, - \,\|\, s} \longmapsto^{k_0} \boxed{- \,\|\, \underline{n} \,\|\, s}.$$

Using Lemmas 3.3.4 and 3.3.5 we have, noting $P_2 :: \mathsf{int}$, that

$$\begin{array}{rcl} \boxed{P_2 \,\|\, - \,\|\, s} & \longmapsto^{k_1} & \boxed{- \,\|\, \underline{n_2} \,\|\, s} \\ \boxed{P_1 :\ op \,\|\, \underline{n_2} \,\|\, s} & \longmapsto^{k_2} & \boxed{- \,\|\, \underline{n} \,\|\, s} \end{array}$$

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

$$\begin{array}{rcl} \boxed{P_1 \,\|\, \underline{n_2} \,\|\, s} & \longmapsto^{k_{21}} & \boxed{- \,\|\, \underline{n_1} : \underline{n_2} \,\|\, s} \\ \boxed{op \,\|\, \underline{n_1} : \underline{n_2} \,\|\, s} & \longmapsto^{k_{22}} & \boxed{- \,\|\, \underline{n} \,\|\, s} \end{array} \qquad (1)$$

where $k_{21} + k_{22} = k_2$. So as $k_1 \le k_0$, by Induction we deduce that $(P_2\,,s) \Downarrow (\underline{n_2}\,,s)$, and from Lemma 3.3.5 that

$$\boxed{P_1 \,\|\, - \,\|\, s} \longmapsto^{k_{21}} \boxed{- \,\|\, \underline{n_1} \,\|\, s}.$$

Also, as $k_{21} \le k_0$, we have Inductively that $(P_1\,,s) \Downarrow (\underline{n_1}\,,s)$ and hence

$$(P_1\ op\ P_2\,,s) \Downarrow (\underline{n_1\ op\ n_2}\,,s).$$

But from Lemma 3.3.2 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,

$$\boxed{\text{for all } P :: \mathsf{cmd}, s, s' \quad \boxed{P \,\|\, - \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, S \,\|\, s'} \quad \textit{implies} \quad S = - \ \textit{and}\ (P\,,s) \Downarrow (\mathsf{skip}\,,s')}$$

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

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

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

for all $P :: \mathsf{cmd}, s, s'$,

$$\boxed{P \,\|\, - \,\|\, s} \longmapsto^{k} \boxed{- \,\|\, S \,\|\, s'} \quad \textit{implies} \quad S = - \quad \textit{and} \quad (P\,,s) \Downarrow (\mathsf{skip}\,,s')$$

for all $k \leq k_0$, then

for all $P :: \mathsf{cmd}, s, s'$,

$$\boxed{P \,\|\, - \,\|\, s} \longmapsto^{k_0+1} \boxed{- \,\|\, S \,\|\, s'} \quad \textit{implies} \quad S = - \quad \textit{and} \quad (P\,,s) \Downarrow (\mathsf{skip}\,,s')$$

Pick arbitrary $P :: \mathsf{cmd}$ and S and s, s' and suppose that

$$\boxed{P \,\|\, - \,\|\, s} \longmapsto^{k_0+1} \boxed{- \,\|\, S \,\|\, s'}$$

We consider cases for P:

(*Case* P is $l{:=}P$): Using Lemma 3.3.2, we must have

$$\boxed{l{:=}P \,\|\, - \,\|\, s} \longmapsto^{1} \boxed{P : \mathsf{STO}(l) \,\|\, - \,\|\, s} \longmapsto^{k_0} \boxed{- \,\|\, S \,\|\, s'}$$

and so by Lemmas 3.3.4 and 3.3.5 (and the typing rules)

$$\begin{aligned} \boxed{P \,\|\, - \,\|\, s} &\longmapsto^{k_1} \boxed{- \,\|\, \underline{c} \,\|\, s} \\ \boxed{\mathsf{STO}(l) \,\|\, \underline{c} \,\|\, s} &\longmapsto^{k_2} \boxed{- \,\|\, S \,\|\, s'} \qquad (1) \end{aligned}$$

where $k_1 + k_2 = k_0$. By determinism for (1) we have $S = -$ and $s\{l \mapsto c\} = s'$. By the first right to left implication for integer expressions (proved above) we have $(P\,,s) \Downarrow (\underline{c}\,,s)$. Hence $(l{:=}P\,,s) \Downarrow (\mathsf{skip}\,,s\{l \mapsto c\})$, and as $s\{l \mapsto c\} = s'$ we are done. NB this case did not make use of the inductive hypotheses $\phi(k)_{k \leq k_0}$!

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

The remaining cases are omitted. □

3.5 Executions (of Interpreted Code)

Examples 3.5.1

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

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

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

$$
\begin{aligned}
\boxed{\mathsf{if}\ l \geq \underline{0}\ \mathsf{then}\ l\!:=\!l-\underline{1}\ \mathsf{else\ skip} \parallel - \parallel s} \quad &\longmapsto \quad \boxed{l \geq \underline{0} : \mathsf{BR}(l\!:=\!l-\underline{1}, \mathsf{skip}) \parallel - \parallel s} \\
&\longmapsto \quad \boxed{\underline{0} : x \geq : \mathsf{BR}(l\!:=\!l-\underline{1}, \mathsf{skip}) \parallel - \parallel s} \\
&\longmapsto \quad \boxed{l : \geq : \mathsf{BR}(l\!:=\!l-\underline{1}, \mathsf{skip}) \parallel \underline{0} \parallel s} \\
&\longmapsto \quad \boxed{\geq : \mathsf{BR}(l\!:=\!l-\underline{1}, \mathsf{skip}) \parallel \underline{1} : \underline{0} \parallel s} \\
&\longmapsto \quad \boxed{\mathsf{BR}(l\!:=\!l-\underline{1}, \mathsf{skip}) \parallel \underline{T} \parallel s} \\
&\longmapsto \quad \boxed{l\!:=\!l-\underline{1} \parallel - \parallel s} \\
&\longmapsto \quad \boxed{l-\underline{1} : \mathsf{STO}(l) \parallel - \parallel s} \\
&\longmapsto \quad \boxed{\underline{1} : x : - : \mathsf{STO}(l) \parallel - \parallel s} \\
&\longmapsto \quad \boxed{l : - : \mathsf{STO}(l) \parallel \underline{1} \parallel s} \\
&\longmapsto \quad \boxed{- : \mathsf{STO}(l) \parallel \underline{1} : \underline{1} \parallel s} \\
&\longmapsto \quad \boxed{\mathsf{STO}(l) \parallel \underline{0} \parallel s} \\
&\longmapsto \quad \boxed{- \parallel - \parallel s\{l \mapsto 0\}}
\end{aligned}
$$

4

Operational Semantics for a Functional Language $\mathbb{FUN}^e$

4.1 Introduction

Motivation 4.1.1 In this chapter we turn our attention to (pure) functional programming languages. Such languages provide a syntax of expressions in which one can write down functions directly, 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. In this chapter we shall study the syntax and type system of a simple functional programming language. Before we begin the details, let us look at some examples. Figure 4.1 gives an example of an *identifier environment*. This gives the types of various constant and function identifiers. Figure 4.2 declares the meanings of the identifiers. Most of this should be clear, but we give a few explanatory comments. If σ is any types then $[\sigma]$ is the type of σ-lists. The empty list is written as nil. If E_2 denotes a list of type $[\sigma]$, and E_1 is of type σ, then $E_1 : E_2$ is the list whose head is E_1 and tail is E_2. Expressions of type $\sigma_1 \to \sigma_2$ are functions with input type σ_1 and output type σ_2. If $E_1 :: \sigma_1 \to \sigma_2$ and $E_2 :: \sigma_1$ then we write $E_1\, E_2$ for the application of the function E_1 to the input E_2.

4.2 Types and Expressions

Motivation 4.2.1 We begin by defining the types and expressions of a simple language called $\mathbb{FUN}^e$. 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 4.2.2 The types of the language $\mathbb{FUN}^e$ are (the syntax trees) given inductively (exercise: what are the rules?) by the grammar

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

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

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

for

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

```
cst :: Int
f :: Int -> Int
g :: Int -> Int -> Int
h :: Int -> Int -> Int -> Int
empty_list :: [Int]
l1 :: [Int]
l2 :: [Int]
h :: Int
t :: Int
length :: [Bool] -> Int
map :: (Int -> Bool) -> [Int] -> [Bool]
```

Note that function types associate to the right. Thus

```
     Int -> Int -> Int          abbreviates   Int -> (Int -> Int)
     Int -> Int -> Int -> Int   abbreviates   Int -> (Int -> (Int -> Int))
```

Figure 4.1: An example of an Identifier Environment

```
cst = 76                            -- definition of constant cst
f x = x
g x y = x+y
h x y z = x+y+z                     -- definition of function identifier h
l1 = 5:(6:(8:(4:(nil))))            -- a list
l2 = 5:6:8:4:nil                    -- the same list
h  = hd (5:6:8:4:nil)               -- head of list
t  = tl (5:6:8:4:nil)               -- tail of list
length l = if elist(l) then 0 else (1 + length t)
map f l  = if elist(l) then nil else (f h) : (map f t)
```

Note that function application associates to the left—thus `g x y` is sugar for `(g x) y` and `h x y z` is sugar for `((h x) y) z`.
The function `length` calculates the length of a list `l` of Booleans, and `map` applies a function `f :: Int -> Bool` to each element of a list `l`. Note also that `l1 = l2`.

Figure 4.2: An example of an Identifier Declaration

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 **identifiers**, with typical elements ranged over by metavariables I, F and K. These symbols will be used to define constants and higher order functions in $\mathbb{FUN}^e$—compare

$Fxy = x+y$ in $\mathbb{FUN}^e$ to `f x y = x+y` in Haskell.

The sugared **expressions** of the functional language $\mathbb{FUN}^e$ are given inductively by the grammar

E	$::=$	x	variables
	$\mid$	$\underline{c}$	integer or Boolean constant
	$\mid$	K	constant identifier
	$\mid$	F	function identifier
	$\mid$	$E_1\ iop\ E_2$	integer valued operator on integers
	$\mid$	$E_1\ bop\ E_2$	Boolean valued operator on integers
	$\mid$	$\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3$	conditional
	$\mid$	$E_1\ E_2$	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
	$\mid$	$\mathsf{elist}(E)$	Boolean test for empty list

Examples 4.2.3

(1) $[(\mathsf{bool},\mathsf{bool}) \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}]$ is sugar for $[(\mathsf{bool},\mathsf{bool}) \to (\mathsf{int} \to (\mathsf{int} \to \mathsf{int}))]$

(2) $((\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}) \to \mathsf{bool}$ is sugar for

$$((\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})) \to ((\mathsf{int} \to \mathsf{int}) \to \mathsf{bool})$$

Remark 4.2.4 We shall adopt a few conventions to make expressions more readable:

• In general, we shall write our "formal" syntax in an informal manner, using brackets "(" and ")" to disambiguate where appropriate—recall that in Pascal and Haskell 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 $E_1\ (E_2\ E_3)$.

• $E_1E_2E_3 \dots E_n$ is shorthand for $(\dots((E_1E_2)E_3)\dots)\ E_n$. We say that application **associates** to the left. For example, $E_1E_2E_3$ 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, $\mathsf{ap}(E,E')$ where ap is a tree constructor, then $(E_1\ E_2)\ E_3$ would be a shorthand notation for the tree denoted by $\mathsf{ap}(\mathsf{ap}(E_1,E_2),E_3)$. Exercise: make sure you understand why function types associate to the right, and function applications to the left.

• 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 $\underline{1} : \underline{4} : \underline{6} : \mathsf{nil}_{\mathsf{int}}$).

• Exercise: Try writing out each of the general expression forms as finite trees, using tree constructors such as cons for the cons operation.

Definitions 4.2.5 The variable x *occurs* in the expression $x\ op\ \underline{3}\ op\ x$. In fact, it occurs twice. For an expression E and a variable v we shall assume that it is clear what v **occurs** in E means. We do not give a formal definition. We shall also talk of the identifiers which occur (or appear) in E.

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 *simultaneously* replacing x_i for each $1 \le i \le n$. (We omit the proof that the finite tree $E[E_1,\dots,E_n/x_1,\dots,x_n]$ is indeed an expression).

Examples 4.2.6 Examples of expressions and substitutions are

(1) x

(2) $\mathsf{hd}(\underline{2} : \underline{4} : \mathsf{nil}_{\mathsf{int}})$

(3) $f\ (g\,y)$ where $f,g,y \in Var$.

(4) $b : \underline{T} : \underline{F} : \mathsf{nil}_{\mathsf{bool}}$

(5) $F\underline{2}\,\underline{3}$

(6) $(xy : \underline{2} : z : \mathsf{nil})[F,\underline{2}+\underline{3},z/x,y,z] = F(\underline{2}+\underline{3}) : \underline{2} : z : \mathsf{nil}$

(7) $(x+y+z)[y,x/x,y][\underline{3},y/x,y] = y+\underline{3}+z$

(8) $(x+y+z)[x/y][u,\underline{4}/x,z] = u+u+\underline{4}$

(9) Exercise: work out $(x+y+z)[z+1,4/x,z]$ and then $((x+y+z)[z+1/x])[4/z]$. What happens? Did you expect this?

(10) $F\ \underline{2}\ (\underline{4} * \underline{-7})$ is sugar for $(F\ \underline{2})\ (\underline{4} * \underline{-7})$

(11) $G\ (F\ (\underline{2} * \underline{-7})\ (\underline{4} * \underline{-7}), (\underline{3} * \underline{6} + \underline{5}) \le \underline{6})$ is sugar for

$$G\ ((F\ (\underline{2} * \underline{-7}))\ (\underline{4} * \underline{-7}), ((\underline{3} * \underline{6}) + \underline{5}) \le \underline{6})$$

Motivation 4.2.7 In this chapter, we will build programs out of identifiers and variables, and in order to help construct sensible programs we shall first assign types to such identifiers and variables, much as we assigned types to locations in $\mathbb{IMP}$.

Definitions 4.2.8 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 $\Gamma = \{\,(x_1,\sigma_1),\ldots,(x_n,\sigma_n)\,\}$. We usually write a typical pair (x,σ) as $x :: \sigma$, and a typical context as

$$\Gamma = x_1 :: \sigma_1,\ldots,x_n :: \sigma_n.$$

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 $\Gamma,\Gamma' \overset{\text{def}}{=} \Gamma \cup \Gamma'$ and $\Gamma, x :: \sigma \overset{\text{def}}{=} \Gamma \cup \{x :: \sigma\}$.

An **identifier type** is a type of the form $\sigma_1 \to \sigma_2 \to \sigma_3 \to \ldots \to \sigma_a \to \sigma$ where a is a natural number and σ is **NOT a function type**. If a is 0 then the type is simply σ. You should think of such an identifier type as typing information for an identifier. If $a = 0$ then the identifier is called a **constant**. If $a > 0$ then the identifier is called a **function**; the identifier will represent a function that takes a *maximum* of a inputs with types σ_i and gives an output of type σ. We call a the **arity** of the identifier. Of course the identifier can take less than a inputs to yield another function—compare `(h 4) 5` in Figure 4.2 where h has $a = 3$. We shall denote identifier types by the Greek letter ι.

An **identifier environment** is specified by a finite set of (identifier, identifier type) pairs, with a typical identifier environment being denoted by

$$I = I_1 :: \iota_1,\ldots,I_m :: \iota_m.$$

We say that ι_i is the identifier type of I_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, and use similar conventions for identifier environments.

Example 4.2.9 A simple example of an identifier environment is

$$I \overset{\text{def}}{=} \mathsf{map} :: (\mathsf{int} \to \mathsf{int}) \to [\mathsf{int}] \to [\mathsf{int}],\ \ \mathsf{suc} :: \mathsf{int} \to \mathsf{int}$$

Note that $(\mathsf{int} \to \mathsf{int}) \to [\mathsf{int}] \to [\mathsf{int}]$ is the identifier type of map. Another simple example of an identifier environment is $\mathsf{plus} :: (\mathsf{int},\mathsf{int}) \to \mathsf{int}$.

Motivation 4.2.10 Given a context Γ of typed variables, and an identifier environment I, we can build up expressions E which use only variables and identifiers which appear in Γ and I. This is how we usually write (functional) programs: we first declare constants and types, possibly also functions and types, and then write our program E which uses these data. We shall define judgements of the form $\Gamma \vdash E :: \sigma$ which should be understood as follows: given an identifier environment I, and a context Γ, then the expression E is well formed and has type σ. Given I and Γ, we say that E is **assigned** the type σ. We call $\Gamma \vdash E :: \sigma$ a **type assignment** relation.

Definitions 4.2.11 Given any identifier environment I, we shall inductively define a type assignment (ternary) relation which takes the form $\Gamma \vdash E :: \sigma$ using the rules in Table 4.1.

Proposition 4.2.12 Given an identifier environment I, a context Γ and an expression E, if there is a type σ for which $\Gamma \vdash E :: \sigma$, then such a type is unique. Thus $\mathbb{FUN}^e$ is **monomorphic**.

Proof Omitted. Exercise. □

Examples 4.2.13

(1) We give a deduction of the following type assignment $x :: \mathsf{bool} \vdash G\ (\underline{2} \le \underline{6}, x) :: \mathsf{bool}$ where $G :: (\mathsf{bool},\mathsf{bool}) \to \mathsf{bool}$.

$$\dfrac{\mathcal{D} \qquad \dfrac{\dfrac{\dfrac{}{x :: \mathsf{bool} \vdash \underline{2} :: \mathsf{int}} \qquad \dfrac{}{x :: \mathsf{bool} \vdash \underline{6} :: \mathsf{int}}}{x :: \mathsf{bool} \vdash \underline{2} \le \underline{6} :: \mathsf{bool}} \qquad \dfrac{}{x :: \mathsf{bool} \vdash x :: \mathsf{bool}}}{x :: \mathsf{bool} \vdash (\underline{2} \le \underline{6}, x) :: (\mathsf{bool},\mathsf{bool})}}{x :: \mathsf{bool} \vdash G\ (\underline{2} \le \underline{6}, x) :: \mathsf{bool}}$$

where $\mathcal{D}$ is

$$\dfrac{}{x :: \mathsf{bool} \vdash G :: (\mathsf{bool},\mathsf{bool}) \to \mathsf{bool}}$$

(2) With I as in Example 4.2.9, we have

$$x :: \mathsf{int}, y :: \mathsf{int}, z :: \mathsf{int} \vdash \mathsf{map}\,\mathsf{suc}\,(x : y : z : \mathsf{nil}_{\mathsf{int}}) :: [\mathsf{int}]$$

(3) Let I be $\mathsf{twicehead} :: [\mathsf{int}] \to \mathsf{int} \to (\mathsf{int},\mathsf{int})$. Then we have

$$y :: [\mathsf{int}], x :: \mathsf{int} \vdash \mathsf{twicehead}\ y\ x :: (\mathsf{int},\mathsf{int})$$

(4) We have

$$\varnothing \vdash \mathsf{if}\ \underline{T}\ \mathsf{then}\ \mathsf{fst}((\underline{2} : \mathsf{nil}_{\mathsf{int}}, \mathsf{nil}_{\mathsf{int}}))\ \mathsf{else}\ (\underline{2} : \underline{6} : \mathsf{nil}_{\mathsf{int}}) :: [\mathsf{int}]$$

4.3 Function Declarations and Programs

Motivation 4.3.1 An *identifier declaration* is a method for declaring that identifiers have certain meanings. We look at two examples:

We begin by specifying an identifier environment, such as $\mathsf{plus} :: (\mathsf{int},\mathsf{int}) \to \mathsf{int}$ or $\mathsf{fac} :: \mathsf{int} \to \mathsf{int}$ or $\mathsf{b} :: \mathsf{bool}$. Then to declare that plus is a function which takes a pair of integers

$$\frac{}{\Gamma \vdash x :: \sigma}\ (\text{ where } x :: \sigma \in \Gamma) \quad :: \textsc{var} \qquad \frac{}{\Gamma \vdash \underline{n} :: \mathsf{int}} :: \textsc{int}$$

$$\frac{}{\Gamma \vdash \underline{T} :: \mathsf{bool}} :: \textsc{true} \qquad \frac{}{\Gamma \vdash \underline{F} :: \mathsf{bool}} :: \textsc{false}$$

$$\frac{\Gamma \vdash E_1 :: \mathsf{int} \quad \Gamma \vdash E_2 :: \mathsf{int}}{\Gamma \vdash E_1\ iop\ E_2 :: \mathsf{int}} :: \textsc{op}_1$$

$$\frac{\Gamma \vdash E_1 :: \mathsf{int} \quad \Gamma \vdash E_2 :: \mathsf{int}}{\Gamma \vdash E_1\ bop\ E_2 :: \mathsf{bool}} :: \textsc{op}_2$$

$$\frac{\Gamma \vdash E_1 :: \mathsf{bool} \quad \Gamma \vdash E_2 :: \sigma \quad \Gamma \vdash E_3 :: \sigma}{\Gamma \vdash \mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3 :: \sigma} :: \textsc{cond}$$

$$\frac{\Gamma \vdash E_1 :: \sigma_2 \to \sigma_1 \quad \Gamma \vdash E_2 :: \sigma_2}{\Gamma \vdash E_1\ E_2 :: \sigma_1} :: \textsc{ap}$$

$$\frac{}{\Gamma \vdash I :: \iota}\ (\text{ where } I :: \iota \in I) \quad :: \textsc{idr}$$

$$\frac{}{\Gamma \vdash \mathsf{nil}_\sigma :: [\sigma]} :: \textsc{nil} \qquad \frac{\Gamma \vdash E_1 :: \sigma \quad \Gamma \vdash E_2 :: [\sigma]}{\Gamma \vdash E_1 : E_2 :: [\sigma]} :: \textsc{cons}$$

$$\frac{\Gamma \vdash E :: [\sigma]}{\Gamma \vdash \mathsf{hd}(E) :: \sigma} :: \textsc{hd} \qquad \frac{\Gamma \vdash E :: [\sigma]}{\Gamma \vdash \mathsf{tl}(E) :: [\sigma]} :: \textsc{tl} \qquad \frac{\Gamma \vdash E :: [\sigma]}{\Gamma \vdash \mathsf{elist}(E) :: \mathsf{bool}} :: \textsc{elist}$$

Table 4.1: Type Assignment Relation $\Gamma \vdash E :: \sigma$ in $\mathbb{FUN}^e$

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

$$\mathsf{fac}\,x = \mathsf{if}\ x == \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x * \mathsf{fac}(x - \underline{1})$$

And to declare that b denotes $\underline{T}$ we write $\mathsf{b} = \underline{T}$.

Thus in general, if F is a function identifier, we might write $Fx = E$ 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 $Fx = E$ and $K = E'$ which are regarded as preliminary data to writing a program—we *declare* the definitions of certain functions and constants. The language is then able to provide the user with identifiers 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 Haskell.

In general, an identifier declaration will specify the action of a finite number of function identifiers, and moreover the definitions can be mutually recursive—each identifier may be defined in terms of itself or indeed *the others*. Note that the factorial function given above is defined recursively: the identifier fac actually appears in the expression which gives “the result” of the function.

A *program* in $\mathbb{FUN}^e$ is an expression in which there are no variables and each of the 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 often 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. Functions without arguments are also values. We now make all of these ideas precise.

Definitions 4.3.2 Let $I = I_1 :: \iota_1, \ldots, I_m :: \iota_m$ be a given, fixed, identifier environment for which

$$\iota_j = \sigma_{j1} \to \sigma_{j2} \to \sigma_{j3} \to \ldots \to \sigma_{ja_j} \to \sigma_j. \quad (j \in \{1, \ldots, m\})$$

Then an **identifier declaration** dec_I consists of the following data:

$$\begin{array}{rcl} I_1\, x_{11} \ldots x_{1a_1} & = & E_{I_1} \\ I_2\, x_{21} \ldots x_{2a_2} & = & E_{I_2} \\ & \vdots & \\ I_j\, x_{j1} \ldots x_{ja_j} & = & E_{I_j} \\ & \vdots & \\ I_m\, x_{m1} \ldots x_{ma_m} & = & E_{I_m} \end{array}$$

We define a **program expression** P to be any expression for which no variables occur in P. A **program** in $\mathbb{FUN}^e$ is a judgement of the form

$$dec_I \quad in\ P$$

where dec_I is a given identifier declaration and the program expression P satisfies a type assignment of the form

$$\varnothing \vdash P :: \sigma \qquad (P :: \sigma)$$

and the declarations in dec_I satisfy

$$\begin{gathered} x_{11} :: \sigma_{11}, \ldots, x_{1a_1} :: \sigma_{1a_1} \vdash E_{I_1} :: \sigma_1 \\ x_{21} :: \sigma_{21}, \ldots, x_{2a_2} :: \sigma_{2a_2} \vdash E_{I_2} :: \sigma_2 \\ \vdots \\ x_{j1} :: \sigma_{j1}, \ldots, x_{ja_j} :: \sigma_{ja_j} \vdash E_{I_j} :: \sigma_j \\ \vdots \\ x_{m1} :: \sigma_{m1}, \ldots, x_{ma_m} :: \sigma_{ma_m} \vdash E_{I_m} :: \sigma_m \end{gathered}$$

Note that the data which are *specified* in dec_I just consist of the declarations $I_j\vec{x} = E_{I_j}$; the type assignments just need to hold of the specified E_{I_j}. In practice, such type checking will be taken care of by the compiler. We shall sometimes abbreviate the jth type assignment to $\Gamma_{I_j} \vdash E_{I_j} :: \sigma_j$. We call the expression E_{I_j} the **definitional body** of I_j. Note that the type assignments force each of the variables in $\{x_{j1}, \ldots, x_{ja_j}\}$ to be *distinct* (for each $j \in \{1, \ldots, m\}$). 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 $dec_I \;\; in \;\; P$ (and sometimes just say σ is the type of P).

Examples 4.3.3

(1) Let $I = I_1 :: [\mathsf{int}] \to \mathsf{int} \to \mathsf{int}, I_2 :: \mathsf{int} \to \mathsf{int}, I_3 :: \mathsf{bool}$. Then an example of an identifier declaration dec_I is

$$\begin{aligned} I_1 x_{11} x_{12} &= \mathsf{hd}(\mathsf{tl}(\mathsf{tl}(x_{11}))) + I_2 x_{12} \\ I_2 x_{21} &= x_{21} * x_{21} \\ I_3 &= \underline{T} \end{aligned}$$

Note that here we labelled the variables with subscripts to match the general definition of identifier declaration—in future we will not bother to do this. It is easy to see that the declaration is well defined: for example $x_{21} :: \mathsf{int} \vdash x_{21} * x_{21} :: \mathsf{int}$.

(2) Let I be $F :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$. Then we have a declaration dec_I

$$F\, x\, y\, z = x + y + z$$

where of course $x :: \mathsf{int}, y :: \mathsf{int}, z :: \mathsf{int} \vdash x + y + z :: \mathsf{int}$.

(3) The next few examples are all programs

$$F\, x = \mathsf{if}\ x \le \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x * F\ (x - \underline{1}) \quad in\ \ F\ \underline{4}$$

(4)

$$\left.\begin{array}{lcl} F_1\,x\,y\,z & = & \mathsf{if}\ x \leq \underline{1}\ \mathsf{then}\ y\ \mathsf{else}\ z \\ F_2\,x & = & F_1\,x\,\underline{1}\,(x * F_2\,(x - \underline{1})) \end{array}\right\}\quad \mathit{in}\ F_2\,\underline{4}$$

(5)

$$\left.\begin{array}{lcl} \mathsf{Ev}\,x & = & \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{T}\ \mathsf{else}\ \mathsf{Od}\,(x - \underline{1}) \\ \mathsf{Od}\,x & = & \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{F}\ \mathsf{else}\ \mathsf{Ev}\,(x - \underline{1}) \end{array}\right\}\quad \mathit{in}\ \mathsf{Ev}\,\underline{12}$$

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) $F\ x = F\ x \quad in\ \ F\ (\underline{3} : \mathsf{nil}_{\mathsf{int}})$ is a program which does not evaluate to a value; the program *loops*—see Chapter 4.

4.4 An Eager Evaluation Relation

Motivation 4.4.1 The operational semantics of $\mathbb{FUN}^e$ gives rules for proving that a program P evaluates to a value V within a given identifier declaration dec_I. For any given identifier declaration, we write this as $P \Downarrow^e V$, and a trivial example is, say, $\underline{3}+\underline{4}+\underline{10} \Downarrow^e \underline{17}$ or $\mathsf{hd}(\underline{2} : \mathsf{nil}_{\mathsf{int}}) \Downarrow^e \underline{2}$.

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:

Let $F\,x\,y = x+y$. We would expect $F\ (\underline{2}*\underline{3})\ (\underline{4}*\underline{5}) \Downarrow^e \underline{26}$. But how do we reach this value? The first possibility is *call-by-value* evaluation. First we calculate the first argument to get $F\ \underline{6}\ (\underline{4}*\underline{5})$, then the second to get $F\ \underline{6}\ \underline{20}$. Having got the *values* of the function *arguments*, we *call* the function to get $\underline{6}+\underline{20}$, which evaluates to $\underline{26}$.

In evaluating a function application FP_1P_2 we first compute values for P_1 and P_2, say V_1 and V_2, and then evaluate FV_1V_2. In evaluating a list we compute values for each element of the list, before the list itself is passed to a function.

Definitions 4.4.2 Let dec_I be a identifier declaration. A **value expression** is any expression V which can be produced by the following grammar

$$V ::= \underline{c} \mid \mathsf{nil}_\sigma \mid V : V \mid F\,\vec{V}$$

where c is any Boolean or integer, σ is any type, and $\vec{V}$ abbreviates $V_1\ V_2\ \ldots\ V_{k-1}\ V_k$ where $0 \leq k < a$, and a is the maximum number of inputs taken by F. A **value** is any value expression for which $dec_I \quad in \ V$ is a valid $\mathbb{FUN}^e$ program. Note that constants K are *not* values. Note also that k is *strictly* less than a, and that if $a = 1$ then $F\,\vec{V}$ denotes F.

Definitions 4.4.3 We shall define an **evaluation relation** whose judgements will take the form

$$P \Downarrow^e V$$

where P and V are respectively a program expression and value expression whose function identifiers appear in an identifier declaration dec_I. The rules for inductively generating these judgements are given by the rules in Table 4.2.

Remark 4.4.4 You may find the definition of $F\vec{V}$ 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 $F\vec{V}$ is indeed a value expression; a small example will clarify:

Suppose that

$$F :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int},$$

and that P_1 and P_2 are integer programs, which compute to the values $\underline{n_1}$ and $\underline{n_2}$. Then $F\,P_1$ is not a value, because the language is eager. It will evaluate to $F\,\underline{n_1}$. 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 $F\,\underline{n_1}$ is a value. Giving it the argument P_2, we have a program $F\,\underline{n_1}\,P_2$ which evaluates to the value $F\,\underline{n_1}\,\underline{n_2}$. Again, we have a value, as the expression cannot be computed any further. Finally, however, we can supply a third argument to $F\,\underline{n_1}\,\underline{n_2}$ giving $F\,\underline{n_1}\,\underline{n_2}\,P_3$. This evaluates to $F\,\underline{n_1}\,\underline{n_2}\,\underline{n_3}$, 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 $\Downarrow^e$ FID.

Examples 4.4.5

(1) Let $I \stackrel{\text{def}}{=} G :: \mathsf{int} \to \mathsf{int}, K :: \mathsf{int}$ be an identifier environment. Suppose also that dec_I is

$$\begin{array}{rcl} G\,x & = & x * \underline{2} \\ K & = & \underline{3} \end{array}$$

We prove that $G\,K \Downarrow^e \underline{6}$. To do this, we produce a deduction tree. First note that the program being evaluated is an application, that G is a value, but K is not a value. So the rule used in the final deduction step *must* be rule AP, hence we need to show that $G \Downarrow^e F\,\vec{V}$ (easy: take F to be G with the $\vec{V}$ "empty"), that $K \Downarrow^e V$ for some V which is easy, as we can guess that V must be $\underline{3}$, and that $G\,V \Downarrow^e \underline{6}$. The latter must be a conclusion of

$$\frac{}{V \Downarrow^e V} \Downarrow^e\text{VAL} \qquad \frac{P_1 \Downarrow^e \underline{m} \quad P_2 \Downarrow^e \underline{n}}{P_1\ op\ P_2 \Downarrow^e \underline{m\ op\ n}} \Downarrow^e\text{OP}$$

$$\frac{P_1 \Downarrow^e \underline{T} \quad P_2 \Downarrow^e V}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V} \Downarrow^e\text{COND}_1$$

$$\frac{P_1 \Downarrow^e \underline{F} \quad P_3 \Downarrow^e V}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V} \Downarrow^e\text{COND}_2$$

$$\frac{\left\{\begin{array}{l} P_1 \Downarrow^e F\,\vec{V} \quad P_2 \Downarrow^e V_2 \quad F\,\vec{V}\,V_2 \Downarrow^e V \\ \text{where either } P_1 \text{ or } P_2 \text{ is not a value}\end{array}\right.}{P_1\,P_2 \Downarrow^e V} \Downarrow^e\text{AP}$$

$$\frac{E_F[V_1,\ldots,V_a/x_1,\ldots,x_a] \Downarrow^e V}{FV_1\ldots V_a \Downarrow^e V} [F\vec{x} = E_F \text{ declared in } dec_I] \ \Downarrow^e\text{FID}$$

$$\frac{E_K \Downarrow^e V}{K \Downarrow^e V} [K = E_K \text{ declared in } dec_I] \ \Downarrow^e\text{CID}$$

$$\frac{P \Downarrow^e V : V'}{\mathsf{hd}(P) \Downarrow^e V} \Downarrow^e\text{HD} \qquad \frac{P \Downarrow^e V : V'}{\mathsf{tl}(P) \Downarrow^e V'} \Downarrow^e\text{TL}$$

$$\frac{P_1 \Downarrow^e V \quad P_2 \Downarrow^e V'}{P_1 : P_2 \Downarrow^e V : V'} \Downarrow^e\text{CONS}$$

$$\frac{P \Downarrow^e \mathsf{nil}_\sigma}{\mathsf{elist}(P) \Downarrow^e \underline{T}} \Downarrow^e\text{ELIST}_1 \qquad \frac{P \Downarrow^e V : V'}{\mathsf{elist}(P) \Downarrow^e \underline{F}} \Downarrow^e\text{ELIST}_2$$

Table 4.2: Evaluation Relation $P \Downarrow^e V$ in $\mathbb{FUN}^e$

FID and so we now need to show that $(x*\underline{2})[V/x] \Downarrow^e \underline{6}$. This is also easy following from OP. Putting this altogether we get

$$\dfrac{\dfrac{}{G \Downarrow^e G}\,\text{VAL} \qquad \dfrac{\dfrac{}{\underline{3}\Downarrow^e \underline{3}}\,\text{VAL}}{K\Downarrow^e \underline{3}}\,\text{CID} \qquad \dfrac{\dfrac{\dfrac{}{\underline{3}\Downarrow^e\underline{3}}\,\text{VAL} \qquad \dfrac{}{\underline{2}\Downarrow^e\underline{2}}\,\text{VAL}}{(x*\underline{2})[\underline{3}/x]=\underline{3}*\underline{2}\Downarrow^e\underline{6}}\,\text{OP}}{G\,\underline{3}\Downarrow^e\underline{6}}\,\text{FID}}{G\,K\Downarrow^e\underline{6}}\,\text{AP}$$

(2) Suppose that

$$F :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \qquad \text{where} \qquad F\,x\,y\,z = x+y+z$$

Then the expressions $F\,\underline{2}$ and $F\,\underline{2}\,\underline{3}$ are (programs and) values. $F\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1})$ is a program, but not a value: the function F takes a maximum of three inputs, and can now be evaluated. Note that $F\,\underline{2}\,\underline{3}$ is sugar for $(F\,\underline{2})\,\underline{3}$ and that $F\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1})$ is sugar for the expression $((F\,\underline{2})\,\underline{3})\,(\underline{4}+\underline{1})$. In Definitions 4.4.2, $a=3$, and in $F\underline{2}\underline{3}$ we have $\vec{V}=\underline{2}\underline{3}$ and $l=2<3$.

We can prove that

$$F\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1}) \Downarrow^e \underline{10}$$

where $F\,x\,y\,z = x+y+z$ as follows:

$$\dfrac{\dfrac{}{F\,\underline{2}\,\underline{3}\Downarrow^e F\,\underline{2}\,\underline{3}}\Downarrow^e\text{VAL} \qquad \dfrac{\dfrac{}{\underline{4}\Downarrow^e\underline{4}} \qquad \dfrac{}{\underline{1}\Downarrow^e\underline{1}}}{\underline{4}+\underline{1}\Downarrow^e\underline{5}} \qquad T}{F\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1})\Downarrow^e\underline{10}}\Downarrow^e\text{AP}$$

where T is the tree

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{2}\Downarrow^e\underline{2}} \qquad \dfrac{}{\underline{3}\Downarrow^e\underline{3}}}{\underline{2}+\underline{3}\Downarrow^e\underline{5}} \qquad \dfrac{}{\underline{5}\Downarrow^e\underline{5}}}{\underline{2}+\underline{3}+\underline{5}\Downarrow^e\underline{10}}}{(x+y+z)[\underline{2},\underline{3},\underline{5}/x,y,z]\Downarrow^e\underline{10}}}{F\,\underline{2}\,\underline{3}\,\underline{5}\Downarrow^e\underline{10}}\Downarrow^e\text{FID}$$

It is an exercise to fill in the missing labels on the rules, and missing brackets.

(3) Let $G\,x = x+2$.

Prove that $\mathsf{hd}(G\,\underline{3} : \mathsf{nil}) \Downarrow^e \underline{5}$. To do this, we derive a deduction tree:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{nil} \Downarrow^e \mathsf{nil}}\,\text{VAL} \qquad \dfrac{\dfrac{\dfrac{}{\underline{3} \Downarrow^e \underline{3}}\,\text{VAL} \qquad \dfrac{}{\underline{2} \Downarrow^e \underline{2}}\,\text{VAL}}{(x+\underline{2})[\underline{3}/x] = \underline{3}+\underline{2} \Downarrow^e \underline{5}}\,\text{OP}}{G\,\underline{3} \Downarrow^e \underline{5}}\,\text{FID}}{G\,\underline{3} : \mathsf{nil} \Downarrow^e \underline{5} : \mathsf{nil}}\,\text{CONS}}{\mathsf{hd}(G\,\underline{3} : \mathsf{nil}) \Downarrow^e \underline{5}}\,\text{HD}$$

Motivation 4.4.6 We shall now prove two results about the language $\mathbb{FUN}^e$. The first is that evaluation is deterministic: when we evaluate a program, if this results in a value, that value is unique. We shall also prove that the type of the value is the same as that of the original program.

Theorem 4.4.7 Let dec_I be a identifier 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

$$P \Downarrow^e V_1 \; \textit{and} \; P \Downarrow^e V_2$$

then $V_1 = V_2$.

Proof We prove by Rule Induction that

$$\forall P \Downarrow^e V_1. \quad \boxed{\forall V_2. \, (P \Downarrow^e V_2 \; \textit{implies} \; V_1 = V_2)}$$

This is an exercise. □

Motivation 4.4.8 We shall now show that if a program is evaluated, then the resulting value has the same type as the original program. This is stated precisely in Theorem 4.4.10. However, the proof will require the following lemma in order to deal with the rule $\Downarrow^e$ FID.

Lemma 4.4.9 Suppose that we have $\Gamma, x_1 :: \sigma_1, \ldots, x_n :: \sigma_n \vdash E :: \sigma$ and that $\Gamma \vdash P_i :: \sigma_i$ for $i \in \{1, \ldots, n\}$. Then $\Gamma \vdash E[P_1, \ldots, P_n/x_1, \ldots, x_n] :: \sigma$.

Proof Essentially the proof is a Rule Induction on the derivation of the type assignment for E. However, to set things up properly, we have to jiggle the data around.

Let us pick arbitrary type assignments $\Gamma \vdash P_i :: \sigma_i$ where $i \in \{1, \ldots, n\}$, and pick arbitrary $x_1, \ldots, x_n$.

We then show by Rule Induction that

$$\forall \Delta \vdash E :: \sigma. \quad \boxed{(\Delta = \Gamma, x_1 :: \sigma_1, \ldots, x_n :: \sigma_n) \textit{ implies } \Gamma \vdash E[P_1, \ldots, P_n / x_1, \ldots, x_n] :: \sigma}$$

where Δ denotes a context.

We look at property closure for the (base) rule

$$\frac{}{\Delta \vdash x :: \sigma}\ (\text{ where } x :: \sigma \in \Delta) \qquad :: \textsc{var}$$

Suppose that $\Delta = \Gamma, x_1 :: \sigma_1, \ldots, x_n :: \sigma_n$. Then either $x :: \sigma \in \Gamma$, or $x :: \sigma$ is equal to one of the $x_i :: \sigma_i$. In the first case,

$$x[P_1, \ldots, P_n / x_1, \ldots, x_n] = x$$

But certainly $\Gamma \vdash x :: \sigma$ by assumption! In the second case, $x[P_1, \ldots, P_n / x_1, \ldots, x_n] = P_i$ as $x = x_i$ from the assumption. But certainly $\Gamma \vdash P_i :: \sigma$ as $\sigma = \sigma_i$. This completes the work for rule $::$ VAR.

The remaining property closures are left as exercises. □

Theorem 4.4.10 Evaluating a program $dec_I \;\; in \;\, P$ does not alter its type. More precisely,

$$(\varnothing \vdash P :: \sigma \textit{ and } P \Downarrow^e V) \quad \textit{implies} \quad \varnothing \vdash V :: \sigma$$

for any P, V, σ and I. The conservation of type during program evaluation is called **subject reduction**.

Proof We prove by Rule Induction that

$$\forall P \Downarrow^e V. \quad \boxed{\forall \sigma (\varnothing \vdash P :: \sigma \quad \textit{implies} \quad \varnothing \vdash V :: \sigma).}$$

The proof is an exercise. □

5

A Compiled SECD Machine

5.1 Why Introduce the Machine?

Motivation 5.1.1 We have seen how to define an evaluation relation $\Downarrow^e$ for the language $\mathbb{FUN}^e$. If in fact $P \Downarrow^e V$, how do we effectively compute V from P?

We seek a formal execution mechanism which can take a program P, and mechanically produce the value V of P:

$$P \equiv P_0 \mapsto P_1 \mapsto P_2 \mapsto \ldots \mapsto V$$

Now, "mechanically produce" can be made precise by saying that we require a relation $P \mapsto P'$ between programs, which is defined by a set of rules in which there are *no hypotheses*. Such rules are called **re-writes** (see Chapter 2).

An evaluation semantics, $\Downarrow^e$, is very much an opposite to the notion of a re-write relation $\mapsto$. To show that $P \Downarrow^e V$ requires a "large" proof search for a deduction tree, and completely suppresses any notion of "mechanistic evaluation" of P to V. However, $\Downarrow^e$ is more useful for proving general properties of programs. We illustrate these ideas in Figure 5.1.

We will define a "machine", an *SECD machine*, which will "mechanically compute" certain programs to values, using re-write rules. Landin invented the original SECD machine. It was developed as an interpreter for a programming language based upon lambda terms and function applications. SECD machines can be implemented directly on silicon. The original evaluation strategy was eager. The machine described here is based upon the original machine, but has been developed by the author for the direct execution of the $\mathbb{FUN}^e$ language, and the presentation is entirely original.

In this chapter we shall show how to perform such mechanical computations for a fragment of the language $\mathbb{FUN}^e$. The expressions of this language fragment are given by the (very restricted) grammar

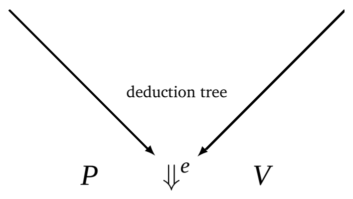
$$E ::= x \mid \underline{n} \mid F \mid E\,E$$

and the definition of program is just as in Chapter 4 but using this restricted set of expressions. The reason for making this restriction is simply to illustrate the SECD machine, without being cluttered by too many computation rules which deal with the various kinds of program which normally appear in $\mathbb{FUN}^e$.

$\ggg$ **NOTE 5.1.2** ***The SECD machine has an environment which maps variables to expressions. This is slightly different from our previous use of "typing" environments. As the SECD terminology is established, we stick with it.***

$$P_0 \mapsto P_1 \mapsto P_2 \mapsto P_3 \mapsto P_4 \ldots \mapsto V$$

Re-Write Rules (Abstract Machine)



Evaluation Semantics

Figure 5.1: Illustrating Two Kinds of Operational Semantics

5.2 Architecture of the Machine

In order to define the SECD machine, we first need a few preliminary definitions. The SECD machine consists of rules for transforming **SECD configurations**. It has a typical configuration (S,E,C,D) consisting of four components:

The **stack** S is a (possibly empty) finitely branching tree whose nodes are constants or closures. The empty stack is denoted by $-$. Non-empty stacks are generated by the following informal grammar where $n \in \mathbb{N}$ and $a \geq l \geq 0$ where a is the arity of F

$$S ::= \begin{array}{c} \underline{n} \\ \uparrow \end{array} \mid \begin{array}{c} S_l \ldots S_1 \\ clo_F \\ \uparrow \end{array}$$

Each node has a **level.** $\underline{n}$ is at level 1; if a node is at level α in any S_i then that node is at level $\alpha+1$ in any

$$\begin{array}{c} S_l \ldots S_i \ldots S_1 \\ clo_F \\ \uparrow \end{array}$$

Given any stack S, we can identify the finite set of nodes at level $\alpha \in \mathbb{N}$, which may be empty. The set of nodes at level 0 is $\varnothing$ by definition. A stack S has a **height** which is given by the maximum level at which a clo_F node occurs, and is otherwise 0. For any stack S of height h at least 1 and natural number $h \geq \alpha \geq 1$, there is a (unique) left-most closure node clo_F at level α. We call this the α-**prescribed** node, and may write **.**clo_F to indicate the occurrence within a stack; we also write $\alpha \quad S$ to indicate that S has an α-prescribed node. Given such an $\alpha \quad S$ for which in general there is a sub-stack of shape

$$\begin{array}{c} S_l \ \dots \ S_1 \\ \bullet clo_F \\ \uparrow \end{array}$$

and given any other stack S_{l+1}, then there is a stack $S_{l+1} \oplus S$ which "extends S with the stack S_{l+1}" in the sense that the α-prescribed node now looks like

$$\begin{array}{c} S_{l+1} \ S_l \ \dots \ S_1 \\ \bullet clo_F \\ \uparrow \end{array}$$

Finally, we shall sometimes write ${}_{\mathsf{Av}}\alpha \quad S$ to indicate that S is of a certain form, which we call an **application value**. The predicate ${}_{\mathsf{Av}}$ is called the **status** of the stack. Application values are any stacks different from $\overset{n}{\uparrow}$ and $\overset{clo_F}{\uparrow}$; roughly speaking they are machine representations of values of the form $F\,\vec{V}$ where $\vec{V}$ is non-empty. It is an exercise to draw some examples of such stacks (finitely branching trees) and to use them to understand the definitions.

The **environment** E takes the form $x_1 =?S_1 : \dots : x_n =?S_n$, essentially meaning that the variables $x_1,\dots,x_n$ currently have the values $S_1,\dots,S_n$ respectively. The environment may be empty $-$. The value of each ? is determined by the form of an S_i. If S_i is $\overset{n}{\uparrow}$ then ? is 0; if S_i is $\overset{clo_F}{\uparrow}$ then ? is 1; in any other case, ? is ${}_{\mathsf{Av}}1$.

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

$$\begin{array}{rcl} ins & ::= & x \mid \underline{n} \mid F \mid \mathsf{APP} \\ C & ::= & - \mid ins : C \end{array}$$

where $\underline{n}$ and F are any numbers or identifiers. The objects *ins* are SECD **instructions**. We shall write $-$ to indicate an empty code. We shall also abbreviate $ins : -$ to *ins*.

The **dump** D is either empty $-$ or is another machine configuration (S,E,C,D'). So a typical dump looks like

$$(S_1,E_1,C_1,(S_2,E_2,C_2,\dots(S_n,E_n,C_n,-)\dots))$$

It is essentially a list of triples $(S_1,E_1,C_1),(S_2,E_2,C_2),\dots,(S_n,E_n,C_n)$ and serves as the function call stack—the empty dump has **size** 0, with such a non-empty dump being of size n (the length of the list).

The machine will execute $\mathbb{FUN}^e$ expressions from the restricted grammar $E := x \mid \underline{n} \mid F \mid E_1\ E_2$ where for technical convenience we will assume that F has arity $a \geq 1$. In particular, such expressions will always be considered in the context of a given declaration dec_I. We shall now compile such expressions into SECD codes. We shall assume that any given program expression has already been through the type checking phase of compilation. We shall define a function $[\![-]\!]\colon Exp \to SECDcodes$ which takes an SECD expression and turns it into code.

- $[\![x]\!] \overset{\text{def}}{=} x$
- $[\![\underline{n}]\!] \overset{\text{def}}{=} \underline{n}$
- $[\![F]\!] \overset{\text{def}}{=} F$
- $[\![E_1\, E_2]\!] \overset{\text{def}}{=} [\![E_1]\!] : [\![E_2]\!] : \mathsf{APP}$

Given a function identifier F, then we define $clo_F \overset{\text{def}}{=} \mathsf{CLO}(\vec{x}, [\![E_F]\!])$ where $F\ \vec{x} \overset{\text{def}}{=} E_F$ occurs in the current function declaration. There is then a representation of program values as stacks, given by

- $(\!|\underline{n}|\!) \overset{\text{def}}{=} \begin{array}{c} \underline{n} \\ \uparrow \end{array}$
- $(\!|F\ V_1 \ldots V_k|\!) \overset{\text{def}}{=} \begin{array}{c} (\!|V_k|\!)\ \ldots\ (\!|V_1|\!) \\ clo_F \\ \uparrow \end{array} = (\!|V_k|\!) \oplus \ldots \oplus (\!|V_1|\!) \oplus \begin{array}{c} \bullet clo_F \\ \uparrow \end{array}$

Let us write SECD machine configurations as arrays:

S	Stack, S
E	Environment, E
C	Control, C
D	Dump, D

To evaluate the (restricted) $\mathbb{FUN}^e$ program P, the machine begins execution in the **initial configuration**, where P is in the Control and all other components are empty:

S	$-$
E	$-$
C	P
D	$-$

If the control is non-empty, then its first command triggers a configuration re-write, whereby the SECD machine changes to a new configuration. The re-writes are deterministic, and are determined by the element at the head of the Control list, the status and level of the stack, and the structural form of the stack at the α-prescribed node.

Here are the re-writes; note that in specifying the rules, $a \geq 1$, $a > k \geq 1$, and in the patterns $\ldots S_1$ each $S_i \not\equiv -$; however, S may be empty:

A number is pushed onto the stack (the initial stack can be of any status):

$$\begin{array}{|c||c|}\hline \mathbf{S} & [_{\mathsf{Av}}]\alpha \quad S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & \underline{n} : C \\ \hline \mathbf{D} & D \\ \hline \end{array} \xmapsto{\text{num}} \begin{array}{|c||c|}\hline \mathbf{S} & \alpha \quad \begin{array}{c}\underline{n} \\ \uparrow\end{array} \quad \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

A function is pushed onto the stack (the initial stack can be of any status):

$$\begin{array}{|c||c|}\hline \mathbf{S} & [_{\mathsf{Av}}]\alpha \quad S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & F : C \\ \hline \mathbf{D} & D \\ \hline \end{array} \xmapsto{\text{fn}} \begin{array}{|c||c|}\hline \mathbf{S} & \alpha+1 \quad \begin{array}{c}\mathbf{.}clo_F \\ \uparrow\end{array} \quad \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

A variable's value is pushed onto the stack, provided that the environment E contains $x = ?T \equiv [_{\mathsf{Av}}]\delta \quad T$ (where δ is 0 or 1). Note that by definition, the status of T determines the status of the re-written stack, irrespective of the status of the initial stack:

$$\begin{array}{|c||c|}\hline \mathbf{S} & [_{\mathsf{Av}}]\alpha \quad S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & x : C \\ \hline \mathbf{D} & D \\ \hline \end{array} \xmapsto{\text{var}} \begin{array}{|c||c|}\hline \mathbf{S} & [_{\mathsf{Av}}]\delta+\alpha \quad T \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

An application command creates an application value, type 0:

$$\begin{array}{|c||c|}\hline \mathbf{S} & \alpha \quad \begin{array}{c} S_k \ldots S_1 \\ \mathbf{.}clo_F \\ \uparrow\end{array} \quad \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & \mathsf{APP} : C \\ \hline \mathbf{D} & D \\ \hline \end{array} \xmapsto{\text{cav0}} \begin{array}{|c||c|}\hline \mathbf{S} & {}_{\mathsf{Av}}\alpha \quad \begin{array}{c} S_k \ldots S_1 \\ \mathbf{.}clo_F \\ \uparrow\end{array} \quad \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

An application command creates an application value, type 1:

$$\begin{array}{|c||c|}\hline \mathbf{S} & \alpha \quad \begin{array}{ll} \mathbf{.}clo_H & \\ \uparrow & S_{k-1} \ldots S_1 \\ & clo_F \\ & \uparrow \end{array} \quad \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & \mathsf{APP} : C \\ \hline \mathbf{D} & D \\ \hline \end{array} \xmapsto{\text{cav1}} \begin{array}{|c||c|}\hline \mathbf{S} & {}_{\mathsf{Av}}\alpha-1 \quad \begin{array}{ll} clo_H & \\ \uparrow & S_{k-1} \ldots S_1 \\ & \mathbf{.}clo_F \\ & \uparrow \end{array} \quad \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

An application command produces an application value from an application value:

S	$_{\mathsf{Av}}\alpha \quad \begin{array}{cc} S_k \ldots S_1 & \\ \bullet clo_F & S'_{k'-1} \ldots S'_1 \\ \uparrow & \\ clo_G & \\ \uparrow & \end{array} \quad \oplus S$
E	E
C	$\mathsf{APP} : C$
D	D

$\xmapsto{\text{avtav}}$

S	$_{\mathsf{Av}}\alpha - 1 \quad \begin{array}{cc} S_k \ldots S_1 & \\ clo_F & S'_{k'-1} \ldots S'_1 \\ \uparrow & \\ \bullet clo_G & \\ \uparrow & \end{array} \quad \oplus S$
E	E
C	C
D	D

An application command calls a function, type 0:

S	$\alpha \quad \begin{array}{c} S_a \ldots S_1 \\ \bullet clo_F \\ \uparrow \end{array} \quad \oplus S$
E	E
C	$\mathsf{APP} : C$
D	D

$\xmapsto{\text{call0}}$

S	$-$
E	$x_a = ?S_a : \ldots : x_1 = ?S_1 : E$
C	$[\![E_F]\!]$
D	$(\alpha - 1 \quad S, E, C, D)$

An application command calls a function, type 1:

S	$\alpha \quad \begin{array}{cc} \bullet clo_H & S_{a-1} \ldots S_1 \\ \uparrow & \\ clo_F & \\ \uparrow & \end{array} \quad \oplus S$
E	E
C	$\mathsf{APP} : C$
D	D

$\xmapsto{\text{call1}}$

S	$-$
E	$x_a = ?S_a : \ldots : x_1 = ?S_1 : E$
C	$[\![E_F]\!]$
D	$(\alpha - 2 \quad S, E, C, D)$

An application command calls a function, type 2:

S	$_{\mathsf{Av}}\alpha \quad \begin{array}{cc} S_k \ldots S_1 & \\ \bullet clo_F & S'_{a-1} \ldots S'_1 \\ \uparrow & \\ clo_G & \\ \uparrow & \end{array} \quad \oplus S$
E	E
C	$\mathsf{APP} : C$
D	D

$\xmapsto{\text{call2}}$

S	$-$
E	$x_a = ?S'_a : \ldots : x_1 = ?S'_1 : E$
C	$[\![E_G]\!]$
D	$(\alpha - 2 \quad S, E, C, D)$

Restore, where the final status is determined by the initial status:

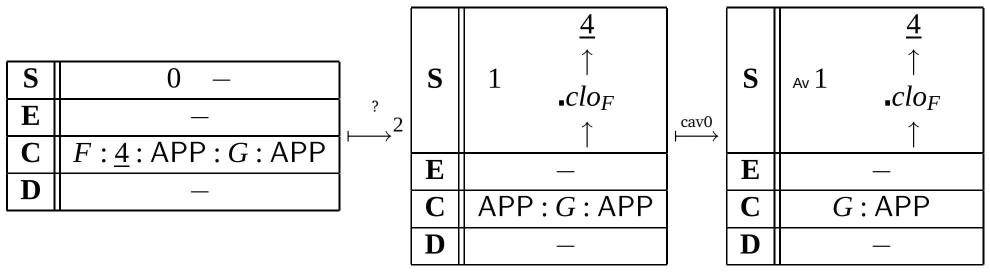
S	$[_{\mathsf{Av}}]\beta \quad T$
E	E'
C	$-$
D	$(\alpha \quad S, E, C, D)$

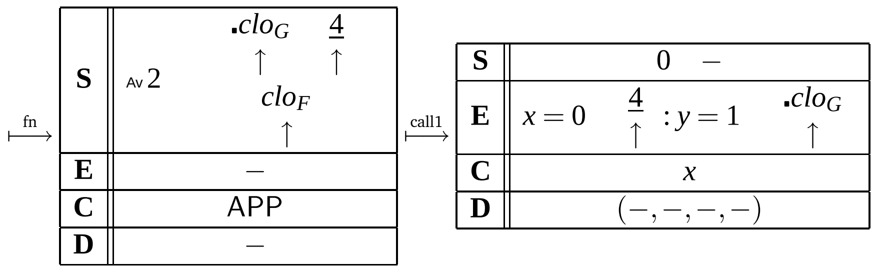
$\xmapsto{\text{res}}$

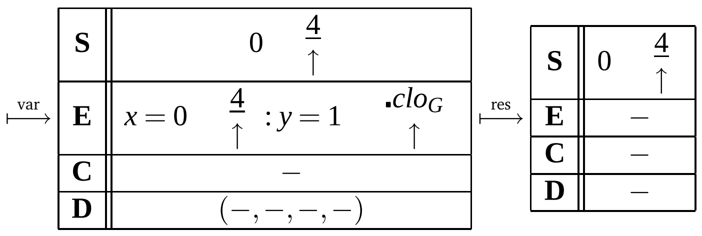
S	$[_{\mathsf{Av}}]\alpha + \beta \quad T \oplus S$
E	E
C	C
D	D

5.3 Executions

1. Take the declaration $F\,x\,y = x$ and $G\,u = u$. Then $[\![F\ \underline{4}\ G]\!] \overset{\text{def}}{=} F : \underline{4} : \mathsf{APP} : G : \mathsf{APP}$. Note that $F\ \underline{4}\ G \Downarrow^e \underline{4}$. It is a very short exercise to fill in the ? re-write labels.







2. Suppose that K, N and MN are functions which are also values, and that

   - $F\,x\,y = x$
   - $L\,u\,v = u$
   - $I\,a\,b = b$
   - $H\,z = L\,(M\,N)\,z$

   Then $(F\ (H\ \underline{4}))\ (I\ \underline{2}\ K) \Downarrow^e M\,N$. Note that

   $[\![(F\ (H\ \underline{4}))\ (I\ \underline{2}\ K)]\!] = (11. \overset{\text{def}}{=} F) : H : \underline{4} : \mathsf{APP} : \mathsf{APP} : I : \underline{2} : \mathsf{APP} : K : \mathsf{APP} : (\mathsf{APP} \overset{\text{def}}{=} 1.)$

and

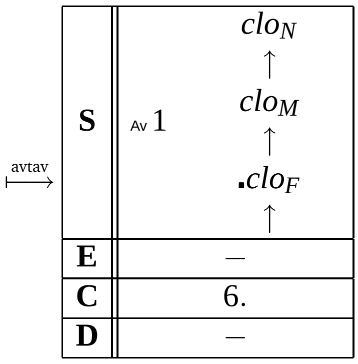
$$[\![L\ (M\ N)\ z]\!] \overset{\text{def}}{=} 7. \overset{\text{def}}{=} L : M : N : \mathsf{APP} : \mathsf{APP} : z : \mathsf{APP} \overset{\text{def}}{=} 1.$$

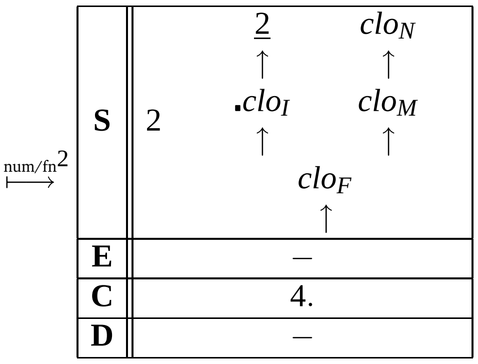
$$\begin{array}{c||c} \mathbf{S} & 0 \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & 11. \\ \hline \mathbf{D} & - \end{array} \xrightarrow{\text{num/fn}}{}^{3} \begin{array}{c||c} \mathbf{S} & 2 \quad \begin{array}{c} \underline{4} \\ \uparrow \\ .clo_H \\ \uparrow \\ clo_F \\ \uparrow \end{array} \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & 8. \\ \hline \mathbf{D} & - \end{array} \xrightarrow{\text{call0}} \begin{array}{c||c} \mathbf{S} & 0 \quad - \\ \hline \mathbf{E} & E' \overset{\text{def}}{=} z = 0 \quad \begin{array}{c} \underline{4} \\ \uparrow \end{array} \\ \hline \mathbf{C} & [\![L\ (M\ N)\ z]\!] \\ \hline \mathbf{D} & \xi \overset{\text{def}}{=} (1 \begin{array}{c} clo_F \\ \uparrow \end{array}, -, 7., -) \end{array}$$

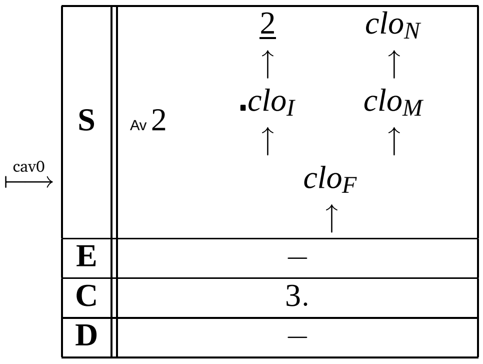
$$\xrightarrow{\text{fn}}{}^{3} \begin{array}{c||c} \mathbf{S} & 3 \quad \begin{array}{c} .clo_N \\ \uparrow \\ clo_M \\ \uparrow \\ clo_L \\ \uparrow \end{array} \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & 4. \\ \hline \mathbf{D} & \xi \end{array} \xrightarrow{\text{cav1}} \begin{array}{c||c} \mathbf{S} & {}_{\mathsf{Av}}2 \quad \begin{array}{c} clo_N \\ \uparrow \\ .clo_M \\ \uparrow \\ clo_L \\ \uparrow \end{array} \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & 3. \\ \hline \mathbf{D} & \xi \end{array} \xrightarrow{\text{avtav}} \begin{array}{c||c} \mathbf{S} & {}_{\mathsf{Av}}1 \quad \begin{array}{c} clo_N \\ \uparrow \\ clo_M \\ \uparrow \\ .clo_L \\ \uparrow \end{array} \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & 2. \\ \hline \mathbf{D} & \xi \end{array}$$

$$\xrightarrow{\text{num}} \begin{array}{c||c} \mathbf{S} & 1 \quad \begin{array}{c} \underline{4} \\ \uparrow \end{array} \quad \begin{array}{c} clo_N \\ \uparrow \\ clo_M \\ \uparrow \\ .clo_L \\ \uparrow \end{array} \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & 1. \\ \hline \mathbf{D} & \xi \end{array} \xrightarrow{\text{call0}} \begin{array}{c||c} \mathbf{S} & 0 \quad - \\ \hline \mathbf{E} & E'' \overset{\text{def}}{=} u = {}_{\mathsf{Av}}1 \quad \begin{array}{c} clo_N \\ \uparrow \\ .clo_M \\ \uparrow \end{array} \quad : v = 0 \quad \begin{array}{c} \underline{4} \\ \uparrow \end{array} \ : E' \\ \hline \mathbf{C} & u \\ \hline \mathbf{D} & \xi' \overset{\text{def}}{=} (-, E', -, \xi) \end{array}$$

$$\xrightarrow{\text{var}} \begin{array}{c||c} \mathbf{S} & {}_{\mathsf{Av}}1 \quad \begin{array}{c} clo_N \\ \uparrow \\ .clo_M \\ \uparrow \end{array} \\ \hline \mathbf{E} & E'' \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & \xi' \end{array} \xrightarrow{\text{res}} \begin{array}{c||c} \mathbf{S} & {}_{\mathsf{Av}}1 \quad \begin{array}{c} clo_N \\ \uparrow \\ .clo_M \\ \uparrow \end{array} \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & \xi \end{array} \xrightarrow{\text{res}} \begin{array}{c||c} \mathbf{S} & {}_{\mathsf{Av}}2 \quad \begin{array}{c} clo_N \\ \uparrow \\ .clo_M \\ \uparrow \\ clo_F \\ \uparrow \end{array} \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & 7. \\ \hline \mathbf{D} & - \end{array}$$







It is an exercise to complete the sequence of re-writes. Also, try the problem again where K is removed from the program (yielding 9 instructions), checking that instruction 3. is now 1.≡ APP and giving rise to call2.

3. Check that the rules are deterministic. Note that this requires great care! The conditions on a, k and the non-emptiness of the S_i are crucial.

6

Correctness of the Compiled SECD Machine

6.1 A Correctness Theorem

Motivation 6.1.1 We shall show that the SECD machine is correct. In more detail, we prove the following theorem.

Theorem 6.1.2 For all programs $dec_I \quad in \;\; P$ for which $\varnothing \vdash P :: \sigma$ we have

$$P \Downarrow^e V \qquad \textit{iff} \qquad \begin{array}{|c||c|}\hline \mathbf{S} & - \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![P]\!] \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^t \begin{array}{|c||c|}\hline \mathbf{S} & (\!|V|\!) \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

6.2 Preliminary Results

Lemma 6.2.1 The SECD machine re-writes are deterministic.

Proof This follows by inspecting the rules. Notice that one needs to examine both the head command and the status and form of the stack. □

Lemma 6.2.2 Given any sequence of SECD 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 stacks, environments, codes, and dumps, if C_1 is non-empty then

$$\begin{array}{|c||c|}\hline \mathbf{S} & S_1 \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^k \begin{array}{|c||c|}\hline \mathbf{S} & S_2 \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_2 \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

implies

$$\begin{array}{|c||c|}\hline \mathbf{S} & S_1 \oplus S_3 \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 : C_3 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^k \begin{array}{|c||c|}\hline \mathbf{S} & S_2 \oplus S_3 \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_2 : C_3 \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

where $\longmapsto^k$ means k re-writes.

Proof We sketch the proof. It bears similarity to Lemma 3.3.3, but requires a slight insight. In fact in order to prove the lemma, we shall prove a slightly *stronger result*, namely that in any case where the dump D is non-empty (say (S',E',C',D')), we can not only extend the stack and code as above (ie the stack and code which occur in the machine's stack and code components), but we can also extend any of the stacks and codes which occur in (S',E',C',D') by any other S_i and C_i we choose. We label an extended D by $\overline{D}$, and if M is an SECD configuration we write $\overline{M}$ for an arbitrary extension of M's stack, code and dump.

We use induction on k. If $k=0$ then there is nothing to prove; extension is trivial. Suppose that the *stronger result* holds for all $k \leq k_0$ where k_0 is arbitrary. We prove that we can extend any re-write of length $M \longmapsto^{k_0+1} M'$. By determinism, we have $M \longmapsto^{1} M'' \longmapsto^{k_0} M'$. There are two cases to consider.

Suppose that the first re-write is not a function call or restore. By examining each of the re-write rules which apply for each case (there are a number of possible rules if the head is APP), we see $M \longmapsto^{1} M''$ can be extended to $\overline{M} \longmapsto^{1} \overline{M''}$. But by induction we have $\overline{M''} \longmapsto^{k_0} \overline{M'}$ and we are done.

Now consider what happens when the head C_1 is $\mathsf{APP} : C$ and there is a function call. We must have

$$M \overset{\text{def}}{=} \begin{array}{|c||c|}\hline \mathbf{S} & T \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & \mathsf{APP} : C \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{1} \begin{array}{|c||c|}\hline \mathbf{S} & - \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & \llbracket E_F \rrbracket \\ \hline \mathbf{D} & (S,E,C,D) \\ \hline \end{array} \longmapsto^{k_1} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E'' \\ \hline \mathbf{C} & C'' \\ \hline \mathbf{D} & (S,E,C,D) \\ \hline \end{array} \overset{\text{res}}{\longmapsto}^{1} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \oplus S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k_2} M'$$

where there are no function calls in the k_2 re-writes (and k_2 might possibly be 0); exercise: why? Note that $k_1 \geq 1$ as $\llbracket E_F \rrbracket$ is non-empty. After the k_1 transitions the dump (S,E,C,D) remains un-changed: This is because the restored dump D must be restored from a dump $(\hat{S},\hat{E},\hat{C},D)$. However, each re-write either does not alter the dump, or a fresh stack, code and environment are appended to the dump. Hence the D resulting from the restore must be restored from (S,E,C,D). By induction, followed by an instance of $_{\text{res}}$, we have

$$\overline{M''} \overset{\text{def}}{=} \begin{array}{|c||c|}\hline \mathbf{S} & - \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & \llbracket E_F \rrbracket \\ \hline \mathbf{D} & (S \oplus S_3, E, C_1 : C_3, \overline{D}) \\ \hline \end{array} \longmapsto^{k_1} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E'' \\ \hline \mathbf{C} & C'' \\ \hline \mathbf{D} & (S \oplus S_3, E, C_1 : C_3, \overline{D}) \\ \hline \end{array} \overset{\text{res}}{\longmapsto}^{1} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \oplus S \oplus S_3 \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 : C_3 \\ \hline \mathbf{D} & \overline{D} \\ \hline \end{array}$$

It is easy to see that $\overline{M} \longmapsto^{1} \overline{M''}$. If $k_2 = 0$ we are done. If $k_2 \geq 1$ then we can similarly extend the stack, code and dump of the final $k_2 \geq 1$ transitions by induction and we are also done. □

Lemma 6.2.3 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 any stacks, environments, codes, and dumps, if C_1 and C_2 are non-empty then

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 : C_2 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

implies

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k_1} \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D \\ \hline \end{array} \quad \textit{and} \quad \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_2 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k_2} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

where $k = k_1 + k_2$ and $k_i \geq 1$.

Proof The proof is similar to that of Lemma 6.2.2. We prove by induction on k that if the hypothesis has k re-writes, then the conclusions follow with k_1 and k_2 re-writes. Let k_0 be arbitrary, and suppose the result holds for all $k \leq k_0$. The proof requires a case analysis on the head of the non-empty C_1. In every case where there is no function call, the $k_0 + 1$ re-writes split as 1 followed by $k_0 \geq 1$ re-writes, and we are done by induction and/or re-write inspection. When there is a function call, we have $S = U \oplus T$ for some U and T, and

$$\begin{array}{|c||c|}\hline \mathbf{S} & U \oplus T \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & \mathsf{APP} : C_1 : C_2 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{1} \begin{array}{|c||c|}\hline \mathbf{S} & - \\ \hline \mathbf{E} & \hat{E} \\ \hline \mathbf{C} & \hat{C} \\ \hline \mathbf{D} & (T, E, C_1 : C_2, D) \\ \hline \end{array} \longmapsto^{k_1} \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & C' \\ \hline \mathbf{D} & (T, E, C_1 : C_2, D) \\ \hline \end{array}$$

$$\begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & C' \\ \hline \mathbf{D} & (T, E, C_1 : C_2, D) \\ \hline \end{array} \longmapsto^{1} \begin{array}{|c||c|}\hline \mathbf{S} & S' \oplus T \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & C_1 : C_2 \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k_2} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D \\ \hline \end{array}$$

The induction hypothesis applies to the final k_2 re-writes. However, notice that for the initial $1 + k_1 + 1$ re-writes, they remain valid if $C_1 : C_2$ is replaced by C_1. The result follows.

$\square$

Lemma 6.2.4 For any well typed $\mathbb{FUN}^e$ program $dec_I \quad in \ P$ where $P :: \sigma$ and $P \Downarrow^e V$,

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & [\![P]\!] \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k} \begin{array}{|c||c|}\hline \mathbf{S} & \hat{S} \\ \hline \mathbf{E} & \hat{E} \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & \hat{D} \\ \hline \end{array} \quad \textit{implies} \quad (\exists \overline{k} \leq k) \begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & [\![V]\!] \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k'} \begin{array}{|c||c|}\hline \mathbf{S} & \hat{S} \\ \hline \mathbf{E} & \hat{E} \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & \hat{D} \\ \hline \end{array}$$

with equality only if P is a value (and hence equal to V).

Proof

Induction on $\Downarrow^e$. For axiom $\Downarrow^e$ VAL the result is trivial. We consider the rule $\Downarrow^e$ AP and leave property closure of $\Downarrow^e$ FID as an exercise. Suppose that

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & [\![P_1]\!] : [\![P_2]\!] : \mathsf{APP} \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k} \begin{array}{|c||c|}\hline \mathbf{S} & \hat{S} \\ \hline \mathbf{E} & \hat{E} \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & \hat{D} \\ \hline \end{array}$$

By Lemma 6.2.3 we have

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & [\![P_1]\!] \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{k_1} \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D' \\ \hline \end{array} \quad \textit{and} \quad \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & [\![P_2]\!] : \mathsf{APP} \\ \hline \mathbf{D} & D' \\ \hline \end{array} \longmapsto^{k_2} \begin{array}{|c||c|}\hline \mathbf{S} & \hat{S} \\ \hline \mathbf{E} & \hat{E} \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & \hat{D} \\ \hline \end{array}$$

Again by Lemma 6.2.3 we have

$$\begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & [\![P_2]\!] \\ \hline \mathbf{D} & D' \\ \hline \end{array} \longmapsto^{k'_1} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E'' \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D'' \\ \hline \end{array} \quad \textit{and} \quad \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E'' \\ \hline \mathbf{C} & \mathsf{APP} \\ \hline \mathbf{D} & D'' \\ \hline \end{array} \longmapsto^{k'_2} \begin{array}{|c||c|}\hline \mathbf{S} & \hat{S} \\ \hline \mathbf{E} & \hat{E} \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & \hat{D} \\ \hline \end{array}$$

So by induction we have

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & [\![F\ \vec{V}]\!] \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{\overline{k_1}} \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D' \\ \hline \end{array} \quad \textit{and} \quad \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & E' \\ \hline \mathbf{C} & [\![V_2]\!] \\ \hline \mathbf{D} & D' \\ \hline \end{array} \longmapsto^{\overline{k'_1}} \begin{array}{|c||c|}\hline \mathbf{S} & S'' \\ \hline \mathbf{E} & E'' \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & D'' \\ \hline \end{array}$$

where $P_1 \Downarrow^e F\ \vec{V}$ and $P_2 \Downarrow^e V_2$

By repeated use of Lemma 6.2.2 (exercise: check) we get

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & [\![F\,\vec{V}]\!] : [\![V_2]\!] : \mathsf{APP} \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{(\overline{k_1}+\overline{k_1'})+k_2'} \begin{array}{|c||c|}\hline \mathbf{S} & \hat{S} \\ \hline \mathbf{E} & \hat{E} \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & \hat{D} \\ \hline \end{array}$$

Again, by induction, we have

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & E \\ \hline \mathbf{C} & [\![V]\!] \\ \hline \mathbf{D} & D \\ \hline \end{array} \longmapsto^{\overline{(\overline{k_1}+\overline{k_1'})+k_2'}} \begin{array}{|c||c|}\hline \mathbf{S} & \hat{S} \\ \hline \mathbf{E} & \hat{E} \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & \hat{D} \\ \hline \end{array}$$

where $F\,\vec{V}V_2 \Downarrow^e V$. It is an exercise to check that $\overline{(\overline{k_1}+\overline{k_1'})+k_2'} \leq k$ (easy!) with equality just in case $P_1\,P_2$ is a value (careful!).

□

6.3 Proving Theorem 6.1.2

Proof $(\Longrightarrow)$: We use Rule Induction for $\Downarrow$. We show property closure for just two example rules:

(*Case* $\Downarrow^e$VAL): For this axiom, we must prove for all V

$$\begin{array}{|c||c|}\hline \mathbf{S} & - \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![V]\!] \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^{t} \begin{array}{|c||c|}\hline \mathbf{S} & (\!|V|\!) \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

We do this by a separate induction on V. If V is $\underline{n}$ or clo_F, this is trivial. By way of illustration, we consider the inductive steps for a value $F\,V_1$; the steps for $F\,\vec{V}$ are similar. By induction, together with Lemma 6.2.2 we have

$$\begin{array}{|c||c|}\hline \mathbf{S} & \begin{array}{c} clo_F \\ \uparrow \end{array} \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![V_1]\!] : \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^{t} \begin{array}{|c||c|}\hline \mathbf{S} & \begin{array}{c} (\!|V_1|\!) \\ clo_F \\ \uparrow \end{array} \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array} \overset{\mathrm{cav}\delta}{\longmapsto} \begin{array}{|c||c|}\hline \mathbf{S} & \begin{array}{c} (\!|V_1|\!) \\ .clo_F \\ \uparrow \end{array} \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

from which the required result follows easily from re-write $\overset{\mathrm{fn}}{\longmapsto}$.

(*Case* $\Downarrow^e$AP): The three induction hypotheses are easy to write down—do this as a simple exercise. Using the first two hypotheses, and Lemma 6.2.2, we get

$$\begin{array}{|c||c|}\hline \mathbf{S} & - \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![P_1]\!] : [\![P_2]\!] : \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^{t} \begin{array}{|c||c|}\hline \mathbf{S} & \begin{array}{c}(\!|V_1|\!)\\ clo_F\\ \uparrow\end{array} \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![P_2]\!] : \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^{t} \begin{array}{|c||c|}\hline \mathbf{S} & \begin{array}{c}(\!|V_2|\!)\ (\!|V_1|\!)\\ clo_F\\ \uparrow\end{array} \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

Now, we can also use Lemma 6.2.2, together with property closure for the axiom (checked above), to show that

$$\begin{array}{|c||c|}\hline \mathbf{S} & - \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & F : [\![V_1]\!] : \mathsf{APP} : [\![V_2]\!] : \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^{t} \begin{array}{|c||c|}\hline \mathbf{S} & \begin{array}{c}(\!|V_2|\!)\ (\!|V_1|\!)\\ clo_F\\ \uparrow\end{array} \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

Thus the result follows by the third induction hypothesis plus determinacy—details are left as an exercise.

($\Longleftarrow$): We shall prove that if $P :: \sigma$ then

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![P]\!] \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^{k} \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array} \qquad \textit{implies} \qquad (\exists V) \quad S' = (\!|V|\!) \oplus S \quad \textit{and} \quad P \Downarrow^e V$$

from which the required result follows. Induction on k. If P is a number or a function the result is trivial. Else P has the form $P_1 P_2$. Suppose that

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![P_1]\!] : [\![P_2]\!] : \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^{k_0+1} \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

Then appealing to Lemma 6.2.3 and the induction hypothesis, we get

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![P_1]\!] \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^{k_1} \begin{array}{|c||c|}\hline \mathbf{S} & (\!|F\,\vec{V}|\!) \oplus S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array} \qquad \textit{and} \qquad \begin{array}{|c||c|}\hline \mathbf{S} & (\!|F\,\vec{V}|\!) \oplus S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![P_2]\!] : \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto^{k_2} \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

where $P_1 \Downarrow^e F\ \vec{V}$. Appealing to Lemma 6.2.3 again, and by induction,

$$\begin{array}{|c||c|}\hline \mathbf{S} & (\!|F\ \vec{V}|\!) \oplus S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![P_2]\!] \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto_{k'_1} \begin{array}{|c||c|}\hline \mathbf{S} & (\!|V_2|\!) \oplus (\!|F\ \vec{V}|\!) \oplus S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

and

$$\begin{array}{|c||c|}\hline \mathbf{S} & (\!|V_2|\!) \oplus (\!|F\ \vec{V}|\!) \oplus S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto_{k'_2} \begin{array}{|c||c|}\hline \mathbf{S} & S' \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & - \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

where $P_2 \Downarrow^e V_2$. By Lemma 6.2.4, and Lemma 6.2.2 we have (check!)

$$\begin{array}{|c||c|}\hline \mathbf{S} & S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & [\![F\ \vec{V}\ V_2]\!] \\ \hline \mathbf{D} & - \\ \hline \end{array} \longmapsto_{\overline{k_1}+\overline{k'_1}} \begin{array}{|c||c|}\hline \mathbf{S} & (\!|V_2|\!) \oplus (\!|F\ \vec{V}|\!) \oplus S \\ \hline \mathbf{E} & - \\ \hline \mathbf{C} & \mathsf{APP} \\ \hline \mathbf{D} & - \\ \hline \end{array}$$

and so $S' = (\!|V|\!) \oplus S$ for some V where $F\ \vec{V}\ V_2 \Downarrow^e V$. Hence $P_1\ P_2 \Downarrow^e V$ as required.

$\square$

A
Mathematical Prerequisites

A.1 Introduction

Definitions A.1.1 We shall adopt a few conventions:

• If we give a definition, the entity being defined will be written in **boldface**; and when we emphasise something it appears in an *italic* typeface.

• Variables will be denoted by notation such as x, x', x'', x_1, x_2, x_3 and so on.

• If we wish to define a set A whose elements are known as widgets, then we shall simply say "let A be the set of **widgets**."

• Suppose we wish to speak of a set A, and indicate that the set A happens to be a subset of a set X. We will write "consider the set $A \subseteq X$ …" for this. For example, we might say "let $O \subseteq \mathbb{N}$ be the set of odd numbers" to emphasise that we are considering the set of odd numbers denoted by O, which happen to be a subset of the natural numbers (denoted by $\mathbb{N}$).

• We often use particular characters for particular purposes. For example, capital letters such as A and X often represent sets, and lower case letters such as a and x represent elements of sets. When you write down Mathematics or Computing, *make sure your lower and upper case letters are clearly distinguishable!*

• We shall often use characters from the Greek alphabet; some of these appear in Table A.1.

• If you read the notes and do not understand something, try reading ahead and looking at examples. You may need to read definitions and look at examples of the definitions simultaneously—each reinforces the other. When you read definitions, try to work out your own simple examples, and see if you can understand the basic ideas behind the technical details. *Many of the examples have details omitted, which you need to fill in using a pencil and paper.*

A.2 Logic

We sometimes 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 $P \vee Q$);

α	alpha
β	beta
γ	gamma (lower case)
Γ	gamma (upper case)
δ	delta (lower case)
Δ	delta (upper case)
ε	epsilon
ι	iota
λ	lambda (lower case)
Λ	lambda (upper case)
ω	omega (lower case)
Ω	omega (upper case)
ρ	rho (lower case)
σ	sigma (lower case)
Σ	sigma (upper case)
θ	theta (lower case)
Θ	theta (upper case)
τ	tau

Figure A.1: Some Greek Characters

• *P implies Q* (sometimes written $P \Rightarrow Q$ or $P \rightarrow Q$);

• *not P* (sometimes written $\neg P$);

• *P if and only if Q* (often written $P \Leftrightarrow Q$ or $P \leftrightarrow Q$ or P *iff* Q)—this is simply an abbreviation for

$$(P\ implies\ Q) \quad and \quad (Q\ implies\ P);$$

• *for all x, P* (sometimes written $\forall x.\ P$ or $(\forall x)\ P$);

• *there exists x, P* (sometimes written $\exists x.\ P$ or $(\forall x)\ P$).

In this course, we shall often prove propositions of the form $\forall x \in X.P(x)$ where $P(x)$ is a is a proposition depending on x, and X is a given set. Then to *prove* (that is, show true) $\forall x \in X.P(x)$, we choose a new variable,[1] say a, and write down a proof of $P(a)$. Providing no assumptions are made about a (we sometimes say a is *arbitrary*) we can conclude that $\forall x \in X.P(x)$ is true. For example, $P(x)$ might be $odd(2*x+1)$. Then to prove $\forall x \in \mathbb{N}.odd(2*x+1)$ we let n denote an arbitrary natural number, and prove $odd(2*n+1)$ (exercise: do it!).

A.3 Sets

Definitions A.3.1 We assume that the idea of a *set* is understood, being an unordered collection of distinct objects. A capital letter such as A or B or X or Y will often be used to denote an arbitrary set. If a is any object in a set A, we say that a is an **element** of A, and write $a \in A$ for this. If a is not an element of A, we write $a \notin A$. The idea of *union* $A \cup B$, *intersection* $A \cap B$, *difference* $A \setminus B$, and *powerset* $\mathcal{P}(A)$ of sets should already be known. We collect the definitions here:

Subset	$S \subseteq A$	$\stackrel{\text{def}}{=}$	$\forall x\,(x \in S\ implies\ x \in A)$
Union	$A \cup B$	$\stackrel{\text{def}}{=}$	$\{\,x \mid x \in A\ or\ x \in B\,\}$
Intersection	$A \cap B$	$\stackrel{\text{def}}{=}$	$\{\,x \mid x \in A\ and\ x \in B\,\}$
Difference	$A \setminus B$	$\stackrel{\text{def}}{=}$	$\{\,x \mid x \in A\ and\ x \notin B\,\}$
Powerset	$\mathcal{P}(A)$	$\stackrel{\text{def}}{=}$	$\{\,S \mid S \subseteq A\,\}$
Finite Powerset	$\mathcal{P}_{fin}(A)$	$\stackrel{\text{def}}{=}$	$\{\,S \mid S \subseteq A\ and\ S$ is finite $\}$

Recall that the **empty** set, $\varnothing$, is the set with no elements. Note that $\varnothing \subseteq A$ for any set A, because $x \in \varnothing$ is always false. So $\varnothing \in \mathcal{P}(A)$. We regard $\varnothing$ as a *finite* set. We shall also use the following sets

[1] NOTE: Sometimes we do not choose a new variable, but work with the original, in this case x. This is okay, providing one remembers the role that the original variable is playing when it is used in the proof.

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

Two sets A and B are **equal**, written $A = B$, if they have the same elements. So, for example, $\{1,2\} = \{2,1\}$. Here, the critical point is whether an object is an element of a set or not: if we write down the elements of a set, it is irrelevant what order they are written down in. But we shall need a way of writing down "a set of objects" in which the order is important.

To see this, think about the map references "1 along and 2 up" and "2 along and 1 up." These two references are certainly different, both involve the numbers 1 and 2, but we cannot use the sets $\{1,2\}$ and $\{2,1\}$ as a mathematical notation for the map references because the sets are equal. Thus we introduce the idea of a pair to model this. If A and B are sets, with $a \in A$ and $b \in B$, we shall write (a,b) for the **pair** of a and b. The crucial property of pairs is that (a,b) and (a',b') are said to be **equal** iff $a = a'$ *and* $b = b'$. We write

$$(a,b) = (a',b')$$

to indicate that the two pairs are indeed equal. We could write $(1,2)$ and $(2,1)$ for our map references. Note that the definition of equality of pairs captures the exact property required of map references. We can also consider n-**tuples** $(a_1,\ldots,a_n)$ and regard such an n-tuple as equal to another n-tuple $(a'_1,\ldots,a'_n)$ iff $a_i = a'_i$ for each $1 \le i \le n$. Note that a pair is a 2-tuple.

The **cartesian product** of A and B, written[2] $A \times B$, is a set given by

$$A \times B \overset{\text{def}}{=} \{\, (a,b) \mid a \in A \textit{ and } b \in B \,\}.$$

For example,

$$\{1,2\} \times \{a,b,c\} = \{\,(1,a),(1,b),(1,c),(2,a),(2,b),(2,c)\,\}.$$

Examples A.3.2

(1) $\{1,2,3\} \cup \{x,y\} = \{1,2,3,x,y\} = \{x,1,y,3,2\} = \ldots$ The written order of the elements is irrelevant.

(2) $\{a,b\} \setminus \{b\} = \{a\}$.

(3) $A \setminus A = \varnothing$.

(4) $\mathcal{P}(\{1,2\}) = \{\{1,2\},\{1\},\{2\},\varnothing\}$.

(5) $\{a\} \times \{b\} = \{(a,b)\}$.

(6) $(x,y) = (2,100)$ *iff* $x = 2$ *and* $y = 100$.

[2]NOTE: In many programming languages, $A \times B$ is denoted by $A * B$ or even (A,B). The latter is used in Haskell.

A.4 Relations

Definitions A.4.1 Given sets A and B, a **relation** R between A and B is a subset $R \subseteq A \times B$. Informally, R is the set whose elements are pairs (a,b) for which "a is in a relationship to b"—see Examples A.4.3. Given a set A, a **binary relation** R on A is a relation between A and itself. So, by definition, R is a subset of $A \times A$.

Remark A.4.2 Note that a *relation* is a set: it is the set of all pairs for which the first element of the pair is in a relationship to the second element of the pair. If $R \subseteq A \times B$ is a relation, it is convenient to write $a\,R\,b$ instead of $(a,b) \in R$. So, for example, *is_the_father_of* is a relation on the set *Humans* of humans, that is

$$\mathit{is\_the\_father\_of} \subseteq \mathit{Humans} \times \mathit{Humans}$$

and if $(\mathrm{Ron},\mathrm{Roy}) \in \mathit{is\_the\_father\_of}$ then we can write instead

$$\mathrm{Ron}\ \mathit{is\_the\_father\_of}\ \mathrm{Roy}.$$

Reading the latter statement corresponds much more closely to common parlance. Note that if $(a,b) \notin R$ then we write $a \not{R}\, b$ for this.

Example A.4.3 Being strictly less than is a binary relation, written $<$, on the natural numbers $\mathbb{N}$. So $< \subseteq \mathbb{N} \times \mathbb{N}$, and

$$< = \{\,(0,1),(0,2),(0,3),(0,4)\ldots,(1,2),(1,3)\ldots,(2,3),\ldots\}.$$

Thus $<$ is the set of pairs (m,n) for which m and n are natural numbers, and m is strictly less than n. Being less than or equal to is also a binary relation on $\mathbb{N}$, written $\leq$. We have

$$\leq = \{\,(0,0),(0,1),(0,2),(0,3),\ldots,(1,1),(1,2),\ldots\}.$$

Definitions A.4.4 We will be interested in binary relations which satisfy certain important properties. Let A be any set and R any binary relation on A. Then

(i) R is **reflexive** iff for all $a \in A$ we have $a\,R\,a$;

(ii) R is **symmetric** iff for all $a,b \in A$, $a\,R\,b$ implies $b\,R\,a$;

(iii) R is **transitive** iff for all $a,b,c \in A$, $a\,R\,b$ and $b\,R\,c$ implies $a\,R\,c$;

(iv) R is **anti-symmetric** iff for all $a,b \in A$, $a\,R\,b$ and $b\,R\,a$ implies $a = b$.

Examples A.4.5 Let $A \stackrel{\text{def}}{=} \{\alpha,\beta,\gamma\}$ be a three element set, and recall the binary relations $<$ and $\leq$ on $\mathbb{N}$ from Example A.4.3.

(1) $R \overset{\text{def}}{=} \{ (\alpha,\alpha),(\beta,\beta),(\gamma,\gamma),(\alpha,\gamma) \}$ is reflexive, but $<$ is not reflexive.

(2) $R \overset{\text{def}}{=} \{ (\alpha,\beta),(\beta,\alpha),(\gamma,\gamma) \}$ is symmetric, but $\leq$ is not.

(3) $R \overset{\text{def}}{=} \{ (\alpha,\beta),(\beta,\gamma),(\alpha,\gamma) \}$ is transitive, as are $<$ and $\leq$.

(4) $R \overset{\text{def}}{=} \{ (\alpha,\beta),(\beta,\gamma),(\alpha,\gamma) \}$ is anti-symmetric. Both $<$ and $\leq$ are anti-symmetric.

(5) Note that R in (1) is also transitive—what other properties hold of the other examples?

Motivation A.4.6 Given any set A, the binary relation of *equality* on A is reflexive, symmetric and transitive. For if $a,b,c \in A$ are any elements of A, $a = a$, if $a = b$ then $b = a$, and if $a = b$ and $b = c$, then $a = c$. An *equivalence relation* is any binary relation which enjoys these three properties. Informally, such a relation can be thought of as behaving like “equality” or “being the same as”. Later, we will use equivalence relations to define a notion of equality between programs; two programs will be related when they have the same execution behaviours, but possibly different codes.

Definitions A.4.7 An **equivalence** relation on a set A, denoted by $\sim$, is any binary relation on A which is reflexive, symmetric and transitive. Given any element a of A, the **equivalence class** of a, denoted by $\overline{a}$, is defined by

$$\overline{a} \overset{\text{def}}{=} \{ a' \mid a' \in A \textit{ and } a \sim a' \}.$$

So the equivalence class of a is the set of all elements of A which are related to a by $\sim$. Note that if $x \in \overline{a}$ then $\overline{x} = \overline{a}$ because $\sim$ is an equivalence relation—check!! We call any element x of $\overline{a}$ a **representative** of $\overline{a}$, because the equivalence class of x equals that of a. We also say that $\overline{a}$ is **represented** by any of its elements; in particular, $\overline{a}$ is represented by a. We shall write $A/\!\sim$ for the set of equivalence classes of elements of A, that is,

$$A/\!\sim \overset{\text{def}}{=} \{ \overline{a} \mid a \in A \}.$$

Example A.4.8 We can define an equivalence relation $\sim$ on the set $\mathbb{Z}$ of integers by setting

$$\forall x \in \mathbb{Z}.\ \forall y \in \mathbb{Z} \qquad x \sim y \quad \textit{iff} \quad x - y \text{ is even.}$$

For example, $3 \sim 5$, $12 \sim 14$, but $100 \not\sim 101$. Examples of equivalence classes are:

$$\overline{1} = \{\ldots,-5,-3,-1,1,3,5,\ldots\} \textit{ and } \overline{4} = \{\ldots,-4,-2,0,2,4,6,8,\ldots\}$$

Examples of representatives of $\overline{1}$ are -997, 31, 1, indeed any integer not divisible by 2. Representatives of $\overline{4}$ are 4, -10000, -8 and so on. Note that $\mathbb{Z}/\!\sim$ is a two element set; for example

$$\mathbb{Z}/\!\sim = \{\overline{1},\overline{2}\} = \{\overline{31},\overline{4}\} = \ldots$$

Definitions A.4.9 Let R be a binary relation on a set A. Then there is a binary relation R^* on A which is defined by

$$\forall a,b \in A. \qquad a\, R^*\, b \overset{\text{def}}{=} \quad \exists a_i \quad (\, a = a_0\, R\, a_1\, R\, a_2 \ldots a_{n-1}\, R\, a_n = b) \qquad (n \geq 0)$$

$$\forall a,b \in A. \qquad a\, R^t\, b \overset{\text{def}}{=} \quad \exists a_i \quad (\, a = a_0\, R\, a_1\, R\, a_2 \ldots a_{n-1}\, R\, a_n = b) \qquad (n \geq 1)$$

We call R^* the **reflexive, transitive closure** of R. Note that if $n = 0$ then $a = b$. We call R^t the **transitive closure** of R.

A.5 Functions

A.5.1 Total Functions

We define the **set of total functions** between sets A and B to be[3]

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

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 $(g \circ f)(a) \overset{\text{def}}{=} g(f(a))$ 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 $id_A(a) \overset{\text{def}}{=} a$ on each $a \in A$.

A.5.2 Partial Functions

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

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

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". The subset of A on which f is defined is called the **domain of definition** of f. If this is finite, say $\{a_1, \ldots, a_n\}$, and $f(a_i) = b_i$, then we sometimes write

$$f = \langle a_1 \mapsto b_1, \ldots, a_n \mapsto b_n \rangle$$

[3]If $\Phi(x)$ is a proposition involving x, then $\exists$ a unique $x.\Phi(x)$ abbreviates

$$(\exists x.\Phi(x)) \ \mathit{and}\ (\forall x,x'.\Phi(x) \ \mathit{and}\ \Phi(x') \ \mathit{implies}\ x = x')$$

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?

B

Abstract Syntax and Rule Induction

B.1 Inductively Defined Sets

B.1.1 Abstract Syntax Trees

Motivation B.1.1 Consider conditional expressions. A typical example is

```
if true then 2 else 3
```

which will be written as a text string in a program file. However, a computer must *work out* that such a string denotes a conditional which is built out of three pieces of data, namely the Boolean and the two numbers. In a real language, it is the job of the compiler to extract such information out of textual strings, usually during the early phases of compilation, namely *lexing* and *parsing*. Crudely speaking the compiler converts the textual (program) string into a *parse tree* which makes this information explicit (see examples below). We shall be looking at simple compilation later on, but for the time being we want to ignore the process of parsing, and *write down programs directly as parse trees*. It would be messy to always draw pictures of such trees—thus we

- develop a simple notation for parse trees, which cuts out the drawing but is awkward to read; and then
- agree on a way to make the notation more readable—we call this syntactic **sugar**.

Let us look at an example where l denotes a list. Here is the readable (sugared) notation:

$$\mathsf{if}\ \ \mathsf{elist}(l)\ \ \mathsf{then}\ \ \underline{0}\ \ \mathsf{else}\ (\ \mathsf{hd}(l)+\mathsf{sum}(\mathsf{tl}(l)))$$

It has the form

$$\mathsf{if}\ B\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2$$

where, for example, B is $\mathsf{elist}(l)$. The conditional (if-then-else) expression requires three arguments, B, E_1 and E_2, and to make this clear it is helpful to write it as

$$\mathsf{cond}(\mathsf{elist}(l)\ ,\ \underline{0}\ ,\ \mathsf{hd}(l)+\mathsf{sum}(\mathsf{tl}(l)))\qquad (*)$$

and think of the conditional as a *constructor* which acts on three arguments, to "construct" a new program (you might like to think of a constructor as a function). Now we look at a sub-part of the program body, $\mathsf{hd}(l)+\mathsf{sum}(\mathsf{tl}(l))$. We can think of $+$ as a constructor which acts on two arguments, and to make this visually clear, it is convenient to write the latter expression as

$$+(\mathsf{hd}(l)\ ,\ \mathsf{sum}(\mathsf{tl}(l))).$$

Finally, looking at one of the sub-parts of this expression, namely $\mathsf{hd}(l)$, we can think of $\mathsf{hd}(l)$ as a constructor hd acting on a single argument, l.

Definitions B.1.2 Let us make this a little clearer. 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 $\mathsf{C}(T_1,T_2,T_3,\ldots,T_n)$ for the finite tree which has the form

$$\begin{tikzcd}
 & & \mathsf{C} \arrow[dll] \arrow[dl] \arrow[d] \arrow[drr] & & \\
T_1 & T_2 & T_3 & \ldots & T_n
\end{tikzcd}$$

Each T_i is itself of the form $\mathsf{C}'(T_1',T_2',T_3',\ldots,T_m')$. We call C a **constructor** and say that C takes n arguments. Any constructor which takes 0 arguments is a **leaf** node. We call C the **root** node of the tree. The roots of the trees T_i are called the **children** of C. The constructors are **labels** for the *nodes* of the tree. Each of the T_i above is a **subtree** of the whole tree—in particular, any leaf node is a subtree. Leaf nodes do not have children. A **proper** subtree T' of a tree T is any subtree T' such that $T' \neq T$.

If we say that cond is a constructor which takes three arguments, $+$ a constructor which takes two arguments, and so on, then $(*)$ denotes the tree

Note that in this (finite) tree, we regard each node as a constructor. To do this, we can think of both l and $\underline{0}$ as constructors which take no arguments!!. These form the *leaves* of the tree. We call the root of the tree the **outermost** constructor, and refer to trees of this kind as **abstract syntax** trees. We often refer to an abstract syntax tree by its outermost constructor—the tree above is a "conditional".

B.1.2 Rule Sets

Definitions B.1.3 Let us first introduce some notation. Consider

statement 1 implies statement 2.

It is sometimes convenient to write this as

$$\frac{\textit{statement 1}}{\textit{statement 2}}$$

Consider

$$\textit{statement 1 iff statement 2.}$$

It is sometimes convenient to write this as

$$\begin{array}{c}\textit{statement 1}\\ \hline\hline \textit{statement 2}\end{array}$$

For example, we can write "$x \leq 4$ *implies* $x \leq 6$" as

$$\frac{x \leq 4}{x \leq 6}$$

The usefulness of this notation will soon become clear.

Motivation B.1.4 We are going to introduce the notion of an *inductively defined set*. Such a set is one whose elements are defined using a special technique known as *induction*. Before we can do this, we need to define things called *rules*. We will give the definitions, and then some examples. We will see that many sets which arise in the description of programming languages can be defined inductively.

Definitions B.1.5 Let us fix a set $\mathcal{U}$. A **rule** R is a pair (H,c) where $H \subseteq \mathcal{U}$ is any finite set, and $c \in \mathcal{U}$ is any 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 $H = \{h_1, \ldots, h_k\}$ where $1 \leq k$. We can write down a base rule $R = (\varnothing, c)$ using the following notation

Base

$$\frac{}{c}\,(R)$$

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

Inductive

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

Given a set $\mathcal{U}$ and a set $\mathcal{R}$ of rules based on $\mathcal{U}$, a **deduction** is a finite tree with nodes labelled by elements of $\mathcal{U}$ such that

• each leaf node label c arises as a base rule $(\varnothing, c) \in \mathcal{R}$

• for any non-leaf node label c, if H is the set of children of c then $(H, c) \in \mathcal{R}$ is an inductive rule.

We then say that the set **inductively defined** by $\mathcal{R}$ consists of those elements $u \in \mathcal{U}$ which have a deduction with root node labelled by u.

Examples B.1.6

(1) Let $\mathcal{U}$ be the set $\{u_1, u_2, u_3, u_4, u_5, u_6\}$ where the u_i are six fixed elements of $\mathcal{U}$, and let $\mathcal{R}$ be the set of rules

$$\{\ R_1 = (\varnothing, u_1), R_2 = (\varnothing, u_3), R_3 = (\{u_1, u_3\}, u_4), R_4 = (\{u_1, u_3, u_4\}, u_5), R_5 = (\{u_2\}, u_4)\ \}$$

Then a deduction for u_5 is given by the tree

which is more normally written up-side down and in the following style

$$\dfrac{\dfrac{}{u_1}\,R_1 \qquad \dfrac{}{u_3}\,R_2 \qquad \dfrac{\dfrac{}{u_1}\,R_1 \qquad \dfrac{}{u_3}\,R_2}{u_4}\,R_3}{u_5}\,R_4$$

(2) A set $\mathcal{R}$ of rules for defining the set $E \subseteq \mathbb{N}$ of even numbers is $\mathcal{R} = \{R_1, R_2\}$ where

$$\dfrac{}{0}\,(R_1) \qquad\qquad \dfrac{e}{e+2}\,(R_2)$$

Note that rule R_2 is, strictly speaking, a rule **schema**, that is e is acting as a variable. There is a "rule" for each instantiation of e. A deduction of 6 is given by

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0}\,(R_1)}{0+2}\,(R_2)}{2+2}\,(R_2)}{4+2}\,(R_2)$$

(3) The set I of integer multiples of 3 can be inductively defined by a set of rules $\mathcal{R} = \{A,B,C\}$ where

$$\frac{-}{0}\,(A) \qquad \frac{i}{i+3}\,(B) \qquad \frac{i}{i-3}\,(C)$$

and informally you should think of the symbol i as a variable, that is, (B) and (C) are rule schemas.

(4) 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 **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[1]

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

Suppose that $\Sigma \stackrel{\text{def}}{=} \{a,b,c\}$. Then a deduction tree for $abac$ is

$$\dfrac{\dfrac{\dfrac{-}{a}\,(1) \qquad \dfrac{-}{b}\,(1)}{ab}\,(2) \qquad \dfrac{\dfrac{-}{a}\,(1) \qquad \dfrac{-}{c}\,(1)}{ac}\,(2)}{abac}\,(2)$$

(5) We can use sets of rules to define the language of propositional logic. Let Var be a set of **propositional variables** with typical elements written P, Q or R. Then the set $Prpn$ of **propositions** of propositional logic is inductively defined by the rules

$$\frac{-}{P}\,[P \in Var]\ (A) \qquad \frac{\phi \quad \psi}{\phi \wedge \psi}\,(\wedge)$$

$$\frac{\phi \quad \psi}{\phi \vee \psi}\,(\vee) \qquad \frac{\phi \quad \psi}{\phi \to \psi}\,(\to) \qquad \frac{\phi}{\neg\phi}\,(\neg)$$

Exercise: Give a deduction for $((P \to Q) \vee (Q \to P)) \wedge R$.

B.2 Principles of Induction

Motivation B.2.1 In this section we see how inductive techniques of proof which the reader has met before fit into the framework of inductively defined sets. We shall write $\phi(x)$ to denote a proposition about x. For example, if $\phi(x) \stackrel{\text{def}}{=} x \geq 2$, then $\phi(3)$ is true and $\phi(0)$ is false. If $\phi(a)$ is *true* then we often say that $\phi(a)$ **holds**.

Definitions B.2.2 We present in Figure B.1 a useful principle called **Rule Induction**. It will be used throughout the course to prove facts about programming languages.

[1]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.

Rule Induction

Let I be inductively defined by a set of rules $\mathcal{R}$. Suppose we wish to show that a proposition $\phi(i)$ holds for all elements $i \in I$, that is, we wish to prove

$$\forall i \in I. \quad \phi(i).$$

Then all we need to do is

• for every base rule $\frac{}{b} \in \mathcal{R}$ prove that $\phi(b)$ holds; and

• for every inductive rule $\frac{h_1 \ldots h_k}{c} \in \mathcal{R}$ prove that whenever $h_i \in I$,

$$(\phi(h_1) \textit{ and } \phi(h_2) \textit{ and } \ldots \textit{ and } \phi(h_k)) \quad \textit{implies} \quad \phi(c)$$

We call the propositions $\phi(h_j)$ **inductive hypotheses**. We refer to carrying out the bulleted (•) tasks as "verifying **property closure**".

Figure B.1: Rule Induction

Motivation B.2.3 The Principle of Mathematical Induction arises as a special case of Rule Induction. We can regard the set $\mathbb{N}$ as inductively defined by the rules

$$\frac{-}{0}\,(\textit{zero}) \qquad \frac{n}{n+1}\,(\textit{add1})$$

Suppose we wish to show that $\phi(n)$ holds for all $n \in \mathbb{N}$, that is $\forall n \in \mathbb{N}.\phi(n)$. According to Rule Induction, we need to verify

• property closure for *zero*, that is $\phi(0)$; and

• property closure for *add1*, that is for every natural number n, $\phi(n)$ implies $\phi(n+1)$, that is $\forall n \in \mathbb{N}.\ (\phi(n) \textit{ implies } \phi(n+1))$

and this amounts to precisely what one needs to verify for Mathematical Induction.

Examples B.2.4

(1) We can define sets of abstract syntax trees inductively. Let us just give an example. Let a set of constructors be $\mathbb{Z} \cup \{+, -, <\}$. The integers will label leaf nodes, and $+$, $-$ and $<$ will take two arguments written with an infix notation. The set of abstract syntax trees $\mathcal{T}$ inductively defined by these constructors is given by

$$\frac{-}{z}\,[z \in Z] \qquad \frac{T_1 \quad T_2}{T_1 + T_2} \qquad \frac{T_1 \quad T_2}{T_1 - T_2} \qquad \frac{T_1 \quad T_2}{T_1 < T_2}$$

Note that the base rules correspond to leaf nodes. The (parse) tree

$$\begin{tikzcd}
 & & < \arrow[dl] \arrow[dr] & \\
 & - \arrow[dl] \arrow[dr] & & 2 \\
55 & & 7 &
\end{tikzcd}$$

is an element of $\mathcal{T}$. In sugared notation it is written $(55-7) < 2$. Show this by giving a deduction tree. **Do not get confused by the fact that the elements of $\mathcal{T}$ are defined by deduction trees—but each element is itself a parse tree!**

$55-7$ is a subtree of $(55-7) < 2$, as are the leaves 55, 7 and 2. If abstract syntax trees are used to describe programming language constructs, we often call them program **expressions**, and refer to **subexpressions**.

Remark B.2.5 You will notice that the BNF **grammar**

$$T ::= n \mid b \mid T+T \mid T-T \mid T<T$$

"defines" the same set of abstract syntax trees (assuming that $+$, $-$ and $<$ are regarded as constructors). In this module we will regard such BNF grammars as short hand for an inductive definition. Given a BNF grammar, there is a corresponding set of rules.

The principle of **structural induction** is defined to be rule induction as applied to syntax trees. Make sure you understand that if $\mathcal{T}$ is an inductively defined set of syntax trees, to prove $\forall T \in \mathcal{T}.\phi(T)$ we have to prove:

- $\phi(L)$ for each leaf node L; and
- assuming $\phi(T_1)$ and ... and $\phi(T_n)$ prove $\phi(C(T_1,\dots,T_n))$ for *each* constructor C and *all* trees $T_i \in \mathcal{T}$.

(2) Let $\Sigma = \{a,b,c\}$ and let a set[2] S of words be defined inductively by the rules

$$\frac{-}{b}\,(1) \qquad \frac{-}{c}\,(2) \qquad \frac{w}{aaw}\,(3) \qquad \frac{w \quad w'}{ww'}\,(4)$$

Suppose that we wish to prove that every word in S has an even number of occurrences of a. Write $\#(w)$ for the number of occurrences of a in w, and

$$\phi(w) \quad \overset{\text{def}}{=} \quad \#(w) \text{ is even.}$$

We prove that $\forall w \in S.\phi(w)$ holds, using Rule Induction; thus we need to verify property closure for each of the rules (1) to (4):

[2]Note that $S \subseteq \Sigma^*$. So any element of S is a word, but there are some words based on the alphabet Σ which are not in S.

(*Rule (1)*): $\#(b) = 0$, even. So $\phi(b)$ holds.

(*Rule (2)*): $\#(c) = 0$, even. So $\phi(c)$ holds.

(*Rule (3)*): Suppose that $w \in S$ is any element and $\phi(w)$ holds, that is $\#(w)$ is even (this is the inductive hypothesis). Then $\#(aaw) = 2 + \#(w)$ which is even, so $\phi(aaw)$ holds.

(*Rule (4)*): Suppose $w, w' \in S$ are any elements and $\#(w)$ and $\#(w')$ are even (these are the inductive hypotheses). Then so too is $\#(ww') = \#(w) + \#(w')$.

Thus by Rule Induction we are done: we have $\forall w \in S.\phi(w)$.

B.3 Recursively Defined Functions

Definitions B.3.1 Let I be inductively defined by a set of rules $\mathcal{R}$, and A any set. A function $f{:}I \to A$ can be defined by

- specifying an element $f(b) \in A$ for every base rule $\frac{}{b} \in \mathcal{R}$; and
- specifying $f(c) \in A$ in terms of $f(h_1) \in A$ and $f(h_2) \in A$ .... and $f(h_k) \in A$ for every inductive rule $\frac{h_1 \ldots, h_k}{c} \in \mathcal{R}$,

provided that each instance of a rule in $\mathcal{R}$ introduces a different element of I—why do we need this condition? When a function is defined in this way, it is said to be **recursively** defined.

Examples B.3.2

(1) The factorial function $F{:}\mathbb{N} \to \mathbb{N}$ is usually defined recursively. We set

- $F(0) \stackrel{\text{def}}{=} 1$ and
- $\forall n \in \mathbb{N}.F(n+1) \stackrel{\text{def}}{=} (n+1) * F(n)$.

Thus $F(3) = (2+1) * F(2) = 3 * 2 * F(1) = 3 * 2 * 1 * F(0) = 3 * 2 * 1 * 1 = 6$. Are there are brackets missing from the previous calculation? If so, insert them.

C

Summary of Rules

$$\frac{}{\underline{n} :: \mathsf{int}}\ [\text{any } n \in \mathbb{Z}] \quad :: \textsc{int} \qquad \frac{}{\underline{T} :: \mathsf{bool}} :: \textsc{true} \qquad \frac{}{\underline{F} :: \mathsf{bool}} :: \textsc{false}$$

$$\frac{}{l :: \mathsf{int}}\ [l :: \mathsf{int} \in \mathcal{L}] \quad :: \textsc{intloc} \qquad \frac{}{l :: \mathsf{bool}}\ [l :: \mathsf{bool} \in \mathcal{L}] \quad :: \textsc{boolloc}$$

$$\frac{P_1 :: \mathsf{int} \quad P_2 :: \mathsf{int}}{P_1\ iop\ P_2 :: \mathsf{int}}\ [\ iop\ \in IOpr] \quad :: \textsc{iop}$$

$$\frac{P_1 :: \mathsf{int} \quad P_2 :: \mathsf{int}}{P_1\ bop\ P_2 :: \mathsf{bool}}\ [\ bop\ \in BOpr] \quad :: \textsc{bop}$$

$$\frac{}{\mathsf{skip} :: \mathsf{cmd}} :: \textsc{skip} \qquad \frac{l :: \sigma \quad P :: \sigma}{l{:=}P :: \mathsf{cmd}}[\sigma \text{ is int or bool}] \quad :: \textsc{ass}$$

$$\frac{P_1 :: \mathsf{cmd} \quad P_2 :: \mathsf{cmd}}{P_1\ ;\ P_2 :: \mathsf{cmd}} :: \textsc{seq}$$

$$\frac{P_1 :: \mathsf{bool} \quad P_2 :: \mathsf{cmd} \quad P_3 :: \mathsf{cmd}}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 :: \mathsf{cmd}} :: \textsc{cond} \qquad \frac{P_1 :: \mathsf{bool} \quad P_2 :: \mathsf{cmd}}{\mathsf{while}\ P_1\ \mathsf{do}\ P_2 :: \mathsf{cmd}} :: \textsc{loop}$$

$$\frac{}{(l\,, s) \leadsto (\underline{s(l)}\,, s)}\ [\text{ provided that } s(l) \text{ is defined }]{\leadsto}\mathrm{LOC}$$

$$\frac{(P_1\,, s) \leadsto (P_2\,, s)}{(P_1\ op\ P\,, s) \leadsto (P_2\ op\ P\,, s)}\ {\leadsto}\mathrm{OP}1$$

$$\frac{(P_1\,, s) \leadsto (P_2\,, s)}{(\underline{n}\ op\ P_1\,, s) \leadsto (\underline{n}\ op\ P_2\,, s)}\ {\leadsto}\mathrm{OP}2 \qquad \frac{}{(\underline{n_1}\ op\ \underline{n_2}\,, s) \leadsto (\underline{n_1\ op\ n_2}\,, s)}\ {\leadsto}\mathrm{OP}3$$

$$\frac{(P_1\,, s) \leadsto (P_2\,, s)}{(l{:=}P_1\,, s) \leadsto (l{:=}P_2\,, s)}\ {\leadsto}\mathrm{ASS}1 \qquad \frac{(P_1\,, s) \leadsto (P_2\,, s)}{(l{:=}P_1\,, s) \leadsto (l{:=}P_2\,, s)}\ {\leadsto}\mathrm{ASS}2$$

$$\frac{}{(l{:=}\underline{c}\,, s) \leadsto (\mathsf{skip}\,, s\{l \mapsto c\})}\ {\leadsto}\mathrm{ASS}3$$

$$\frac{(P_1\,, s_1) \leadsto (P_2\,, s_2)}{(P_1\,;P\,, s_1) \leadsto (P_2\,;P\,, s_2)}\ {\leadsto}\mathrm{SEQ}1 \qquad \frac{}{(\mathsf{skip}\ ;P\,, s) \leadsto (P\,, s)}\ {\leadsto}\mathrm{SEQ}2$$

$$\frac{(P_1\,, s) \leadsto (P_2\,, s)}{(\mathsf{if}\ P_1\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,, s) \leadsto (\mathsf{if}\ P_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,, s)}\ {\leadsto}\mathrm{COND}1$$

$$\frac{}{(\mathsf{if}\ \underline{T}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,, s) \leadsto (P_1\,, s)}\ {\leadsto}\mathrm{COND}2$$

$$\frac{}{(\mathsf{if}\ \underline{F}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,, s) \leadsto (P_2\,, s)}\ {\leadsto}\mathrm{COND}3$$

$$\frac{}{(\mathsf{while}\ P\ \mathsf{do}\ P\,, s) \leadsto (\mathsf{if}\ P\ \mathsf{then}\ (P\ ;\mathsf{while}\ P\ \mathsf{do}\ P)\ \mathsf{else}\ \mathsf{skip}\,, s)}\ {\leadsto}\mathrm{LOOP}$$

$$\frac{}{(l\,, s) \Downarrow (\underline{s(l)}\,, s)}\;[\text{ provided } l \in \text{ domain of } s]\Downarrow\text{LOC} \qquad \frac{}{(\underline{c}\,, s) \Downarrow (\underline{c}\,, s)}\Downarrow\text{CONST}$$

$$\frac{(P_1\,, s) \Downarrow (\underline{n_1}\,, s) \quad (P_2\,, s) \Downarrow (\underline{n_2}\,, s)}{(P_1\ iop\ P_2\,, s) \Downarrow (\underline{n_1\ iop\ n_2}\,, s)}\Downarrow\text{OP}_1$$

$$\frac{(P_1\,, s) \Downarrow (\underline{n_1}\,, s) \quad (P_2\,, s) \Downarrow (\underline{n_2}\,, s)}{(P_1\ bop\ P_2\,, s) \Downarrow (\underline{n_1\ bop\ n_2}\,, s)}\Downarrow\text{OP}_2$$

$$\frac{}{(\mathsf{skip}\,, s) \Downarrow (\mathsf{skip}\,, s)}\Downarrow\text{SKIP}$$

$$\frac{(P\,, s) \Downarrow (\underline{n}\,, s)}{(l := P\,, s) \Downarrow (\mathsf{skip}\,, s\{l \mapsto n\})}\Downarrow\text{ASS}_1 \qquad \frac{(P\,, s) \Downarrow (\underline{b}\,, s)}{(l := P\,, s) \Downarrow (\mathsf{skip}\,, s\{l \mapsto b\})}\Downarrow\text{ASS}_2$$

$$\frac{(P_1\,, s_1) \Downarrow (\mathsf{skip}\,, s_2) \quad (P_2\,, s_2) \Downarrow (\mathsf{skip}\,, s_3)}{(P_1\,;P_2\,, s_1) \Downarrow (\mathsf{skip}\,, s_3)}\Downarrow\text{SEQ}$$

$$\frac{(P\,, s_1) \Downarrow (\underline{T}\,, s_1) \quad (P_1\,, s_1) \Downarrow (\mathsf{skip}\,, s_2)}{(\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,, s_1) \Downarrow (\mathsf{skip}\,, s_2)}\Downarrow\text{COND}_1$$

$$\frac{(P\,, s_1) \Downarrow (\underline{F}\,, s_1) \quad (P_2\,, s_1) \Downarrow (\mathsf{skip}\,, s_2)}{(\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,, s_1) \Downarrow (\mathsf{skip}\,, s_2)}\Downarrow\text{COND}_2$$

$$\frac{(P\,, s_1) \Downarrow (\underline{T}\,, s_1) \quad (P\,, s_1) \Downarrow (\mathsf{skip}\,, s_2) \quad (\mathsf{while}\ P\ \mathsf{do}\ P\,, s_2) \Downarrow (\mathsf{skip}\,, s_3)}{(\mathsf{while}\ P\ \mathsf{do}\ P\,, s_1) \Downarrow (\mathsf{skip}\,, s_3)}\Downarrow\text{LOOP}_1$$

$$\frac{(P\,, s) \Downarrow (\underline{F}\,, s)}{(\mathsf{while}\ P\ \mathsf{do}\ P\,, s) \Downarrow (\mathsf{skip}\,, s)}\Downarrow\text{LOOP}_2$$

$$\frac{}{\Gamma \vdash x :: \sigma} \text{(where } x :: \sigma \in \Gamma) \quad :: \textsc{var} \qquad \frac{}{\Gamma \vdash \underline{n} :: \mathsf{int}} :: \textsc{int}$$

$$\frac{}{\Gamma \vdash \underline{T} :: \mathsf{bool}} :: \textsc{true} \qquad \frac{}{\Gamma \vdash \underline{F} :: \mathsf{bool}} :: \textsc{false}$$

$$\frac{\Gamma \vdash E_1 :: \mathsf{int} \quad \Gamma \vdash E_2 :: \mathsf{int}}{\Gamma \vdash E_1 \; iop \; E_2 :: \mathsf{int}} :: \textsc{op}_1$$

$$\frac{\Gamma \vdash E_1 :: \mathsf{int} \quad \Gamma \vdash E_2 :: \mathsf{int}}{\Gamma \vdash E_1 \; bop \; E_2 :: \mathsf{bool}} :: \textsc{op}_2$$

$$\frac{\Gamma \vdash E_1 :: \mathsf{bool} \quad \Gamma \vdash E_2 :: \sigma \quad \Gamma \vdash E_3 :: \sigma}{\Gamma \vdash \mathsf{if}\; E_1 \;\mathsf{then}\; E_2 \;\mathsf{else}\; E_3 :: \sigma} :: \textsc{cond}$$

$$\frac{\Gamma \vdash E_1 :: \sigma_2 \to \sigma_1 \quad \Gamma \vdash E_2 :: \sigma_2}{\Gamma \vdash E_1 \; E_2 :: \sigma_1} :: \textsc{ap}$$

$$\frac{\Gamma \vdash E_1 :: \sigma_1 \quad \Gamma \vdash E_2 :: \sigma_2}{\Gamma \vdash (E_1, E_2) :: (\sigma_1, \sigma_2)} :: \textsc{pair}$$

$$\frac{\Gamma \vdash E :: (\sigma_1, \sigma_2)}{\Gamma \vdash \mathsf{fst}(E) :: \sigma_1} :: \textsc{fst} \qquad \frac{\Gamma \vdash E :: (\sigma_1, \sigma_2)}{\Gamma \vdash \mathsf{snd}(E) :: \sigma_2} :: \textsc{snd}$$

$$\frac{}{\Gamma \vdash I :: \iota} \text{(where } I :: \iota \in I) \quad :: \textsc{idr}$$

$$\frac{}{\Gamma \vdash \mathsf{nil}_\sigma :: [\sigma]} :: \textsc{nil} \qquad \frac{\Gamma \vdash E_1 :: \sigma \quad \Gamma \vdash E_2 :: [\sigma]}{\Gamma \vdash E_1 : E_2 :: [\sigma]} :: \textsc{cons}$$

$$\frac{\Gamma \vdash E :: [\sigma]}{\Gamma \vdash \mathsf{hd}(E) :: \sigma} :: \textsc{hd} \qquad \frac{\Gamma \vdash E :: [\sigma]}{\Gamma \vdash \mathsf{tl}(E) :: [\sigma]} :: \textsc{tl} \qquad \frac{\Gamma \vdash E :: [\sigma]}{\Gamma \vdash \mathsf{elist}(E) :: \mathsf{bool}} :: \textsc{elist}$$

$$\frac{\Gamma \vdash E_1 :: \sigma \quad \Gamma \vdash E_2[E_1/x] :: \sigma'}{\Gamma \vdash \mathsf{let}\; x = E_1 \;\mathsf{in}\; E_2 :: \sigma'} :: \textsc{let} \qquad \frac{\Gamma, x :: \sigma \vdash E :: \tau}{\Gamma \vdash \mathsf{fn}\, x.E :: \sigma \to \tau} :: \textsc{abs}$$

$$\frac{}{V \Downarrow^e V}\Downarrow^e\text{VAL} \qquad \frac{P_1 \Downarrow^e \underline{m} \quad P_2 \Downarrow^e \underline{n}}{P_1\ op\ P_2 \Downarrow^e \underline{m\ op\ n}}\Downarrow^e\text{OP}$$

$$\frac{P_1 \Downarrow^e \underline{T} \quad P_2 \Downarrow^e V}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V}\Downarrow^e\text{COND1} \qquad \frac{P_1 \Downarrow^e \underline{F} \quad P_3 \Downarrow^e V}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V}\Downarrow^e\text{COND2}$$

$$\frac{P_1 \Downarrow^e V_1 \quad P_2 \Downarrow^e V_2}{(P_1,P_2) \Downarrow^e (V_1,V_2)}\Downarrow^e\text{PAIR}$$

$$\frac{P \Downarrow^e (V_1,V_2)}{\mathsf{fst}(P) \Downarrow^e V_1}\Downarrow^e\text{FST} \qquad \frac{P \Downarrow^e (V_1,V_2)}{\mathsf{snd}(P) \Downarrow^e V_2}\Downarrow^e\text{SND}$$

$$\frac{\left\{\begin{array}{l} P_1 \Downarrow^e F\,\vec{V} \quad P_2 \Downarrow^e V_2 \quad F\,\vec{V}\,V_2 \Downarrow^e V \\ \text{where either } P_1 \text{ or } P_2 \text{ is not a value}\end{array}\right.}{P_1\,P_2 \Downarrow^e V}\Downarrow^e\text{AP}$$

$$\frac{E_F[V_1,\ldots,V_{k_j}/x_1,\ldots,x_k] \Downarrow^e V}{FV_1\ldots V_k \Downarrow^e V}\ [F\vec{x} = E_F \text{ declared in } dec_I]\ \Downarrow^e\text{FID}$$

$$\frac{E_K \Downarrow^e V}{K \Downarrow^e V}\ [K = E_K \text{ declared in } dec_I]\ \Downarrow^e\text{CID}$$

$$\frac{P \Downarrow^e V : V'}{\mathsf{hd}(P) \Downarrow^e V}\Downarrow^e\text{HD} \qquad \frac{P \Downarrow^e V : V'}{\mathsf{tl}(P) \Downarrow^e V'}\Downarrow^e\text{TL}$$

$$\frac{P_1 \Downarrow^e V \quad P_2 \Downarrow^e V'}{P_1 : P_2 \Downarrow^e V : V'}\Downarrow^e\text{CONS}$$

$$\frac{P \Downarrow^e \mathsf{nil}_\sigma}{\mathsf{elist}(P) \Downarrow^e \underline{T}}\Downarrow^e\text{ELIST1} \qquad \frac{P \Downarrow^e V : V'}{\mathsf{elist}(P) \Downarrow^e \underline{F}}\Downarrow^e\text{ELIST2}$$

$$\frac{P_1 \Downarrow^e \mathsf{fn}\,x.E \quad P_2 \Downarrow^e V' \quad E[V'/x] \Downarrow^e V}{P_1\,P_2 \Downarrow^e V}\Downarrow^e\text{AA} \qquad \frac{E_1 \Downarrow^e V_1 \quad E_2[V_1/x] \Downarrow^e V}{\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \Downarrow^e V}\Downarrow^e\text{LET}$$

$$\frac{}{V \Downarrow^l V} \Downarrow^l{\scriptstyle\text{VAL}} \qquad \frac{P_1 \Downarrow^l \underline{m} \quad P_2 \Downarrow^l \underline{n}}{P_1\ op\ P_2 \Downarrow^l \underline{m\ op\ n}} \Downarrow^l{\scriptstyle\text{OP}}$$

$$\frac{P_1 \Downarrow^l \underline{T} \quad P_2 \Downarrow^l V}{\text{if } P_1 \text{ then } P_2 \text{ else } P_3 \Downarrow^l V} \Downarrow^l{\scriptstyle\text{COND1}} \qquad \frac{P_1 \Downarrow^l \underline{F} \quad P_3 \Downarrow^l V}{\text{if } P_1 \text{ then } P_2 \text{ else } P_3 \Downarrow^l V} \Downarrow^l{\scriptstyle\text{COND2}}$$

$$\frac{P \Downarrow^l (P_1, P_2) \quad P_1 \Downarrow^l V}{\text{fst}(P) \Downarrow^l V} \Downarrow^l{\scriptstyle\text{FST}}$$

$$\frac{P \Downarrow^l (P_1, P_2) \quad P_2 \Downarrow^l V}{\text{snd}(P) \Downarrow^l V} \Downarrow^l{\scriptstyle\text{SND}}$$

$$\frac{\begin{cases} P_1 \Downarrow^l F\,\vec{P} \quad F\,\vec{P}\,P_2 \Downarrow^l V \\ \text{where either } P_1 \text{ or } P_2 \text{ is not a value} \end{cases}}{P_1\,P_2 \Downarrow^l V} \Downarrow^l{\scriptstyle\text{AP}}$$

$$\frac{E_F[P_1, \ldots, P_k / x_1, \ldots, x_k] \Downarrow^l V}{F P_1 \ldots P_k \Downarrow^l V} \ [F\vec{x} = E_F \text{ declared in } dec_I] \ \Downarrow^l{\scriptstyle\text{FID}}$$

$$\frac{E_K \Downarrow^l V}{K \Downarrow^l V} \ [K = E_K \text{ declared in } dec_I] \ \Downarrow^l{\scriptstyle\text{CID}}$$

$$\frac{P_1 \Downarrow^l P_2 : P_3 \quad P_2 \Downarrow^l V}{\text{hd}(P_1) \Downarrow^l V} \Downarrow^l{\scriptstyle\text{HD}}$$

$$\frac{P_1 \Downarrow^l P_2 : P_3 \quad P_3 \Downarrow^l V}{\text{tl}(P_1) \Downarrow^l V} \Downarrow^l{\scriptstyle\text{TL}}$$

$$\frac{P \Downarrow^l \text{nil}_\sigma}{\text{elist}(P) \Downarrow^l \underline{T}} \Downarrow^l{\scriptstyle\text{ELIST1}} \qquad \frac{P_1 \Downarrow^l P_2 : P_3}{\text{elist}(P_1) \Downarrow^l \underline{F}} \Downarrow^l{\scriptstyle\text{ELIST2}}$$

D

Exercises

Note: Some of these exercises refer to concepts taught in courses other than these MGS notes.

1. Give a careful definition of a state s for the language $\mathbb{IMP}$.

2. Suppose that $\mathcal{L}$ is any location environment. Say what it means for a state s to be **sensible** for $\mathcal{L}$.

3. Fix a given location environment $\mathcal{L}$. Suppose that $(P\,,s) \rightsquigarrow (P'\,,s')$ is any transition, that $P :: \sigma$ is any type assignment, and that s is sensible for $\mathcal{L}$. You are asked to show some of the steps involved in verifying that $P' :: \sigma$ and that s' is sensible for $\mathcal{L}$.

   (a) Complete the following precise statement of what needs to be shown, by saying what ?, ??, and ??? are

   $$\forall\,(P\,,s) \rightsquigarrow (P'\,,s') \quad \boxed{\forall\,t.\quad ((P :: ??\ \textit{and}\ s\ ?)\ \textit{implies}\ (P' :: ??\ \textit{and}\ ???)}$$

   (b) Write down the induction hypotheses and the conclusion to be proved, for each of the rules $\rightsquigarrow_{\mathrm{OP}_1}$, $\rightsquigarrow_{\mathrm{SEQ}_1}$ and $\rightsquigarrow_{\mathrm{LOOP}}$.

   (c) Hence verify property closure for each of these three rules, giving *complete and thorough details*.

1. Some genuine $\mathbb{IMP}$ program expressions appear below, written in a sugared syntax. For each one, draw the corresponding abstract syntax tree, where you should make up suitable labels for the nodes.

   (a) $\underline{2} * (\underline{3} - \underline{3}) * (\underline{2} * \underline{5} + \underline{8})$

   (b) $\mathsf{while}\ l \leq \underline{1}\ \mathsf{do}\ (\mathsf{if}\ l = 1\ \mathsf{then}\ l' := \underline{1}\ \mathsf{else}\ l' := l' * l\ ;\ l' := \underline{4}\ ;\ l := l + 1)$

2. Let us write P for $\mathsf{while}\ l > \underline{5}\ \mathsf{do}\ P'$ where P' is the command

   $$\mathsf{if}\ l' = \underline{0}\ \mathsf{then}\ l := l - \underline{1}\ \mathsf{else\ skip}$$

   Suppose that $s \stackrel{\mathrm{def}}{=} \langle l \mapsto 7, l' \mapsto 0 \rangle$ is a state. Give a deduction tree for the evaluation

   $$(P\ ;\ l'' := \underline{7}\,,s) \Downarrow (\mathsf{skip}\,, s\{l \mapsto 5\}\{l'' \mapsto 7\})$$

3. Consider the following grammar of Boolean expressions where $\wedge$ denotes “and” and $\vee$ denotes “or”, and $b \in \mathbb{B}$

$$P ::= \underline{b} \mid B \wedge B \mid B \vee B$$

A language designer wants to explain to a programmer what sort of run time behaviour these Boolean expressions should have, by defining an evaluation relation of the form $B \Downarrow \underline{b}$. Informally, in the evaluation of any abstract syntax tree $B \wedge B'$ or $B \vee B'$, the *right* subtree should be considered first. According to whether B' evaluates to $\underline{T}$ or $\underline{F}$, the final result should be returned if possible. For example the deduction tree for $(\underline{F} \vee \underline{T}) \wedge \underline{F} \Downarrow \underline{F}$ should *not* involve the evaluation of $\underline{F} \vee \underline{T}$.

   (a) Write down rules which define such an evaluation semantics.

   (b) Give a deduction of

$$(\underline{T} \wedge (\underline{F} \vee \underline{T})) \wedge ((\underline{T} \wedge \underline{F}) \vee (\underline{F} \wedge \underline{F})) \Downarrow \underline{F}$$

1. The function $[\![-]\!]: Exp \to CSScodes$ takes $\mathbb{IMP}$ program expressions and turns them into CSS code. For example, $[\![\underline{c}]\!] \overset{\text{def}}{=} \mathsf{PUSH}(\underline{c})$. Write down recursive clauses for locations, arithmetic expressions, assignment, and sequencing.

2. Write down the CSS re-writes for OPerators, STOre, BRanch and LOOP instructions.

3. Suppose that the CSS machine is up-dated to run repeat loops. Such loops can be compiled as follows:

$$[\![\mathsf{repeat}\ P_1\ \mathsf{until}\ P_2]\!] \overset{\text{def}}{=} [\![P_1]\!] : \mathsf{LOOP}([\![P_2]\!] : \delta, [\![P_1]\!])$$

where δ is $\mathsf{BR}(\mathsf{PUSH}(\underline{F}), \mathsf{PUSH}(\underline{T}))$. Give a short, informal explanation of the standard run time semantics of a repeat loop, using this to justify, in terms of CSS re-writes, the compilation given. Then write down the sequence of CSS re-writes starting at

$$\boxed{[\![\mathsf{repeat}\ l := \underline{4} * l\ \mathsf{until}\ l > \underline{20}]\!] \parallel - \parallel \langle l \mapsto 6 \rangle}$$

4. Use the SECD machine to calculate $(\mathsf{fn}\,y.(\mathsf{fn}\,x.x + y)\underline{7})\ \underline{6}$ giving all re-writes.

1. Here is a grammar of value expressions for the language $\mathbb{FUN}^e$

$$V ::= \underline{c} \mid \mathsf{nil}_\sigma \mid (V, V) \mid F\,\vec{V} \mid V : V$$

Give three different, illustrative, concrete examples of value expressions for the “function” and “pair” parts of the grammar. In doing so, you should explain *in detail* the meaning of the notation $F\,\vec{V}$ by using an example of F with a maximum of $k = 3$ inputs.

2. Give the grammar of value expressions for the language $\mathbb{FUN}^l$. Give three different, illustrative, concrete examples of value expressions for the "function" and "list" parts of the grammar. Explain *in general detail, and not by example,* the meaning of the notation $F\,\vec{P}$.

3. Do all value expressions in $\mathbb{FUN}^e$ type check? If not, illustrate this with an example.

4. Given the declaration of a merge function

$$M\,l\,l' \quad = \quad \mathsf{if}\ \mathsf{hd}(l) \leq \mathsf{hd}(l')\ \mathsf{then}\ \mathsf{hd}(l) : (M\ \mathsf{tl}(l)\ l')\ \mathsf{else}\ \mathsf{hd}(l') : (M\ l\ \mathsf{tl}(l'))$$

   give a deduction of $M\ (\underline{5} : \underline{7} : \mathsf{nil})\ (\underline{2} : \underline{3} : \mathsf{nil}) \Downarrow^e \underline{2} : \underline{3} : \underline{5} : \underline{7} : \mathsf{nil}$, where you may assume that the branches close off with $M\,V\,\mathsf{nil} \Downarrow^e V$ for any list value V.

5. The expressions $\mathsf{hd}(l)$ and $\mathsf{hd}(l')$ occur twice in the declaration of M. As such, they are evaluated twice in each recursive call to M.

   (a) Write out a complete and correct declaration of M which deals with the cases when l or l' are empty.

   (b) Modify your new declaration of M so that each list l and l' will be evaluated at most once during each call to M.

Recall that in $\mathbb{IMP}$, the reflexive, transitive closure $\rightsquigarrow^*$ is equal to $\Downarrow$. One can give a transition semantics to $\mathbb{FUN}^e$, so that finite sequences of transitions correspond to evaluations, in the same way.

1. Write down rules, with hypotheses and conclusion of the form $P \rightsquigarrow P'$, which give a transition semantics for $\mathbb{FUN}^e$ expressions of the form $P\ op\ P'$ where op is an arithmetic operator.

2. Recall that abstractions of the form $\mathsf{fn}\,v.E$ are values. In a transition semantics, one should have a sequence of transitions from a (function) application $P\,P'$, resulting in $(\mathsf{fn}\,v.E)\ P'$, and then further transitions to $(\mathsf{fn}\,v.E)\ V'$. The function abstraction is called, and then execution continues. Give *three* rules which provide an eager transition semantics for such $P\,P'$ programs.

3. Use your rules to give the full transition sequence of $((\mathsf{fn}\,x.\mathsf{fn}\,y.x+y)\ \underline{2})\ (\underline{7}+\underline{1})$. Give a deduction tree for the second transition only.

4. In $\mathbb{FUN}^e$, we would hope that the transition semantics, and evaluation semantics, coincide. Certainly we would hope that

$$\forall P \Downarrow^e V \quad \boxed{P \rightsquigarrow^* V}$$

   Give part of the verification of this fact, by proving property closure for the rule $\Downarrow^e$ AP, giving *full and complete details*.

This question introduces a simple imperative language of integer and Boolean expressions with local declarations. Let $A\colon Var \to Loc$ be a finite partial function from a set Var of variables to a set Loc of locations. We call any such function A an *allocator*; it allocates (memory) locations to program variables. A finite partial function $s\colon Loc \to \mathbb{Z} \cup \mathbb{B}$ is a *state*. The standard notations $A\{v \mapsto l\}$ and $s\{l \mapsto c\}$ denote updated allocators and states. Let σ range over the types int and bool. Consider a set of programme expressions inductively defined by four rules, where $\wedge$ is the logical "AND" function:

$$\frac{-}{v}\,[v \in Var] \qquad \frac{-}{\underline{c}}\,[c \in \mathbb{Z} \cup \mathbb{B}] \qquad \frac{P_1 \quad P_2}{P_1\ op\ P_2}\ op \in \{+, \wedge, \leq\} \qquad \frac{P_1 \quad P_2}{\mathsf{let}\ v = P_1\ \mathsf{in}\ P_2} \qquad (\dagger)$$

The idea of $\mathsf{let}\ v = P_1\ \mathsf{in}\ P_2$ is that it provides a "local" value of v in P_2. The variable v is a scoping variable, and its scope is P_2. Thus free occurrences of v in P_2 are *bound* in $\mathsf{let}\ v = P_1\ \mathsf{in}\ P_2$. A *context* Γ is a set of typed variables, typically written

$$\Gamma = x_1 :: \sigma_1, \ldots, x_n :: \sigma_n.$$

A type assignment for the language looks like $\Gamma \vdash P :: \sigma$. An example is

$$x :: \mathsf{bool}, y :: \mathsf{bool} \vdash x \wedge y :: \mathsf{bool}$$

A *configuration* $(P, (A, s))$ consists of a program expression, together with an allocator and a state. A is used to allocate locations to any free variables which appear in P. An example is

$$(x + (\mathsf{let}\ x = \underline{4}\ \mathsf{in}\ x + y), (\langle x \mapsto l, y \mapsto l'\rangle, \langle l \mapsto 2, l' \mapsto 7\rangle)) \qquad (\dagger\dagger)$$

1. Write $fvar(P)$ for the set of free variables in P. Give a recursive definition of $fvar(P)$. Use your definition to calculate the set $fvar(x + (\mathsf{let}\ x = \underline{4}\ \mathsf{in}\ x + y))$ giving your steps.

2. (a) Write down some rules which inductively define an appropriate type assignment system for this language.

   (b) Let $\Gamma \overset{\text{def}}{=} n :: \mathsf{int}, m :: \mathsf{int}, x :: \mathsf{bool}, y :: \mathsf{bool}$. Use your rules to give a deduction of $\Gamma \vdash \mathsf{let}\ v = (m \leq n)\ \mathsf{in}\ (v \wedge x) :: \mathsf{bool}$.

3. (a) A deterministic evaluation relation $(P, (A, s)) \Downarrow \underline{c}$ can be inductively defined by four rules. For the example $(\dagger\dagger)$ above, $c = 13$. In order to evaluate the configuration $(\mathsf{let}\ v = P_1\ \mathsf{in}\ P_2, (A, s))$, one evaluates P_1 with (A, s) which may give a constant $\underline{c_1}$. Then one chooses a *fresh* location $\hat{l}$ which does not appear in the domain of definition of s. P_2 is evaluated in (A', s') in which A' is A with v updated to $\hat{l}$, and s' is s with $\hat{l}$ updated to c_1. Write down four rules which define the evaluation relation, making use of "update" notation.

(b) Let $A \overset{\text{def}}{=} \langle v \mapsto l, m \mapsto l', n \mapsto l'' \rangle$ and $s \overset{\text{def}}{=} \langle l \mapsto T, l' \mapsto 7, l'' \mapsto 1 \rangle$. Use your rules to give a deduction tree for

$$(v \wedge (\mathsf{let}\ v = (m \leq n)\ \mathsf{in}\ (v \wedge \underline{T}))\,,\,(A,s)) \Downarrow \underline{F}$$

This question concerns the language $\mathbb{IMP}$ and the CSS machine. You are reminded that $\mathbb{IMP}$ does *not have* Boolean negation expressions $\mathsf{not}(B)$, and that the CSS machine does *not have* any form of Boolean negation in the standard instruction set.

1. Write down grammars which define the CSS *instructions*, *codes* and *stacks*.

2. Define the function $[\![-]\!]\colon Exp \to CSScodes$ which takes an $\mathbb{IMP}$ program expression and turns it into CSS code, by writing down recursive clauses for constants, locations, arithmetic expressions, assignment, skip, sequencing, conditionals and while loops.

3. Write down the (four) CSS re-writes for STOre, BRanch and LOOP instructions.

4. A $\mathsf{do}\ P\ \mathsf{while}\ P$ loop repeatedly computes P and then P, until P computes to $\underline{F}$ when execution stops. Show how to compile such loops into *the standard CSS instruction set*. Hence write down the complete sequence of re-writes starting at

$$\begin{array}{|c|c|c|}\hline [\![\mathsf{do}\ l := l * \underline{2}\ \mathsf{while}\ l = \underline{20}]\!] & - & \langle l \mapsto 20 \rangle \\ \hline \end{array}$$

Hint: Write P and P for the Boolean and command, and only write out $[\![P]\!]$ and $[\![P]\!]$ in full when the individual instructions are required.

5. A $\mathsf{repeat}\ P\ \mathsf{until}\ P$ loop repeatedly computes P and then P, until P computes to $\underline{T}$ when execution stops. Show how to compile such loops into *the standard CSS instruction set*. Hence write down the sequence of re-writes starting at

$$\begin{array}{|c|c|c|}\hline [\![\mathsf{repeat}\ l := l * \underline{2}\ \mathsf{until}\ l = \underline{20}]\!] & - & \langle l \mapsto 10 \rangle \\ \hline \end{array}$$

6. Give a precise statement saying what it means for the CSS machine to be *correct* for the $\mathbb{IMP}$ evaluation semantics. You are *not* required to give a proof of correctness.

1. What is a **finite transition sequence** for an $\mathbb{IMP}$ configuration $(P\,,s)$? Write down an example of a configuration which has *four* transitions, and then write down the complete transition sequence. *You are not asked to give any deductions of individual transitions.*

2. Give an example of a configuration with an infinite transition sequence, and write down enough transitions to demonstrate that the sequence is infinite. *You are not asked to give any deductions of individual transitions.*

3. Write down the complete transition sequence for the configuration $(P\,,s)$ where

$$\begin{aligned} s &\overset{\text{def}}{=} \langle l \mapsto 8, l'' \mapsto 1\rangle \\ P &\overset{\text{def}}{=} \mathsf{if}\ l \leq \underline{4}\ \mathsf{then}\ l := \underline{3}\ \mathsf{else}\ (\mathsf{skip}\ ;\ l' := \underline{4} + l'') \end{aligned}$$

4. When is a configuration **terminal**? When is a configuration **stuck**? Hence state precisely when a configuration $(P\,,s)$ is *not stuck*.

   For each of the following, say whether or not the configuration is stuck. Justify your answers formally by considering deduction trees for transitions.

   (a) $(\underline{T}\,,s)$
   (b) $(\underline{T}+\underline{4}\,,s)$
   (c) $(\underline{4}+\underline{T}+\underline{5}\,,s)$
   (d) $(\underline{4}+\underline{5}+\underline{T}\,,s)$
   (e) $(l+l'+\underline{5}\,,\langle l \mapsto 1\rangle)$
   (f) $(l+l'+\underline{5}\,,\langle l' \mapsto 1\rangle)$

5. Suppose that we are given a fixed location environment $\mathcal{L}$, and a fixed state s for which every location l in $\mathcal{L}$ is in the domain of definition of s.

   If P is any $\mathbb{IMP}$ expression, σ is any $\mathbb{IMP}$ type, and $P :: \sigma$ is any valid type assignment, then $(P\,,s)$ is not stuck. This can be proved by a Rule Induction over type assignments. Write down the exact statement to be proved by induction, and demonstrate part of the proof by showing Property Closure for the rules $::$ LOC, $::$ LOOP and $::$ IOP.

   Hint: You may assume that, for any transition $(Q\,,r) \rightsquigarrow (Q'\,,r')$*, if* $Q :: \mathsf{int}$*, then* $r = r'$ *and* $Q' :: \mathsf{int}$.

1. Suppose that $\mathsf{Z} :: [\mathsf{int}] \to [\mathsf{int}] \to [(\mathsf{int},\mathsf{int})]$ and that

$$\mathsf{Z}xy \;=\; \mathsf{if\ elist}(x)\ \mathsf{then\ nil\ else}\ ((\mathsf{hd}(x),\mathsf{hd}(y)) : (\mathsf{Z}\,(\mathsf{tl}(x))\,(\mathsf{tl}(y))))$$

   The idea is that the function Z takes two lists of integers of the same length, and produces the list of pairs of integers, with each pair consisting of consecutive elements from the two lists. Show that

$$\mathsf{Z}\,(\underline{3} : \mathsf{nil})\,(\underline{7} : \mathsf{nil}) \Downarrow^{e} (\underline{3},\underline{7}) : \mathsf{nil}$$

2. A professor, who delights in making simple things difficult, wants to code the factorial function in a complicated way. He writes down the identifier environment I

$$\begin{aligned} \mathsf{C} &:: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \\ F &:: \mathsf{int} \to \mathsf{int} \end{aligned}$$

   and the function declaration dec_I

$$\begin{aligned} \mathsf{C}\, x\, y\, z &= \mathsf{if}\ x = \underline{1}\ \mathsf{then}\ y\ \mathsf{else}\ z \\ F\, x &= \mathsf{C}\, x\, \underline{1}\, (x * F\, (x - \underline{1})) \end{aligned}$$

   and says that the $\mathbb{FUN}^e$ program

$$dec_I \quad in\ F\, \underline{n}$$

   will compute the factorial of $\underline{n}$ for each integer $n \geq 1$. A student claims that this is not true, but that $F\, \underline{n}$ *will* evaluate to the factorial of $\underline{n}$ in $\mathbb{FUN}^l$.

   (a) The student's claim that $F\, \underline{n} \Downarrow^l \underline{n!}$ where $n!$ is the factorial of n is correct. Show that this is plausible by giving the deduction tree of $F\, \underline{2} \Downarrow^l \underline{2}$, and explaining briefly what the tree will look like for a general integer $n \geq 1$.

   (b) Is the student's claim, that the professor's definition of factorial does not work, true or false? Explain your answer in *detail*. *Hint: consider the deduction tree of $F\, \underline{2} \Downarrow^e \underline{2}$.*

1. Explain the idea of **eager** evaluation in $\mathbb{FUN}^e$ by giving the type and declaration of a function G for which $G\, \underline{7}\, (\underline{2} * \underline{3})\, (\underline{4} * \underline{5})$ type checks, and explaining *informally* the stages involved in the evaluation of $G\, \underline{7}\, (\underline{2} * \underline{3})\, (\underline{4} * \underline{5})$.

2. Explain the idea of **lazy** evalution in $\mathbb{FUN}^l$ in a similar way.

3. (a) Give the formal definition of **identifier type**. You should explain what **constant** and **function** identifiers are, and what the phrase "k is the maximum number of inputs taken by F" means. Any non-base types referred to in your answer should be defined by a grammar.

   (b) Give the formal grammar which specifies **value expressions** in $\mathbb{FUN}^e$, explaining your notation in detail.

   (c) Use the function G you declared above to give three examples of values in $\mathbb{FUN}^e$ of the form $G\, \vec{V}$, with a different number of inputs in each case. Explain informally, in terms of the evaluation semantics, why your examples are indeed values.

   (d) Are constants values? Explain your answer informally.

Notation Index

Subject Index