Preface

These notes are to accompany the module CO3008. They contain all of the core material for the course. For more motivation and background, examples, and further comments about some of the details of proofs, please attend the lectures.

Please do let me know about any typos or other errors which you find. If you have any other (constructive) comments, please tell me about them.

Books recommended for CO3008 are listed in the Study Guide: ask if you need advice.

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

Acknowledgements

Parts of these notes were originally developed from courses given by Professor A. M. Pitts, University of Cambridge. I also acknowledge many other books and papers, which are too numerous to detail here (some are given in the Study Guide).

First edition 1999. Revised 2000, 2001, 2003.

Contents

List of Tables

List of Figures

1

Mathematical Prerequisites

1.1 Introduction

Definitions 1.1.1 We shall begin by reviewing some mathematics which will be used throughout this course. Some of the material you have seen before. For the material that is new, you may need to flesh out the definitions and concepts using books or other sets of notes. However, most of the *basic* ideas you have met in MC 111.

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 1.1.

• 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.*

1.2 Logic

We sometimes write $A \equiv B$ to indicate **syntactic identity**. Thus $2+3=5$ but $2+3 \not\equiv 5$.

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 1.1: Some Greek Characters

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

- P *and* Q (sometimes written $P \wedge Q$);
- P *or* Q (sometimes written $P \vee Q$);
- P *implies* Q (sometimes written $P \Rightarrow Q$ 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

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

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

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!).

1.3 Sets

Definitions 1.3.1 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:

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

[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.

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

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 $1 \leq i \leq n$. 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

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

For example,

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

Examples 1.3.2

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

(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* $y = 100$.

1.4 Relations

Definitions 1.4.1 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 1.4.3. 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 1.4.2 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

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

and if $(\text{Ron}, \text{Roy}) \in$ *is_the_father_of* then we can write instead

$$\text{Ron } \textit{is\_the\_father\_of } \text{Roy}.$$

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 1.4.3 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 1.4.4 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 1.4.5 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 1.4.3.

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

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

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

(4) $R \stackrel{\text{def}}{=} \{(\alpha,\beta), (\beta,\gamma), (\alpha,\gamma)\}$ 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 1.4.6 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 1.4.7 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

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

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,

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

Example 1.4.8 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:

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

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

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

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

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.

1.5 Functions

1.5.1 Total Functions

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

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

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

1.5.2 Partial Functions

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

$$[A, B]_{par} \stackrel{\text{def}}{=}$$

$$\{\ f \in \mathcal{P}(A \times B)\ \mid\ \forall a \in A, \forall b, b' \in B, ((a, b) \in f \text{ and } (a, b') \in f) \text{ implies } b = b'\ \}.$$

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

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

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

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

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?

2

Abstract Syntax and Rule Induction

2.1 Inductively Defined Sets

2.1.1 Abstract Syntax Trees

Motivation 2.1.1 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

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

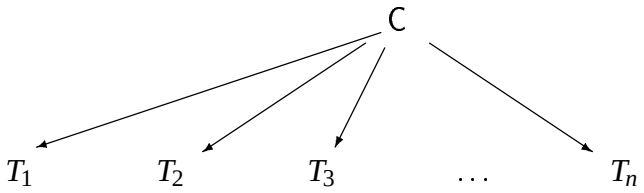
and think of the conditional as a *constructor* which acts on three arguments, to "construct" a new program (you might like to think of a constructor as a function). Now

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

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

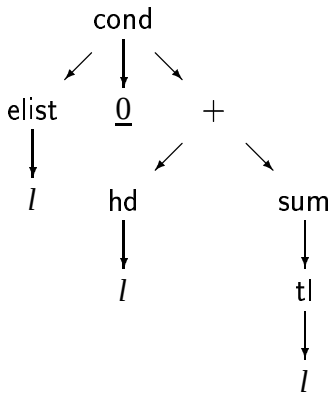
Finally, looking at one of the sub-parts of this expression, namely $\mathsf{hd}(l)$, we can think of $\mathsf{hd}(l)$ as a constructor hd acting on a single argument, l.

Definitions 2.1.2 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



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".

2.1.2 Rule Sets

Definitions 2.1.3 Let us first introduce some notation. Consider

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

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

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

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 2.1.4 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 2.1.5 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.

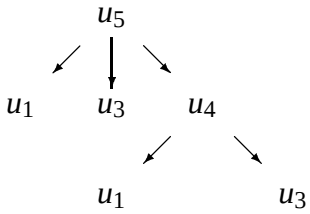
We then say that the set **inductively defined** by $\mathcal{R}$ consists of those elements $u \in \mathcal{U}$ which have a deduction with root node labelled by u.

Examples 2.1.6

(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

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

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

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

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]

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

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

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

(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

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

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

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

[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 2.1: Rule Induction

2.2 Principles of Induction

Motivation 2.2.1 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 2.2.2 We present in Figure 2.1 a useful principle called **Rule Induction**. It will be used throughout the course to prove facts about programming languages.

Motivation 2.2.3 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

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

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 $\forall n \in \mathbb{N}.\ (\phi(n) \textit{ implies } \phi(n+1))$

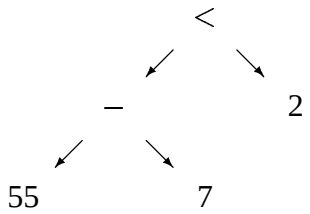
and this amounts to precisely what one needs to verify for Mathematical Induction.

Examples 2.2.4

(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



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 2.2.5 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 $\phi(C(T_1,\ldots,T_n))$ 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

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

[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.

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

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

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):

(*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 $\forall w \in S.\phi(w)$.

2.3 Recursively Defined Functions

Definitions 2.3.1 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 2.3.2

(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.

3

Typing of an Imperative Language

3.1 Introduction

Motivation 3.1.1 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 4 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 4.

3.2 The Syntax of Expressions

Definitions 3.2.1 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. Recall page 9. 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	$\stackrel{\text{def}}{=}$	$\{l_1,l_2,\ldots\}$	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 $\mathbb{Z} \cup \mathbb{B}$. 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
	|	l	memory location
	|	$iop(P,P')$	integer operator
	|	$bop(P,P')$	boolean operator
	|	skip	do nothing
	|	$\mathsf{assign}(l,P')$	assignment
	|	$\mathsf{sequence}(P,P')$	conditional
	|	$\mathsf{cond}(P,P',P'')$	while loop
	|	$\mathsf{while}(P,P')$	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 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 3.2.2 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, for example compare the following two commands:

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

Exercise: draw the syntax trees.

3.3 Type Checking and Inference

Motivation 3.3.1 The time at which types are assigned to expressions, and type errors checked for, varies among languages. **Statically typed** languages carry out type checking by static analysis of code at *compile-time*. Pascal and Haskell are examples of such languages. This is useful, but the richer the type system, the harder it is to achieve without putting in a lot of *explicit type information*—you may have found that with some of your Haskell programs, you had to add in a type declaration to make a program compile. Pascal requires much typing information within program code. **Dynamically typed** languages carry out type checking at *run time*.

We shall soon introduce types for $\mathbb{IMP}$. Let us review some of the basic ideas. Types appear in many kinds of software system and programming language. Expressions of the language can be organized using types to try to help reduce program errors. For example, the operator $+$ should only act on numbers. If a program contains, say, $4+T$, then an error will result. Detecting this kind of error is known as *type checking*. Such checks can be made both at compile time and run time. However, in this course, we shall only ever consider compile time checks. This means that we shall only consider the detection of type errors by looking at the syntax of programs. The error given above will occur when we try to compile $4+T$.

Types also play other roles. For example, *polymorphism*, very crudely, means that certain operators and functions can have more than one type. This allows code re-use; one does not have to write a new list reversing function for *each* type of list (integer

list, Boolean list and so on). We shall return to polymorphism in Chapter 8. Types also allow for the careful structuring of data, using ADTs and modules.

Given a compiled program, there are two major kinds of run-time error. A **trapped** error is one for which execution halts immediately. Examples are dividing by 0 and calling a top level exception. An **untrapped** error is one for which execution does not necessarily halt. An example is accessing data past the end of an array, which one can do in *C*! A language is said to be **safe** if all syntactically legal programs do not yield certain run-time errors. And one way we can check for this legality is by doing type checking at compile time. *C* has types, but is *not safe*. JAVA was claimed to be safe, but in 1997 this was shown not to be the case.

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. **Type safety** is the property that if $P :: \sigma$ then certain kinds of errors can not occur at P's run-time. Given P and σ, **type checking** is the process of checking that $P :: \sigma$ is valid. Given just P, **type inference** is the process of trying to find a type σ for which $P :: \sigma$ if there is one—the process *fails* if no such type exists.

Definitions 3.3.2 The types of the language $\mathbb{IMP}$ are given by the grammar

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

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

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

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 3.1.

Example 3.3.3 We give an example of a deduction of a type assignment, given the location environment $l :: \mathsf{int}, l' :: \mathsf{int}$.

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

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

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

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

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

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

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

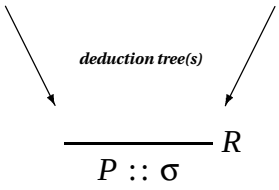
$$\frac{P_1 :: \mathsf{cmd} \quad P_2 :: \mathsf{cmd}}{P_1\ ;\ P_2 :: \mathsf{cmd}} :: \text{SEQ}$$

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

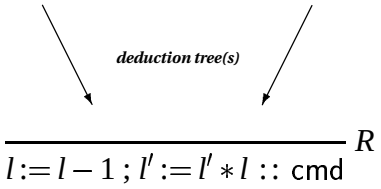
Table 3.1: $\mathbb{IMP}$ type assignments $P :: \sigma$

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

⋙ **NOTE 3.3.4** ***In order to produce the missing deductions, consider the following situation***



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 3.3.3. We have***



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

Example 3.3.5 Perform type checking for $l := l + \underline{4} :: \mathsf{cmd}$ given $\mathcal{L} \stackrel{\text{def}}{=} l :: \mathsf{int}$. We do this by giving a deduction. The outermost constructor is assign, thus the final rule used must be $::$ ASS. Working backwards we have

$$\dfrac{\dfrac{?}{l :: \sigma}\,?? \qquad \dfrac{???}{l + \underline{4} :: \sigma}\,????}{l := l + \underline{4} :: \mathsf{cmd}} :: \text{ASS}$$

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

Motivation 3.3.6 Given a location environment $\mathcal{L}$, and a program expression P, there is a simple algorithm which will infer if P can be assigned a type. If such a type exists we say P is **typable**—in this case the algorithm will *succeed* and given P as input will return the type as output. If not, the algorithm *fails*. In a real language, such type inference is often performed by the compiler. The programmer declares the types of various locations, corresponding to our $\mathcal{L}$, writes a program P, and compiles the program. The compilation succeeds if the type inference algorithm succeeds.

Definitions 3.3.7 Given $\mathcal{L}$ and P, we shall define a function Φ which given P as input will either return a type for P, or will return *FAIL*. It is defined in Table 3.2.

$$
\begin{array}{rcl}
\Phi(\underline{T}) & = & \mathsf{bool} \\
\Phi(\underline{F}) & = & \mathsf{bool} \\
\Phi(\underline{n}) & = & \mathsf{int} \\
\Phi(l) & = & \begin{cases} \tau & \text{if } l :: \tau \in \mathcal{L}, \textit{ and } \tau = \mathsf{int} \textit{ or } \mathsf{bool} \\ \textit{FAIL} & \text{otherwise} \end{cases} \\
\Phi(P_1 \textit{ iop } P_2) & = & \begin{cases} \mathsf{int} & \text{if } \Phi(P_1) = \mathsf{int} \text{ and } \Phi(P_2) = \mathsf{int} \\ \textit{FAIL} & \text{otherwise} \end{cases} \\
\Phi(P_1 \textit{ bop } P_2) & = & \begin{cases} \mathsf{bool} & \text{if } \Phi(P_1) = \mathsf{int} \text{ and } \Phi(P_2) = \mathsf{int} \\ \textit{FAIL} & \text{otherwise} \end{cases} \\
\Phi(l := P) & = & \begin{cases} \mathsf{cmd} & \text{if } \Phi(l) = \tau \text{ and } \Phi(P) = \tau, \textit{ and } \tau = \mathsf{int} \textit{ or } \mathsf{bool} \\ \textit{FAIL} & \text{otherwise} \end{cases} \\
\Phi(P_1 \,;\, P_2) & = & \begin{cases} \mathsf{cmd} & \text{if } \Phi(P_1) = \mathsf{cmd} \text{ and } \Phi(P_2) = \mathsf{cmd} \\ \textit{FAIL} & \text{otherwise} \end{cases} \\
\Phi(\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3) & = & \begin{cases} \mathsf{cmd} & \text{if } \Phi(P_1) = \mathsf{bool} \text{ and } \Phi(P_2) = \mathsf{cmd} \text{ and} \\ & \Phi(P_3) = \mathsf{cmd} \\ \textit{FAIL} & \text{otherwise} \end{cases} \\
\Phi(\mathsf{while}\ P_1\ \mathsf{do}\ P_2) & = & \begin{cases} \mathsf{cmd} & \text{if } \Phi(P_1) = \mathsf{bool} \text{ and } \Phi(P_2) = \mathsf{cmd} \\ \textit{FAIL} & \text{otherwise} \end{cases}
\end{array}
$$

Table 3.2: $\mathbb{IMP}$ Type Inference based on a given $\mathcal{L}$

Examples 3.3.8

(1) Let $\mathcal{L} \stackrel{\text{def}}{=} l :: \mathsf{int}, l' :: \mathsf{bool}$. Then the program expression

$$P \stackrel{\text{def}}{=} \mathsf{if}\ l\ \mathsf{then}\ l := \underline{3}\ \mathsf{else}\ l' := \underline{T}$$

will fail type inference at compilation time. To calculate $\Phi(P)$ we first calculate $\Phi(l)$. Note that $l :: \mathsf{bool}$ is not an element of $\mathcal{L}$, and so $\Phi(l) = \mathit{FAIL}$. Thus $\Phi(P) = \mathit{FAIL}$.

(2) If we change the first l in P to l' and call this new program expression P', then $\Phi(P')$ succeeds with the type cmd. Exercise: check this!

4

Operational Semantics of an Imperative Language

4.1 A Transition Relation

Motivation 4.1.1 Recall that the locations of $\mathbb{IMP}$ are elements of some given set *Loc*. A state is given by specifying the data which is held in the locations. For us, the data is simple and only consists of integers and Booleans. Thus a state can be modelled as a partial function from the set of locations *Loc* to $\mathbb{Z}$. As we have mentioned, a configuration is given by a pair (P,s) where P is a program expression and s a state. Such a configuration can be "executed" or "run". Informally, the idea is that at run time, the expression P causes a sequence of small changes to the state s, and at the end of the run we have a final machine state. Each small change to the state is called a *transition step*. A program run consists of a sequence of such transition steps. We shall define assertions of the form $(P,s) \rightsquigarrow (P',s')$ which assert that in state s, P executes in one transition step to P' with the state after the computation step being s'. Such assertions comprise a formal *operational semantics*. We shall also give an operational semantics which shows how expressions can execute "immediately" to produce a final state (and a program output if the expression is an integer or a Boolean), and show how this style of operational semantics matches the former "transition step" definition in an exact way.

Definitions 4.1.2 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 $\langle l \mapsto 4, l' \mapsto 5\rangle$, $\langle l_1 \mapsto 45, l_2 \mapsto T, l_3 \mapsto 2\rangle$, 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

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

Given that $\mathcal{L}$ is a location environment, we say that state s is **sensible** for $\mathcal{L}$ if for all $l :: \sigma$ in $\mathcal{L}$,

$$l \in dom(s) \quad \textit{and} \quad \underline{s(l)} :: \sigma$$

where $dom(s)$ is the domain of definition of s.

The elements of the set $Exp \times States$ will be known as **configurations**. We shall inductivley define a binary relation on $Exp \times States$, namely

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

by the rules in Table 4.1, where we shall write $(P_1, s_1) \rightsquigarrow (P_2, s_2)$ instead of

$$((P_1, s_1), (P_2, s_2)) \in \rightsquigarrow.$$

We call $\rightsquigarrow$ a **transition** relation, and any instance of a relationship in $\rightsquigarrow$ is called a **transition step**.

Example 4.1.3 Let us write Q for $\mathsf{while}\ l > \underline{0}\ \mathsf{do}\ Q'$ where Q' is the command $l' := l' + \underline{2}$; $l := l - \underline{1}$. Suppose that s is a state for which $s(l) = 1$ and $s(l') = 0$. Let us write $s' \stackrel{\text{def}}{=} s\{l' \mapsto 2\}$ and $s'' \stackrel{\text{def}}{=} s\{l' \mapsto 2\}\{l \mapsto 0\} = s'\{l \mapsto 0\}$. We give an example of a sequence of configuration transitions for the language $\mathbb{IMP}$ in Figure 4.1. We also give, as an example, the deduction of the transition step $\rightsquigarrow_*$ in Figure 4.2. Of course, each of the transition steps given in Figure 4.1 have similar deductions to that for $\rightsquigarrow_*$, but in practice one can write down (correct) transition steps directly, without formal deduction trees, simply by understanding the intended meaning of the language $\mathbb{IMP}$.

Proposition 4.1.4 Let $\mathcal{L}$ be a location environment and s_1 sensible. The binary relation $\rightsquigarrow$ enjoys the following properties:

(i) Suppose that $P_1 :: \sigma$. Then for any transition $(P_1, s_1) \rightsquigarrow (P_2, s_2)$ we have $P_2 :: \sigma$. Thus the type of P_1 is *preserved* under transitions.

(ii) Further, if σ is either int or bool, then $s_1 = s_2$.

Proof We shall prove

$$\forall\, (P_1, s_1) \rightsquigarrow (P_2, s_2) \quad \boxed{\forall\, \sigma. \quad (P_1 :: \sigma \ \textit{implies}\ P_2 :: \sigma)}$$

We have to check property closure for each of the rules defining $\rightsquigarrow$. We look at a couple of examples.

(*Property Closure for* $\rightsquigarrow$ LOC) We have to show that $l :: \sigma$ *implies* $\underline{s(l)} :: \sigma$ for any σ. This is immediate as s is sensible.

Rules for integer and Boolean Expressions

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

$$\frac{(P_1\,,s) \rightsquigarrow (P_2\,,s)}{(P_1\ op\ P\,,s) \rightsquigarrow (P_2\ op\ P\,,s)}\rightsquigarrow_{\text{OP}_1}$$

$$\frac{(P_1\,,s) \rightsquigarrow (P_2\,,s)}{(\underline{n}\ op\ P_1\,,s) \rightsquigarrow (\underline{n}\ op\ P_2\,,s)}\rightsquigarrow_{\text{OP}_2} \qquad \frac{}{(\underline{n_1}\ op\ \underline{n_2}\,,s) \rightsquigarrow (\underline{n_1\ op\ n_2}\,,s)}\rightsquigarrow_{\text{OP}_3}$$

Rules for Commands

$$\frac{(P_1\,,s) \rightsquigarrow (P_2\,,s)}{(l{:=}P_1\,,s) \rightsquigarrow (l{:=}P_2\,,s)}\rightsquigarrow_{\text{ASS}_1} \qquad \frac{}{(l{:=}\underline{c}\,,s) \rightsquigarrow (\mathsf{skip}\,,s\{l\mapsto c\})}\rightsquigarrow_{\text{ASS}_2}$$

$$\frac{(P_1\,,s_1) \rightsquigarrow (P_2\,,s_2)}{(P_1\,;P\,,s_1) \rightsquigarrow (P_2\,;P\,,s_2)}\rightsquigarrow_{\text{SEQ}_1} \qquad \frac{}{(\mathsf{skip}\,;P\,,s) \rightsquigarrow (P\,,s)}\rightsquigarrow_{\text{SEQ}_2}$$

$$\frac{(P\,,s) \rightsquigarrow (P'\,,s)}{(\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,,s) \rightsquigarrow (\mathsf{if}\ P'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,,s)}\rightsquigarrow_{\text{COND}_1}$$

$$\frac{}{(\mathsf{if}\ \underline{T}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,,s) \rightsquigarrow (P_1\,,s)}\rightsquigarrow_{\text{COND}_2}$$

$$\frac{}{(\mathsf{if}\ \underline{F}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,,s) \rightsquigarrow (P_2\,,s)}\rightsquigarrow_{\text{COND}_3}$$

$$\frac{}{(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,s) \rightsquigarrow (\mathsf{if}\ P_1\ \mathsf{then}\ (P_2\,;\mathsf{while}\ P_1\ \mathsf{do}\ P_2)\ \mathsf{else}\ \mathsf{skip}\,,s)}\rightsquigarrow_{\text{LOOP}}$$

Table 4.1: Configuration Transitions $(P\,,s) \rightsquigarrow (P'\,,s')$ in $\mathbb{IMP}$

$$
\begin{array}{lll}
(Q, s) & \rightsquigarrow & (\text{if } l > \underline{0} \text{ then } Q' ; Q \text{ else skip}, s) \\
& \rightsquigarrow & (\text{if } \underline{1} > \underline{0} \text{ then } Q' ; Q \text{ else skip}, s) \\
& \rightsquigarrow & (\text{if } \underline{T} \text{ then } Q' ; Q \text{ else skip}, s) \\
& \rightsquigarrow & (Q' ; Q, s) \\
& \rightsquigarrow & ((l' := \underline{0} + \underline{2} ; l := l - \underline{1}) ; Q, s) \\
& \rightsquigarrow & ((l' := \underline{2} ; l := l - \underline{1}) ; Q, s) \\
& \rightsquigarrow_* & ((\text{skip} ; l := l - \underline{1}) ; Q, s') \\
& \rightsquigarrow & (l := l - \underline{1} ; Q, s') \\
& \rightsquigarrow & (l := \underline{1} - \underline{1} ; Q, s') \\
& \rightsquigarrow & (l := \underline{0} ; Q, s') \\
& \rightsquigarrow & (\text{skip} ; Q, s'') \\
& \rightsquigarrow & (Q, s'') \\
& \rightsquigarrow & (\text{if } l > \underline{0} \text{ then } Q' ; Q \text{ else skip}, s'') \\
& \rightsquigarrow & (\text{if } \underline{0} > \underline{0} \text{ then } Q' ; Q \text{ else skip}, s'') \\
& \rightsquigarrow & (\text{if } \underline{F} \text{ then } Q' ; Q \text{ else skip}, s'') \\
& \rightsquigarrow & (\text{skip}, s'')
\end{array}
$$

Figure 4.1: A Transition Sequence in $\mathbb{IMP}$

$$
\dfrac{\dfrac{\dfrac{}{(l' := \underline{2}, s) \rightsquigarrow (\text{skip}, s')}\,\rightsquigarrow_{\text{ASS}_2}}{(l' := \underline{2} ; l := l - \underline{1}, s) \rightsquigarrow (\text{skip} ; l := l - \underline{1}, s')}\,\rightsquigarrow_{\text{SEQ}_1}}{((l' := \underline{2} ; l := l - \underline{1}) ; Q, s) \rightsquigarrow ((\text{skip} ; l := l - \underline{1}) ; Q, s')}\,\rightsquigarrow_{\text{SEQ}_1}
$$

Figure 4.2: An Example Deduction of a Transition

(*Property Closure for* $\rightsquigarrow_{OP_2}$) The induction hypothesis is

$$\forall\, \sigma.\quad (P_1 :: \sigma \textit{ implies } P_2 :: \sigma) \qquad \textbf{IH}$$

We have to prove

$$\forall\, \sigma.\quad (\underline{n}\ op\ P_1 :: \sigma \textit{ implies } \underline{n}\ op\ P_2 :: \sigma) \qquad \textbf{C}$$

Let σ denote any type and suppose that $\underline{n}\ op\ P_1 :: \sigma$. This can only be deduced from $::_{\text{IOP}}$ or $::_{\text{BOP}}$, so the type assignment holds only if σ is int or bool. We will just look at case of $::_{\text{IOP}}$; the other is similar. So we must have $\underline{n} :: \mathsf{int}$ and $P_1 :: \mathsf{int}$. Using **IH** we deduce that $P_2 :: \mathsf{int}$ and thus, using $::_{\text{IOP}}$, that $\underline{n}\ op\ P_2 :: \mathsf{int}$, as required. As σ was arbitrary, **C** holds. □

Theorem 4.1.5 The operational semantics of $\mathbb{IMP}$, as specified by the transition relation $\rightsquigarrow$, is **deterministic**, that is to say that for all program expressions P, P' and P'', and states s, s' and s'', if

$$(P\,,s) \rightsquigarrow (P'\,,s') \qquad \textit{and} \qquad (P\,,s) \rightsquigarrow (P''\,,s'')$$

then $P' = P''$ and $s' = s''$.

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

$$\phi(((P\,,s),(P'\,,s'))) \stackrel{\text{def}}{=} \boxed{\forall (X\,,x),\quad (P\,,s) \rightsquigarrow (X\,,x) \textit{ implies } (X = P' \textit{ and } x = s')}$$

then we can prove that

$$\forall (P\,,s) \rightsquigarrow (P'\,,s'),\quad \phi(((P\,,s),(P'\,,s'))) \qquad (*)$$

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

We consider property closure for just one rule, say

$$\frac{(P_1\,,s) \rightsquigarrow (P_2\,,s)}{(l{:=}P_1\,,s) \rightsquigarrow (l{:=}P_2\,,s)}\ \rightsquigarrow_{\text{ASS}_1}$$

The inductive hypothesis is $\phi(((P_1\,,s),(P_2\,,s)))$, that is

$$\forall (Y\,,y),\quad (P_1\,,s) \rightsquigarrow (Y\,,y) \textit{ implies } (Y = P_2 \textit{ and } y = s) \qquad (IH)$$

We need to prove $\phi(((l{:=}P_1\,,s),(l{:=}P_2\,,s)))$, that is

$$\forall (Z\,,z),\quad (l{:=}P_1\,,s) \rightsquigarrow (Z\,,z) \textit{ implies } (Z = (l{:=}P_2) \textit{ and } z = s) \qquad (C)$$

In order to prove (C) we must choose an arbitrary configuration (P', s') and suppose that $(l := P_1, s) \rightsquigarrow (P', s')$. We now have to show that $P' = l := P_2$ and $s' = s$. The transition could only be deduced from $\rightsquigarrow_{\text{ASS}_1}$ (why!?) and so $P' = (l := P_3)$ and $s' = s$ for some P_3, where

$$(P_1, s) \rightsquigarrow (P_3, s).$$

Hence using (IH) we can deduce $P_3 = P_2$ (and $s = s$!). Thus $P' = (l := P_2)$ and we already showed that $s' = s$. As (P', s') was arbitrary, we have proved (C).

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

⋙ **NOTE 4.1.6** ***You may care to compare carefully the above proof with the general exposition of Rule Induction. Note that*** $(*)$ ***is***

$$\forall \underbrace{((P, s), (P', s'))}_{i} \in \underbrace{\rightsquigarrow}_{I}, \underbrace{\phi(((P, s), (P', s')))}_{\phi(i)}$$

Definitions 4.1.7 Let us define $V ::= \underline{c} \mid \mathsf{skip}$. One can see from the rules which define $\rightsquigarrow$ that for configurations of the form (V, s) there is no configuration (P', s') for which $(V, s) \rightsquigarrow (P', s')$. These (V, s) configurations have special significance, and will be called **terminal**. Think of them as "results and a final state" after the successful complete execution of a program. We say that any configuration (P, s) is **stuck** if it is non-terminal and there is no configuration (P', s') for which $(P, s) \rightsquigarrow (P', s')$. The configuration $(l, \langle l' \mapsto 2 \rangle)$ is stuck.

Because the transition relation is *deterministic* (Theorem 4.1.5), given any configuration (P, s) there is a *unique* sequence of transitions

$$(P, s) = (P_1, s_1) \rightsquigarrow (P_2, s_2) \rightsquigarrow \ldots$$

An **infinite transition sequence** for a configuration (P, s) takes the form

$$(P, s) = (P_1, s_1) \rightsquigarrow (P_2, s_2) \rightsquigarrow \ldots \rightsquigarrow (P_i, s_i) \rightsquigarrow \ldots$$

where no configuration (P_i, s_i) is either terminal or stuck. We sometimes say that (P, s) **loops**. A **finite transition sequence** for a configuration (P, s) takes the form

$$(P, s) = (P_1, s_1) \rightsquigarrow (P_2, s_2) \rightsquigarrow \ldots \rightsquigarrow (P_m, s_m) \qquad (m \geq 1)$$

If (P_m, s_m) is either stuck or terminal we call the transition sequence **complete**. The length m of the complete sequence is unique by the determinacy. Finally, whenever $(P, s) \rightsquigarrow (P', s')$ we sometimes say that (P, s) **has a transition**.

Motivation 4.1.8 We shall now show a *type safety* result using the terminology built up so far. This is given in Proposition 4.1.11. This result says, in essence, that if one

takes a successfully compiled program, which for us just means shown to be typeable, and runs it in a sensible state, then "the program run will not get stuck". The sequence of transitions which models the program run will either be infinite (the program loops) or it will be finite and the final configuration will be terminal. *Make sure you understand the intuition behind the notions of terminal and stuck.*

Proposition 4.1.9 Fix $\mathcal{L}$ and let s be sensible for $\mathcal{L}$. Then if $P :: \sigma$ is any type assignment, (P, s) is not stuck.

Proof We prove $\forall\, P :: \sigma\ \boxed{(P, s) \text{ is not stuck}}$ by Rule Induction for the rules in Table 3.1.

(*Property Closure for* $::$ LOC) The configuration (l, s) is not stuck as $s(l)$ is defined because s is sensible.

(*Property Closure for* $::$ IOP) The inductive hypotheses are that neither (P_1, s) or (P_2, s) are stuck, where $P_1 :: \mathsf{int}$ and $P_2 :: \mathsf{int}$. We have to prove that $(P_1\ iop\ P_2, s)$ is not stuck, where $P_1\ iop\ P_2 :: \mathsf{int}$. The configuration is obviously non-terminal, and thus we need to show that it has a transition.

If $(P_1, s) \rightsquigarrow (P, s')$ for some P and s', $\rightsquigarrow$ OP$_1$ applies so that $(P_1\ iop\ P_2, s)$ is not stuck (check!). The other possibility is that (P_1, s) is terminal, and thus $P_1 = \underline{n}$ for some n as P_1 is of type int. In this case we examine (P_2, s) in the same way as (P_1, s). If (P_2, s) is not terminal, $(P_1\ iop\ P_2, s)$ is not stuck (why?). If (P_2, s) is terminal, we must have $P_2 = \underline{m}$. In that case $(P_1\ iop\ P_2, s)$ is $(\underline{n}\ iop\ \underline{m}, s)$ which is also not stuck.

It is an exercise to check property closure for the remaining rules. □

Proposition 4.1.10 Fix $\mathcal{L}$. If $P :: \sigma$ and s is sensible for $\mathcal{L}$, and also $(P, s) \rightsquigarrow (P', s')$, then s' is also sensible.

Proof We prove $\forall\, (P, s) \rightsquigarrow (P', s')\ \boxed{\forall \sigma.\quad (P :: \sigma \text{ and } s \text{ sensible})\ \textit{implies}\ s' \text{ sensible}}$ by rule induction for $\rightsquigarrow$. The details are an exercise. □

Proposition 4.1.11 Let $\mathcal{L}$ be a location environment, let $P :: \sigma$, and let s be sensible for $\mathcal{L}$. If there is a configuration (P', s') for which $(P, s) \rightsquigarrow^* (P', s')$, then (P', s') cannot be stuck (but might be terminal). Thus $\mathbb{IMP}$ is *type safe*.

Proof This follows easily from Propositions 4.1.9 and 4.1.10. Exercise: check this. □

Example 4.1.12 Let P be $\mathsf{while}\ \underline{T}\ \mathsf{do\ skip}$. Then we have

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

and this cycle repeats forever. Thus (P, s) has an infinite transition sequence.

4.2 An Evaluation Relation

Motivation 4.2.1 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 $(P, s) \Downarrow (\underline{n}, s)$. The idea is that such an assertion corresponds to the configuration (P, s) making a finite number of transition steps to the configuration $(\underline{n}, s)$. A similar idea applies to Boolean expressions and commands. In Theorem 4.3.2, we clarify these intuitive ideas precisely.

Definitions 4.2.2 We shall inductively define the set $\Downarrow$, where

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

by the rules in Table 4.2.

Examples 4.2.3

(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 4.3. 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.

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

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

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

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

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

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

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

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

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

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

Table 4.2: 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 4.3: An Example Deduction of an Evaluation

(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?

4.3 A Mutual Correctness Proof

Motivation 4.3.1 We shall now show that the transition and evaluation semantics are equivalent, in the sense that if a configuration runs in a finite number of transition steps to a terminal configuration, then the configuration also evaluates to the same terminal configuration.

Theorem 4.3.2 For any configuration $(P\,,s)$ and terminal configuration $(V\,,s')$,

$$(P\,,s) \rightsquigarrow^* (V\,,s') \ \textit{iff}\ \ (P\,,s) \Downarrow (V\,,s')$$

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

Proof We break the proof into three parts:

(a) Prove $(P\,,s) \Downarrow (V\,,s')$ *implies* $(P\,,s) \rightsquigarrow^* (V\,,s')$ by Rule Induction.

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

$$(P\,,s) \rightsquigarrow (P'\,,s') \quad \textit{and} \quad (P'\,,s') \Downarrow (V\,,s'') \quad \textit{implies} \quad (P\,,s) \Downarrow (V\,,s'')$$

(c) Use (b) to deduce $(P\,,s) \rightsquigarrow^* (V\,,s')$ *implies* $(P\,,s) \Downarrow (V\,,s')$.

(a) We shall prove by Rule Induction that

$$\forall (P\,,s) \Downarrow (V\,,s') \quad \boxed{(P\,,s) \rightsquigarrow^* (V\,,s')}$$

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

$$(P_1\,,s_1) \rightsquigarrow^* (\underline{T}\,,s_1) \tag{$H1$}$$

$$(P_2\,,s_1) \rightsquigarrow^* (\mathsf{skip}\,,s_2) \tag{$H2$}$$

$$(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,s_2) \rightsquigarrow^* (\mathsf{skip}\,,s_3) \tag{$H3$}$$

We need to prove that

$$(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,\, s_1) \rightsquigarrow^* (\mathsf{skip}\,,\, s_3) \qquad (C)$$

Let us write Q for $\mathsf{while}\ P_1\ \mathsf{do}\ P_2$. Then

$$\begin{array}{lll}
(Q\,,\, s_1) & \rightsquigarrow & (\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ ;\ Q\ \mathsf{else}\ \mathsf{skip}\,,\, s_1) \qquad (\rightsquigarrow \textsc{loop}) \\
& \rightsquigarrow^* & (\mathsf{if}\ \underline{T}\ \mathsf{then}\ P_2\ ;\ Q\ \mathsf{else}\ \mathsf{skip}\,,\, s_1) \qquad (H1) \text{ and zero or more uses of } (\rightsquigarrow \textsc{cond}_1) \\
& \rightsquigarrow & (P_2\ ;\ Q\,,\, s_1) \qquad (\rightsquigarrow \textsc{cond}_2) \\
& \rightsquigarrow^* & (\mathsf{skip}\ ;\ Q\,,\, s_2) \qquad (H2) \text{ and zero or more uses of } (\rightsquigarrow \textsc{seq}_1) \\
& \rightsquigarrow & (Q\,,\, s_2) \qquad (\rightsquigarrow \textsc{seq}_2) \\
& \rightsquigarrow^* & (\mathsf{skip}\,,\, s_3) \qquad (H3)
\end{array}$$

which proves (C). *NB: Why "zero or more" uses?*

(b) We shall prove by Rule Induction for $\rightsquigarrow$ that

$$\forall (P\,,\, s) \rightsquigarrow (P'\,,\, s'). \quad \boxed{\forall (V\,,\, s''). \ (P'\,,\, s') \Downarrow (V\,,\, s'') \ \textit{implies} \ (P\,,\, s) \Downarrow (V\,,\, s'')}$$

Let us just consider property closure for the rule $(\rightsquigarrow \textsc{loop})$. Pick any $(V\,,\, s'')$ and suppose that

$$(\mathsf{if}\ P_1\ \mathsf{then}\ (P_2\ ;\ Q)\ \mathsf{else}\ \mathsf{skip}\,,\, s) \Downarrow (V\,,\, s'') \tag{1}$$

We need to show that

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

But (1) can hold only if it has been deduced either from $(\Downarrow \textsc{cond}_1)$ or $(\Downarrow \textsc{cond}_2)$. In either case V must be skip. We consider the two cases:

(*Case* $(\Downarrow \textsc{cond}_1)$): (1) was deduced from the hypotheses

$$(P_1\,,\, s) \Downarrow (\underline{T}\,,\, s) \tag{3}$$

and

$$(P_2\ ;\ Q\,,\, s) \Downarrow (\mathsf{skip}\,,\, s'') \tag{4}$$

where the latter assertion is deduced using $(\Downarrow \textsc{seq})$ from the hypotheses

$$(P_2\,,\, s) \Downarrow (\mathsf{skip}\,,\, s') \tag{5}$$

and

$$(Q\,,\, s') \Downarrow (\mathsf{skip}\,,\, s'') \tag{6}$$

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

(*Case* ($\Downarrow$ COND$_2$)): (1) was deduced from the hypotheses

$$(P_1, s) \Downarrow (\underline{F}, s) \tag{7}$$

and

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

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

(c) Exercise: complete the final proof step.

□

Example 4.3.3 See Theorem 4.3.2, part (b). We verify property closure for the rule $\rightsquigarrow$ COND$_1$.

The induction hypothesis IH is

$$\forall (V, s'). \quad (P', s) \Downarrow (V, s') \ \textit{implies} \ (P, s) \Downarrow (V, s')$$

We need to prove that

$$\forall (V, s'). \quad (\mathsf{if}\ P'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, s) \Downarrow (V, s') \ \textit{implies} \ (\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, s) \Downarrow (V, s')$$

Let (V, s') denote any terminal configuration, and suppose that

$$(\mathsf{if}\ P'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, s) \Downarrow (V, s')$$

This must have been deduced from either $\Downarrow$ COND$_1$ or $\Downarrow$ COND$_2$. We give the details for $\Downarrow$ COND$_1$. In this case, we must have $(P', s) \Downarrow (\underline{T}, s)$ and $(P_1, s) \Downarrow (V, s')$. Thus by IH, $(P, s) \Downarrow (\underline{T}, s)$, and using $\Downarrow$ COND$_1$ again we get

$$(\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, s) \Downarrow (V, s')$$

as required. You can fill in the details for the other case $\Downarrow$ COND$_2$.

5

The CSS Machine

5.1 Architecture of the CSS Machine

Motivation 5.1.1 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 5.1.2 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:

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

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

$$\sigma ::= \centerdot \mid \underline{c} \mid \underline{c} : \sigma$$

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

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

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

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

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

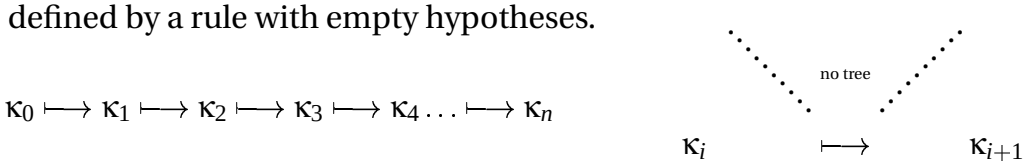
that is, every rule has no hypotheses. We call such a binary relation as $\longmapsto$ which is inductively defined by rules with no hypotheses a **re-write** relation. The CSS re-writes are defined in Table 5.1, where each rule R is written

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

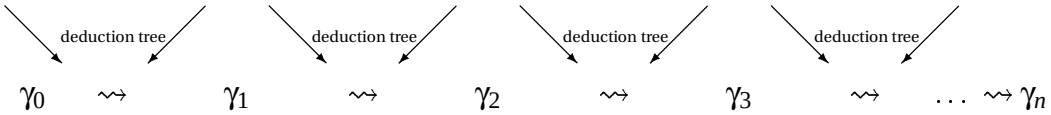
PUSH($\underline{c}$) : C	σ	s	$\longmapsto$	C	$\underline{c} : \sigma$	s
FETCH(l) : C	σ	s	$\longmapsto$	C	$s(\underline{l}) : \sigma$	s
OP(op) : C	$\underline{n_1} : \underline{n_2} : \sigma$	s	$\longmapsto$	C	$n_1\ op\ \underline{n_2} : \sigma$	s
SKIP : C	σ	s	$\longmapsto$	C	σ	s
STO(l) : C	$\underline{c}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

Table 5.1: The CSS Re-Writes

Remark 5.1.3 You may like to compare such re-write rules with the transition steps of $\mathbb{IMP}$ which we met in Chapter 4. They are similar, except that any individual re-write does not require justifying via a deduction tree, because by definition a re-write step is defined by a rule with empty hypotheses.



Rewrite Rules (Abstract Machine)



Transition Semantics

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

Motivation 5.1.4 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 $[\![-]\!]$:*Exp* $\rightarrow$ *CSScodes* which takes a CSS program expression and turns it into CSS code.

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

Table 5.2: Compiling $\mathbb{IMP}$ into CSS Code

Definitions 5.1.5 The function $[\![-]\!]: \mathit{Exp} \to \mathit{CSScodes}$ is specified by the clauses in Table 5.2.

5.2 Correctness of the CSS Machine

Motivation 5.2.1 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 4, the result matches that of the CSS machine, and vice versa. We make this precise in the following theorem:

Theorem 5.2.2 For all $n \in \mathbb{Z}$, $b \in \mathbb{B}$, $P_1 :: \mathsf{int}$, $P_2 :: \mathsf{bool}$, $P_3 :: \mathsf{cmd}$ and $s, s_1, s_2 \in \mathit{States}$ we have

$$
\begin{array}{rll}
(P_1\,, s) \Downarrow (\underline{n}, s) & \mathit{iff} & [\![P_1]\!] \,|\, \cdot \,|\, s \longmapsto^t \cdot \,|\, \underline{n} \,|\, s \\
(P_2\,, s) \Downarrow (\underline{b}, s) & \mathit{iff} & [\![P_2]\!] \,|\, \cdot \,|\, s \longmapsto^t \cdot \,|\, \underline{b} \,|\, s \\
(P_3\,, s_1) \Downarrow (\mathsf{skip}\,, s_2) & \mathit{iff} & [\![P_3]\!] \,|\, \cdot \,|\, s_1 \longmapsto^t \cdot \,|\, \cdot \,|\, s_2
\end{array}
$$

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

Proof See Appendix A □

5.3 CSS Executions

Examples 5.3.1

(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

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

(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

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

Then

$$\begin{array}{rcl}
\boxed{\mathsf{PUSH}(\underline{0}) : \mathsf{FETCH}(l) : \geq : \mathsf{BR}(\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l), \mathsf{SKIP}) \;\|\; \centerdot \;\|\; s} & & \\
\longmapsto & \boxed{\mathsf{FETCH}(l) : \geq : \mathsf{BR}(\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l), \mathsf{SKIP}) \;\|\; \underline{0} \;\|\; s} & \\
\longmapsto & \boxed{\geq : \mathsf{BR}(\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l), \mathsf{SKIP}) \;\|\; \underline{1} : \underline{0} \;\|\; s} & \\
\longmapsto & \boxed{\mathsf{BR}(\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l), \mathsf{SKIP}) \;\|\; \underline{T} \;\|\; s} & \\
\longmapsto & \boxed{\mathsf{PUSH}(\underline{1}) : \mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l) \;\|\; \centerdot \;\|\; s} & \\
\longmapsto & \boxed{\mathsf{FETCH}(l) : \mathsf{OP}(-) : \mathsf{STO}(l) \;\|\; \underline{1} \;\|\; s} & \\
\longmapsto & \boxed{\mathsf{OP}(-) : \mathsf{STO}(l) \;\|\; \underline{1} : \underline{1} \;\|\; s} & \\
\longmapsto & \boxed{\mathsf{STO}(l) \;\|\; \underline{0} \;\|\; s} & \\
\longmapsto & \boxed{\centerdot \;\|\; \centerdot \;\|\; s\{l \mapsto 0\}} &
\end{array}$$

6

Typing of Functional Languages

6.1 Introduction

Motivation 6.1.1 In this chapter we turn our attention to functional programming languages. Such languages provide a syntax of expressions in which one can write down functions directly, 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 6.1 gives an example of an *identifier environment*. This gives the types of various constant and function identifiers. Figure 6.2 declares the meanings of the identifiers. Most of this should be clear, but we give a few explanatory comments. (We shall assume that readers have some familiarity with (the datatypes of) functions, pairs and lists. If not, consult the course notes for MC 103, 104 and 208[1].) If σ_1 and σ_2 are any types (for example, base types int or bool) then $[\sigma_1]$ is the type of σ_1-lists, and (σ_1, σ_2) is the type of pairs whose first element is of type σ_1 and second of type σ_2. 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. Recall also that all well-formed lists are built up from nil using :. Thus, for example, $(\underline{2} : \underline{4} : \mathsf{nil}, T) :: ([\mathsf{int}], \mathsf{bool})$. 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. Look at the function `g`. Its type is given as `Int -> Int -> Int` which is sugar for `Int -> (Int -> Int)`. If `4 :: Int`, then we can apply `g` to this integer to get `g 4 :: Int -> Int`. `g 4` is a itself a function which takes integers as inputs, and yields an integer output. Thus `(g 4) 6` evaluates to an integer, namely `4+6 = 10`. As a further example, `(h 4) 6 :: Int -> Int`, and `((h 4) 6) 1 = 4 + 6 + 1 = 11`.

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

Motivation 6.2.1 We begin by defining the types and expressions of a simple language called $\mathbb{FUN}^e$. 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

[1] It is not essential to have taken MC 208

```
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
p :: (Int,Int)
fst :: (Int,Int) -> 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 6.1: 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
p = (3,4)                           -- definition of constant p
fst (x,y) = x
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 6.2: An example of an Identifier Declaration

as output to the programmer.

A functional language contains expressions whose abstract syntax trees are of the form $\mathsf{ap}(E_1,E_2)$. The idea is that E_1 denotes a function, and that E_2 is the input or argument for the function. We say that the expression $\mathsf{ap}(E_1,E_2)$ is E_1 *applied to* E_2. As we have seen, its sugared form is E_1E_2.

Definitions 6.2.2 The types of the language $\mathbb{FUN}^e$ are (the syntax trees) given inductively (exercise: what are the rules?) by the grammar

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

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

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

for

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

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

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

$$\mathsf{F}\,xy = x+y \quad \text{in } \mathbb{FUN}^e \qquad \text{to} \qquad \texttt{f x y = x+y} \quad \text{in Haskell.}$$

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

E	::=	x	variables
		$\underline{c}$	integer or Boolean constant
		K	constant identifier
		F	function identifier
		$E_1\ iop\ E_2$	integer valued operator on integers
		$E_1\ bop\ E_2$	Boolean valued operator on integers
		$\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3$	conditional
		(E_1,E_2)	pairing
		$\mathsf{fst}(E)$	first projection
		$\mathsf{snd}(E)$	second projection
		E_1E_2	function application
		nil_σ	empty list
		$\mathsf{hd}(E)$	head of list
		$\mathsf{tl}(E)$	tail of list
		$E_1 : E_2$	cons for lists
		$\mathsf{elist}(E)$	Boolean test for empty list

Examples 6.2.3

(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 6.2.4 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_2E_3, and then apply E_1 to the latter expression, we write this as $E_1(E_2E_3)$.

• $E_1E_2E_3\ldots E_n$ is shorthand for $(\ldots((E_1E_2)E_3)\ldots)E_n$. We say that application **associates** to the left. For example, $E_1E_2E_3$ is short for $(E_1E_2)E_3$. Note that if we made the tree structure of applications explicit, rather than using the sugared notation EE' instead of, say, $\mathsf{ap}(E,E')$ where ap is a tree constructor, then $(E_1E_2)E_3$ 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 6.2.5 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 $E_1,\ldots,E_n$ are expressions, then $E[E_1,\ldots,E_n/x_1,\ldots,x_n]$ 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 $E[E_1,\ldots,E_n/x_1,\ldots,x_n]$ is indeed an expression).

Examples 6.2.6 Examples of expressions and substitutions are

(1) x

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

(3) $f\,(gy)$ where $f, g, y \in \mathit{Var}$.

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

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

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

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

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

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

Motivation 6.2.7 Recall discussions about type checking and inference from Chapter 3. 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 6.2.8 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 $\Gamma, \Gamma' \stackrel{\text{def}}{=} \Gamma \cup \Gamma'$ and $\Gamma, x :: \sigma \stackrel{\text{def}}{=} \Gamma \cup \{x :: \sigma\}$.

An **identifier type** is a type of the form $\sigma_1 \to \sigma_2 \to \sigma_3 \to \ldots \to \sigma_k \to \sigma$ where k is a natural number and σ is **NOT a function type**. If k is 0 then the type is simply σ. You should think of such an identifier type as typing information for an identifier. If $k = 0$ then the identifier is called a **constant**. If $k > 0$ then the identifier is called a **function**; the identifier will represent a function that takes a *maximum* of k inputs with types σ_i and gives an output of type σ. Of course the identifier can take less than k inputs to yield another function–compare `(h 4) 5` in Figure 6.2 where `h` has $k = 3$. 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

$$I = \mathsf{I}_1 :: \iota_1, \ldots, \mathsf{I}_m :: \iota_m.$$

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 6.2.9 A simple example of an identifier environment is

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

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 6.2.10 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 6.2.11 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 6.1.

Remark 6.2.12 Let I be an identifier environment. Note that if $\Gamma \vdash E :: \sigma$, then I and Γ may contain identifiers and variables which do not appear in E. For example if $I \stackrel{\text{def}}{=} \mathsf{F} :: \mathsf{int} \to \mathsf{int}$ then

$$x :: \mathsf{int}, y :: \mathsf{int}, z :: \mathsf{int} \vdash \mathsf{F}x :: \mathsf{int}$$

is a valid type assignment. The motto is "just because we declared some variables or identifiers, does not mean we need to program with them".

Note also that the second part of the next proposition says that "given any type assignment, we can add identifiers to the identifier environment, and variables to the context, without changing the type of E." Sometimes this is called **weakening**.

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

$$\frac{}{\Gamma \vdash \underline{T} :: \mathsf{bool}} :: \text{TRUE} \qquad \frac{}{\Gamma \vdash \underline{F} :: \mathsf{bool}} :: \text{FALSE}$$

$$\frac{\Gamma \vdash E_1 :: \mathsf{int} \quad \Gamma \vdash E_2 :: \mathsf{int}}{\Gamma \vdash E_1\ iop\ E_2 :: \mathsf{int}} :: \text{OP}_1$$

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

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

$$\frac{\Gamma \vdash E_1 :: \sigma_2 \to \sigma_1 \quad \Gamma \vdash E_2 :: \sigma_2}{\Gamma \vdash E_1 E_2 :: \sigma_1} :: \text{AP}$$

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

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

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

$$\frac{}{\Gamma \vdash \mathsf{nil}_\sigma :: [\sigma]} :: \text{NIL} \qquad \frac{\Gamma \vdash E_1 :: \sigma \quad \Gamma \vdash E_2 :: [\sigma]}{\Gamma \vdash E_1 : E_2 :: [\sigma]} :: \text{CONS}$$

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

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

Proposition 6.2.13 Let I be an identifier environment. If $\Gamma \vdash E :: \sigma$, then the identifiers which appear in E appear in I, and the variables which appear in E appear in Γ.

Suppose that I, I' is an identifier environment, that Γ, Γ' is a context, and that $\Gamma \vdash E :: \sigma$ given just I. Then in fact $\Gamma, \Gamma' \vdash E :: \sigma$ is also a derivable type assignment for I, I'.

Proof Follows by a simple Rule Induction for Table 6.1. Exercise! □

Proposition 6.2.14 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**—see Chapter 8.

Proof We can prove this using Rule Induction for Table 6.1. In fact we verify that

$$\forall\, (\Gamma \vdash E :: \sigma_1). \quad \boxed{\forall \sigma_2. \quad (\Gamma \vdash E :: \sigma_2 \ \textit{implies}\ \sigma_1 = \sigma_2).}$$

We check property closure for the rule HD: The inductive hypothesis is that

$$\forall \sigma_2, \quad (\Gamma \vdash E :: \sigma_2 \ \textit{implies}\ [\sigma] = \sigma_2)$$

where $\Gamma \vdash E :: [\sigma]$. We wish to prove that

$$\forall \sigma_2, \quad (\Gamma \vdash \mathsf{hd}(E) :: \sigma_2 \ \textit{implies}\ \sigma = \sigma_2) \quad (\dagger)$$

where $\Gamma \vdash \mathsf{hd}(E) :: \sigma$.

Let τ be arbitrary, where $\Gamma \vdash \mathsf{hd}(E) :: \tau$. Then we must have $\Gamma \vdash E :: [\tau]$. From the inductive hypothesis we see that $[\sigma] = [\tau]$. It follows that $\sigma = \tau$ as required. As τ was arbitrary, $(\dagger)$ is proved.

Property closure of the remaining rules is left as an exercise. □

Examples 6.2.15

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

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

where $\mathcal{D}$ is

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

(2) With I as in Example 6.2.9, 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

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

(5) (See Proposition 6.2.13.) Suppose that $\Gamma \vdash E :: \sigma$. Let $var(E)$ be the set of variables which occur in E, and $var(\Gamma)$ be the set of variables which are declared in the context Γ. We explain carefully, but informally, why it is always the case that $var(E) \subseteq var(\Gamma)$.

The following amounts to an informal description of a proof by rule induction that

$$\forall\, \Gamma \vdash E :: \sigma \quad \boxed{var(E) \subseteq var(\Gamma)}$$

Base rules: In $::$ VAR, it is a condition that if $\Gamma \vdash x :: \sigma$, then $x :: \sigma$ appears in Γ. Thus certainly $var(x) = \{x\} \subseteq var(\Gamma)$. In each rule which introduces either a constant or an identifier, clearly $var(\underline{c}) = \varnothing$ and $var(\mathsf{I}) = \varnothing$, and of course $\varnothing \subseteq var(\Gamma)$ for any context Γ. In all other rules, the inductive hypotheses would state that for any $\Gamma \vdash E_i :: \sigma_i$ we have $var(E_i) \subseteq \Gamma$ where $i \in \{1 \ldots n\}$ and n is the number of type assignments appearing as assumptions in the rule. The variables appearing in the expression in the conclusion of the rule must be $var(E_1) \cup var(E_2) \cup \ldots \cup var(E_n)$ and appealing to the inductive hypotheses this is a subset of $var(\Gamma)$, as required.

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

Motivation 6.3.1 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 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 $\mathsf{F}x = E$ where E is an expression which denotes "the result" of applying F to x. In $\mathbb{FUN}^e$, we are able to specify

statements such as $\mathsf{F}x = E$ and $\mathsf{K} = E'$ 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 6.3.2 Let $I = \mathsf{I}_1 :: \iota_1, \ldots, \mathsf{I}_m :: \iota_m$ be a given, fixed, identifier environment for which

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

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

$$\begin{array}{rcl} \mathsf{I}_1 x_{11} \ldots x_{1k_1} & = & E_{\mathsf{I}_1} \\ \mathsf{I}_2 x_{21} \ldots x_{2k_2} & = & E_{\mathsf{I}_2} \\ & \vdots & \\ \mathsf{I}_j x_{j1} \ldots x_{jk_j} & = & E_{\mathsf{I}_j} \\ & \vdots & \\ \mathsf{I}_m x_{m1} \ldots x_{mk_m} & = & E_{\mathsf{I}_m} \end{array}$$

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

$$\varnothing \vdash P :: \sigma$$

and the declarations in dec_I satisfy

$$\begin{array}{c} x_{11} :: \sigma_{11}, \ldots, x_{1k_1} :: \sigma_{1k_1} \vdash E_{\mathsf{I}_1} :: \sigma_1 \\ x_{21} :: \sigma_{21}, \ldots, x_{2k_2} :: \sigma_{2k_2} \vdash E_{\mathsf{I}_2} :: \sigma_2 \\ \vdots \\ x_{j1} :: \sigma_{j1}, \ldots, x_{jk_j} :: \sigma_{jk_j} \vdash E_{\mathsf{I}_j} :: \sigma_j \\ \vdots \\ x_{m1} :: \sigma_{m1}, \ldots, x_{mk_m} :: \sigma_{mk_m} \vdash E_{\mathsf{I}_m} :: \sigma_m \end{array}$$

Note that the data which are *specified* in dec_I just consist of the declarations $\mathsf{I}_j\vec{x} = E_{\mathsf{I}_j}$; 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—recall page 19. We shall sometimes abbreviate the jth type assignment to $\Gamma_{\mathsf{I}_j} \vdash E_{\mathsf{I}_j} :: \sigma_j$. We call the expression E_{I_j} the **definitional body** of I_j. Note that the type assignments force each of the variables in $\{x_{j1}, \ldots, x_{jk_j}\}$ 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). The program type will normally be *inferred* from the given dec_I and P—see page 23.

Examples 6.3.3

(1) Let $I = \mathsf{I}_1 :: [\mathsf{int}] \to \mathsf{int} \to \mathsf{int}, \mathsf{I}_2 :: \mathsf{int} \to \mathsf{int}, \mathsf{I}_3 :: \mathsf{bool}$. Then an example of an identifier declaration dec_I is

$$\begin{array}{rcl} \mathsf{I}_1 x_{11} x_{12} & = & \mathsf{hd}(\mathsf{tl}(\mathsf{tl}(x_{11}))) + \mathsf{I}_2 x_{12} \\ \mathsf{I}_2 x_{21} & = & x_{21} * x_{21} \\ \mathsf{I}_3 & = & \underline{T} \end{array}$$

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 $\mathsf{F} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$. Then we have a declaration dec_I

$$\mathsf{F}\, x\, y\, z = x + y + z$$

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

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

(4)

$$\left.\begin{array}{rcl} \mathsf{F}_1\, x\, y\, z & = & \mathsf{if}\ x \leq \underline{1}\ \mathsf{then}\ y\ \mathsf{else}\ z \\ \mathsf{F}_2\, x & = & \mathsf{F}_1\, x\, \underline{1}\, (x * \mathsf{F}_2\,(x - \underline{1})) \end{array}\right\} \quad \textit{in}\ \ \mathsf{F}_2\,\underline{4}$$

(5)

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

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) $\mathsf{F}\,x = \mathsf{F}\,x \quad \mathit{in}\ \ \mathsf{F}\,(\underline{3} : \mathsf{nil}_{\mathsf{int}})$ is a program which does not evaluate to a value; the program *loops*—see Chapter 7.

7

Operational Semantics of Functional Languages

7.1 Operational Semantics for $\mathbb{FUN}^e$

Motivation 7.1.1 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 $\mathsf{F}\,x\,y = x+y$. We would expect $\mathsf{F}\,(\underline{2}*\underline{3})\,(\underline{4}*\underline{5}) \Downarrow^e \underline{26}$. But how do we reach this value? The first possibility is *call-by-value* evaluation. First we calculate the first argument to get $\mathsf{F}\,\underline{6}\,(\underline{4}*\underline{5})$, then the second to get $\mathsf{F}\,\underline{6}\,\underline{20}$. Having got the *values* of the function *arguments*, we *call* the function to get $\underline{6}+\underline{20}$, which evaluates to $\underline{26}$. The second possibility is that the function is evaluated first to give $(\underline{2}*\underline{3})+(\underline{4}*\underline{5})$ and then we continue to get $\underline{6}+(\underline{4}*\underline{5})$ and $\underline{6}+\underline{20}$. This is called *call-by-name* evaluation, and is dealt with in Section 7.2.

In evaluating a function application $\mathsf{F}P_1P_2$ we first compute values for P_1 and P_2, say V_1 and V_2, and then evaluate $\mathsf{F}V_1V_2$. In evaluating a pair (P_1,P_2), we compute values for P_1 and P_2, say V_1 and V_2, giving a final value of (V_1,V_2). A similar idea applies to lists.

Definitions 7.1.2 Let dec_I be a identifier declaration. A **value expression** is any expression V which can be produced by the following grammar

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

where c is any Boolean or integer, σ is any type, and $\vec{V}$ abbreviates $V_1\,V_2\,\ldots\,V_{l-1}\,V_l$ where $0 \leq l < k$, and k 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 l is *strictly* less than k, and that if $k=1$ then $\mathsf{F}\,\vec{V}$ denotes F.

Definitions 7.1.3 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 7.1.

Remark 7.1.4 You may find the definition of $\mathsf{F}\vec{V}$ as a value expression rather odd. In fact, there is good reason for the definition. The basic idea behind the definition of a value is that "values are those expressions which are as fully evaluated as possible, according to the call-by-value execution strategy". This explains why $\mathsf{F}\vec{V}$ is indeed a value expression; a small example will clarify:

Suppose that

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

and that P_1 and P_2 are integer programs, which compute to the values $\underline{n_1}$ and $\underline{n_2}$. Then $\mathsf{F}\,P_1$ is not a value, because the language is eager. It will evaluate to $\mathsf{F}\,\underline{n_1}$. But this latter expression cannot be evaluated any further—informally, the function F cannot itself be called until it is applied to three integer arguments. Thus $\mathsf{F}\,\underline{n_1}$ is a value. Giving it the argument P_2, we have a program $\mathsf{F}\,\underline{n_1}\,P_2$ which evaluates to the value $\mathsf{F}\,\underline{n_1}\,\underline{n_2}$. Again, we have a value, as the expression cannot be computed any further. Finally, however, we can supply a third argument to $\mathsf{F}\,\underline{n_1}\,\underline{n_2}$ giving $\mathsf{F}\,\underline{n_1}\,\underline{n_2}\,P_3$. This evaluates to $\mathsf{F}\,\underline{n_1}\,\underline{n_2}\,\underline{n_3}$, and at last F has its full quota of three arguments—thus the latter expression is not a value as we can now compute the function F using rule $\Downarrow^e$ FID.

Examples 7.1.5

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

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

We prove that $\mathsf{G}\,\mathsf{K} \Downarrow^e \underline{6}$. 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 $\mathsf{G} \Downarrow^e \mathsf{F}\,\vec{V}$ (easy: take F to be G with the $\vec{V}$ "empty"), that $\mathsf{K} \Downarrow^e V$ for some V which is easy, as we can guess that V must be $\underline{3}$, and that $\mathsf{G}\,V \Downarrow^e \underline{6}$. The latter must be a conclusion of 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

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

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

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

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

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

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

$$\frac{\begin{cases} P_1 \Downarrow^e \mathsf{F}\,\vec{V} \quad P_2 \Downarrow^e V_2 \quad \mathsf{F}\,\vec{V}\,V_2 \Downarrow^e V \\ \text{where either } P_1 \text{ or } P_2 \text{ is not a value} \end{cases}}{P_1 P_2 \Downarrow^e V}\ \Downarrow^e_{\text{AP}}$$

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

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

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

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

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

Table 7.1: Evaluation Relation $P \Downarrow^e V$ in $\mathbb{FUN}^e$

(2) Suppose that

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

Then the expressions $\mathsf{F}\,\underline{2}$ and $\mathsf{F}\,\underline{2}\,\underline{3}$ are (programs and) values. $\mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1})$ is a program, but not a value: the function F takes a maximum of three inputs, and can now be evaluated. Note that $\mathsf{F}\,\underline{2}\,\underline{3}$ is sugar for $(\mathsf{F}\,\underline{2})\,\underline{3}$ and that $\mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1})$ is sugar for the expression $((\mathsf{F}\,\underline{2})\,\underline{3})\,(\underline{4}+\underline{1})$. In Definitions 7.1.2, $k=3$, and in $\mathsf{F}\,\underline{2}\,\underline{3}$ we have $\vec{V} = \underline{2}\,\underline{3}$ and $l = 2 < 3$.

We can prove that

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

where $\mathsf{F}xyz = x+y+z$ as follows:

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

where T is the tree

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

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

(3) Let $\mathsf{G}x = x+2$.

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

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

(4) Consider the program $\mathsf{large}\,x = \underline{1} + \mathsf{large}\,x \quad \textit{in}\ \mathsf{fst}((\underline{3}, \mathsf{large}\,\underline{0}))$. If we attempt to prove

that this program evaluates to a value V, the deduction tree takes the form:

$$\dfrac{\dfrac{\underline{3} \Downarrow^e V \qquad \dfrac{\dfrac{\vdots}{\underline{1} + \mathsf{large}\,\underline{0} \Downarrow^e V'}}{\mathsf{large}\,\underline{0} \Downarrow^e V'}}{(\underline{3}, \mathsf{large}\,\underline{0}) \Downarrow^e (V, V')}}{\mathsf{fst}((\underline{3}, \mathsf{large}\,\underline{0})) \Downarrow^e V}$$

for some value V'. It is easy to see that no finite deduction exists, and so there is no value V for which $\mathsf{fst}((\underline{3}, \mathsf{large}\,\underline{0})) \Downarrow^e V$. Informally, we cannot take the first component of the pair without first evaluating its sub-expressions, as $\mathbb{FUN}^e$ is eager. Compare this evaluation to the execution of the same program in the lazy $\mathbb{FUN}^l$ on page 61.

Motivation 7.1.6 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 7.1.7 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

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

To do this we apply rule induction: we have to verify property closure for the rules in Figure 7.1.

(*Closure under* $\textsc{cond}_2$): The inductive hypotheses are

H1 for all V', if $P_1 \Downarrow^e V'$ then $\underline{F} = V'$, and

H2 for all V', if $P_3 \Downarrow^e V'$ then $V = V'$.

We have to prove that

C for any V', if $\quad \mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V' \quad$ then $V = V'$.

Pick an arbitrary V' for which $\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V'$—(*). Now (*) could be deduced from an application of either COND_1 or COND_2. If it were the former, then $P_1 \Downarrow^e \underline{T}$. So using **H1**, we would have $\underline{F} = \underline{T}$, a contradiction. Hence (*) must be a conclusion to an instance of COND_2, say

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

Hence $P_3 \Downarrow^e V'$ for some program P_3. But using **H2**, it follows that $V = V'$ as required. □

Motivation 7.1.8 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 7.1.10. However, the proof will require the following lemma in order to deal with the rule $\Downarrow^e$ FID.

Lemma 7.1.9 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

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

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

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

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

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

The proof is an exercise. □

7.2 Operational Semantics for $\mathbb{FUN}^l$

Definitions 7.2.1 The language $\mathbb{FUN}^l$ is identical to $\mathbb{FUN}^e$, except that it has a *lazy* operational semantics. We explain below exactly what this means. The expressions, contexts, identifier environments, identifier declarations and type assignments of $\mathbb{FUN}^l$ are exactly the same as for $\mathbb{FUN}^e$. In a nutshell, we can say that the definition of the relationships $\Gamma \vdash E :: \sigma$ for $\mathbb{FUN}^l$ is specified by the rules in Table 6.1.

Motivation 7.2.2 The operational semantics of $\mathbb{FUN}^l$ is **lazy** or **call-by-name**. This means that certain expressions can be evaluated *before* their subexpressions are computed. This method of computation applies to identifiers, pairs and lists. This has the advantage that if any of the subexpressions are not required in the computation of the expression, then no time is lost evaluating the subexpression. *Lazy* refers to the fact that the language does not bother to compute subexpressions if it does not need to. The definition of *program* is the same as before. We shall need a different notion of value in $\mathbb{FUN}^l$. We give the new definition of value, and then give the lazy operational semantics of $\mathbb{FUN}^l$.

Definitions 7.2.3 Let dec_I be a identifier declaration. A **value expression** is any expression V which can be produced by the following grammar

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

where c is any Boolean or integer, σ is any type, P is any program expression, and $\vec{P}$ is of the form $P_1 P_2 \ldots P_{l-1} P_l$ where $0 \leq l < k$, and k is the maximum number of inputs taken by F. *Note* that l is *strictly* less than k. A **value** is any value expression for which $dec_I \quad in \;\; V$ is a valid $\mathbb{FUN}^l$ program.

Definitions 7.2.4 We shall define an **evaluation relation** whose judgements will take the form

$$P \Downarrow^l V$$

where P and V are respectively a program expression and value expression whose identifiers appear in an identifier declaration dec_I. The rules for inductively generating these judgements are given by the rules in Table 7.2.

Motivation 7.2.5 You should note that the rules in Figure 7.2 yield a **lazy** operational semantics for functions and lists. In general, *lazy* means that "subterms of programs are only computed if absolutely necessary". For a general program of the form

$$P = C(E_1, E_2, \ldots, E_n)$$

where C is a program constructor, we only evaluate those E_i to values necessary for the evaluation of P. We illustrate by example:

Consider PP'. Let us write this as the finite tree $\mathsf{ap}(P,P')$ (as originally defined) where ap is the tree constructor. In order to evaluate $\mathsf{ap}(P,P')$, we *must* evaluate P to a function expression, say $\mathsf{F}\vec{P}$, but we do not evaluate the arguments $\vec{P}$. But now we are lazy!! We do not bother to evaluate P' before passing it to $\mathsf{F}\vec{P}$. Either $(\mathsf{F}\vec{P})P'$ is a value, or, if not, the next step of the computation is to evaluate $E_{\mathsf{F}}[\vec{P},P'/\vec{x},x]$. If this expression evaluates to a value, say V, then so too does $\mathsf{ap}(P,P')$. Now look at rule AP, and see how it captures our intended operational semantics!

The same idea applies to lists. Look at rules HD and TL.Consider $P : P'$, that is $\mathsf{cons}(P,P')$ where cons is the program constructor. We regard this as a fully evaluated program—very lazy!! We only compute the subterms P or P' if they are extracted by taking a head or tail. Thus to evaluate $\mathsf{hd}(P)$, we first evaluate the list P to a value of the form $\mathsf{cons}(P_1,P_2)$, but then we only bother (lazy) to evaluate P_1 to a value, say V. Thus $\mathsf{hd}(P)$ evaluates to V, and there is no need to evaluate P_2.

Examples 7.2.6

(1) We can prove that

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

where $\mathsf{F}xyz = x+y+z$ as follows: as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{2}\Downarrow^l\underline{2}} \quad \dfrac{}{\underline{3}\Downarrow^l\underline{3}}}{\underline{2}+\underline{3}\Downarrow^l\underline{5}} \quad \dfrac{\dfrac{}{\underline{4}\Downarrow^l\underline{4}} \quad \dfrac{}{\underline{1}\Downarrow^l\underline{1}}}{\underline{4}+\underline{1}\Downarrow^l\underline{5}}}{\underline{2}+\underline{3}+(\underline{4}+\underline{1})\Downarrow^l\underline{10}}}{(x+y+z)[\underline{2},\underline{3},(\underline{4}+\underline{1})/x,y,z]\Downarrow^l\underline{10}}}{\mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1})\Downarrow^l\underline{10}}\ \Downarrow^l \text{FID}$$

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

$$\frac{P_1 \Downarrow^l \underline{T} \quad P_2 \Downarrow^l V}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^l V}\ \Downarrow^l_{\text{COND}_1} \qquad \frac{P_1 \Downarrow^l \underline{F} \quad P_3 \Downarrow^l V}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^l V}\ \Downarrow^l_{\text{COND}_2}$$

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

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

$$\frac{\left\{\begin{array}{l} P_1 \Downarrow^l \mathsf{F}\vec{P} \quad \mathsf{F}\vec{P}P_2 \Downarrow^l V \\ \text{where } P_1 \text{ is not a value} \end{array}\right.}{P_1 P_2 \Downarrow^l V}\ \Downarrow^l_{\text{AP}}$$

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

$$\frac{E_{\mathsf{K}} \Downarrow^l V}{\mathsf{K} \Downarrow^l V}\ [\mathsf{K} = E_{\mathsf{K}} \text{ declared in } dec_I]\ \Downarrow^l_{\text{CID}}$$

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

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

$$\frac{P \Downarrow^l \mathsf{nil}_\sigma}{\mathsf{elist}(P) \Downarrow^l \underline{T}}\ \Downarrow^l_{\text{ELIST}_1} \qquad \frac{P_1 \Downarrow^l P_2 : P_3}{\mathsf{elist}(P_1) \Downarrow^l \underline{F}}\ \Downarrow^l_{\text{ELIST}_2}$$

Table 7.2: Evaluation Relation $P \Downarrow^l V$ in $\mathbb{FUN}^l$

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

(2) Recall Examples 7.1.5. Consider the program $\mathsf{large}\,x = \underline{1} + \mathsf{large}\,x \quad \mathit{in}\ \mathsf{fst}((\underline{3}, \mathsf{large}\,\underline{0}))$ once again. Let us try to see if this program evaluates to a value, say V. Working the rules in Table 7.2 backwards, there must be P_1 and P_2 and rules R and R' such that we have

$$\dfrac{\dfrac{}{(\underline{3}, \mathsf{large}\,\underline{0}) \Downarrow^l (P_1, P_2)}\,R \qquad \dfrac{}{P_1 \Downarrow^l V}\,R'}{\mathsf{fst}((\underline{3}, \mathsf{large}\,\underline{0})) \Downarrow^l V}\Downarrow^l \text{FST}$$

and clearly we have a valid (finite) deduction tree when P_1 is $\underline{3}$, P_2 is $\mathsf{large}\,\underline{0}$, V is $\underline{3}$ and R and R' are both instances of $\Downarrow^l$ VAL. In the lazy language, we can extract the first component of a pair without having first to compute the second component.

(3) Let I be $\mathsf{F} :: \mathsf{int} \to [\mathsf{int}]$, and dec_I be $\mathsf{F}x = x : \mathsf{F}\,(x + \underline{2})$. Then there is a program $dec_I \quad \mathit{in}\ \mathsf{hd}(\mathsf{tl}(\mathsf{F}\,\underline{1}))$. We prove that $\mathsf{hd}(\mathsf{tl}(\mathsf{F}\,\underline{1})) \Downarrow^l \underline{3}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{1} : \mathsf{F}\,(\underline{1}+\underline{2}) \Downarrow^l \underline{1} : \mathsf{F}\,(\underline{1}+\underline{2})}\Downarrow^l \text{VAL}}{\mathsf{F}\,\underline{1} \Downarrow^l \underline{1} : \mathsf{F}\,(\underline{1}+\underline{2})}\Downarrow^l \text{FID} \qquad T}{\mathsf{tl}(\mathsf{F}\,\underline{1}) \Downarrow^l (\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2})} \qquad \dfrac{\dfrac{}{\underline{1} \Downarrow^l \underline{1}} \qquad \dfrac{}{\underline{2} \Downarrow^l \underline{2}}}{\underline{1}+\underline{2} \Downarrow^l \underline{3}}}{\mathsf{hd}(\mathsf{tl}(\mathsf{F}\,\underline{1})) \Downarrow^l \underline{3}}$$

where T is the tree

$$\dfrac{\dfrac{}{(\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2}) \Downarrow^l (\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2})}\Downarrow^l \text{VAL}}{\mathsf{F}\,(\underline{1}+\underline{2}) \Downarrow^l (\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2})}$$

It is an exercise to check this deduction tree is correct, adding in the labels for the rules. Why might we call $\mathsf{F}\,\underline{1}$ the *lazy list of odd numbers*? Try evaluating $\mathsf{F}\,\underline{1}$ using the eager semantics. What happens?

Theorem 7.2.7 $\mathbb{FUN}^l$ is monomorphic, deterministic, and satisfies subject reduction.

Proof The proofs of these facts are basically the same as for $\mathbb{FUN}^e$ and are omitted. □

Example 7.2.8 The following is a typical example of the more difficult part of an examination question:

Suppose we are given the function environment

$$I \stackrel{\text{def}}{=} \mathsf{F} :: \mathsf{int} \to \mathsf{int}, \quad \mathsf{G} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$$

and the declaration dec_I

$$\begin{array}{rcl} \mathsf{F}\,x & = & \mathsf{if}\ x == \underline{0}\ \mathsf{then}\ \underline{0}\ \mathsf{else}\ \mathsf{G}\,x\,(\underline{0}-x) \\ \mathsf{G}\ x\ y & = & \mathsf{if}\ x == \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ (\mathsf{if}\ y == \underline{0}\ \mathsf{then}\ \underline{-1}\ \mathsf{else}\ \mathsf{G}\ \ (x-\underline{1})\ \ (y-\underline{1})) \end{array}$$

1. Use Mathematical Induction to prove that

   $$\forall n \geq 1. \quad \mathsf{G}\ \underline{n}\ \underline{-n} \Downarrow^e \underline{1}$$

   Hint: You may assume that $\mathsf{G}\ \underline{1}\ \underline{-n} \Downarrow^e \underline{1}$ *for any* $n \geq 1$

2. For each $z \in \mathbb{Z}$, state what integer you think the program $\mathsf{F}\,\underline{z}$ evaluates to when $z > 0$, $z < 0$ and $z = 0$.

3. Suppose that we consider the language $\mathbb{FUN}^e$ in which the only Boolean valued arithmetic operator is $==$ (test for equality). Give a function declaration of the form

   $\mathsf{lt}\ \ x\ \ y = E_{\mathsf{lt}}$ which computes $x < y$ for x and y of type int.

Here is an answer:

1. We prove $\forall n \geq 1.\phi(n)$ where

   $$\phi(n) \quad \stackrel{\text{def}}{=} \quad \forall r \geq 1.\ \mathsf{G}\ \underline{n}\ \underline{-r} \Downarrow^e \underline{1}$$

   Note that $\phi(1)$ is the given assumption (which is easily provable!). Now let $n \geq 1$ be arbitrary; we assume $\phi(n)$ and prove $\phi(n+1)$. To show the latter, let $k \geq 1$ be arbitrary, and consider:

   $$\dfrac{\dfrac{\dfrac{}{\underline{n+1} \Downarrow^e \underline{n+1}} \quad \dfrac{}{\underline{0} \Downarrow^e \underline{0}}}{\underline{n+1} == \underline{0} \Downarrow^e \underline{F}} \quad \dfrac{\dfrac{\dfrac{}{\underline{-k} \Downarrow^e \underline{-k}} \quad \dfrac{}{\underline{0} \Downarrow^e \underline{0}}}{\underline{-k} == \underline{0} \Downarrow^e \underline{F}} \quad T}{\mathsf{if}\ \underline{-k} == \underline{0}\ \mathsf{then}\ \underline{-1}\ \mathsf{else}\ \mathsf{G}\ \ (\underline{n+1}-\underline{1})\ \ (\underline{-k}-\underline{1}) \Downarrow^e \underline{1}}}{\mathsf{if}\ \underline{n+1} == \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ (\mathsf{if}\ \underline{-k} == \underline{0}\ \mathsf{then}\ \underline{-1}\ \mathsf{else}\ \mathsf{G}\ \ (\underline{n+1}-\underline{1})\ \ (\underline{-k}-\underline{1})) \Downarrow^e \underline{1}}}{\mathsf{G}\ \underline{n+1}\ \underline{-k} \Downarrow^e \underline{1}}$$

   where T is

   $$\dfrac{\dfrac{}{\mathsf{G} \Downarrow^e \mathsf{G}} \quad \dfrac{\dfrac{}{\underline{n+1} \Downarrow^e \underline{n+1}} \quad \dfrac{}{\underline{1} \Downarrow^e \underline{1}}}{\underline{n+1}-\underline{1} \Downarrow^e \underline{n}} \quad \dfrac{}{\mathsf{G}\underline{n} \Downarrow^e \mathsf{G}\underline{n}}}{\mathsf{G}(\underline{n+1}-\underline{1}) \Downarrow^e \mathsf{G}\underline{n}} \quad \dfrac{\dfrac{}{\underline{-k} \Downarrow^e \underline{-k}} \quad \dfrac{}{\underline{1} \Downarrow^e \underline{1}}}{\underline{-k}-\underline{1} \Downarrow^e \underline{-k}-\underline{1}} \quad \dfrac{T'}{\mathsf{G}\ \underline{n}\ \underline{-k}-\underline{1} \Downarrow^e \underline{1}}$$

   $$\dfrac{}{\mathsf{G}\ \ (\underline{n+1}-\underline{1})\ \ (\underline{-k}-\underline{1}) \Downarrow^e \underline{1}}$$

and the deduction tree T' exists by appeal to the inductive hypothesis $\phi(n)$, taking r to be $k+1$. Clearly $k+1 \geq 1$ and $-(k+1) = -k-1$. As $k \geq 1$ was arbitrary, the above trees show that $\phi(n+1)$ holds. As n was itself arbitrary, we have verified

$$\forall n \geq 1.\phi(n) \ \textit{implies} \ \phi(n+1)$$

So by induction we have $\forall n \geq 1.\forall r \geq 1.\ \mathsf{G}\ \underline{n}\ \underline{-r} \Downarrow^e \underline{1}$ and the desired result follows from this.

2. If $z \geq 1$ then $\mathsf{F}\,\underline{z} \Downarrow^e \underline{1}$ and $\mathsf{F}\,\underline{-z} \Downarrow^e \underline{-1}$. Also $\mathsf{F}\,\underline{0} \Downarrow^e \underline{0}$.

3. Define E_{lt} to be

$$\mathsf{F}\,(x-y) == \underline{-1}$$

7.3 Function Abstraction and Locality

Motivation 7.3.1 So far, each function we have defined has had a name, typically F. We can define programs which act as functions, without being explicitly named by an identifier. The expression $\mathsf{fn}\,x.x+\underline{2}$ is a program whose intended meaning is the function which "adds 2". We can apply the function to an input, writing $(\mathsf{fn}\,x.x+\underline{2})\,\underline{4}$. This expression will evaluate to $\underline{4}+\underline{2}$ (and thus to $\underline{6}$). If we wrote $\mathsf{F}\,x = x+\underline{2}$ in an identifier declaration, then F and $\mathsf{fn}\,x.x+\underline{2}$ would be interchangeable. In practical programming, it is often convenient to code a function directly, without bothering to give it a name via an identifier.

It is also useful to be able to make *local declarations*. Suppose that E_2 is some code in which the expression E_1 appears many times. We could cut the code down by declaring $\mathsf{K} = E_1$ and using K for E_1 in E_2. However, is is also useful to be able to do this without *naming* E_1 *globally*, which is what the would happen if we declared K. The syntax $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$ achieves this.

We shall now consider the language $\mathbb{FUN}^e$ extended with these two new kinds of expressions.

Definitions 7.3.2 We extend the expression grammar for $\mathbb{FUN}^e$ with the following two clauses:

$$E ::= \ldots \mid \mathsf{fn}\,x.E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E$$

We call $\mathsf{fn}\,x.E$ a **function abstraction** and $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$ a **local declaration**. The type assignment rules are given in Table 7.3.

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

Table 7.3: Extending the Type Assignment Relation $\Gamma \vdash E :: \sigma$ in $\mathbb{FUN}^e$

Remark 7.3.3 We shall adopt a few conventions to make terms more readable: We take function abstraction $\mathsf{fn} x.E$ to mean $\mathsf{fn} x.(E)$. Thus we can write $\mathsf{fn} x.\mathsf{fn} y.y + \underline{2}$ instead of the more clumsy $\mathsf{fn} x.(\mathsf{fn} y.(y + \underline{2}))$. We call E the **body** of $\mathsf{fn} x.E$.

Examples 7.3.4 Examples of new expressions are

(1) $\mathsf{fn} x.x$ (the identity function—why?);

(2) $f\,(g\,y)$;

(3) $\mathsf{fn}\, f.\mathsf{fn} g.\, f\, g\, \underline{4}$; and

(4) $\mathsf{let}\, x = \underline{4}\, \mathsf{in}\, \mathsf{let}\, y = \underline{T}\, \mathsf{in}\, (x,y)$.

7.3.1 Free and Bound Variables

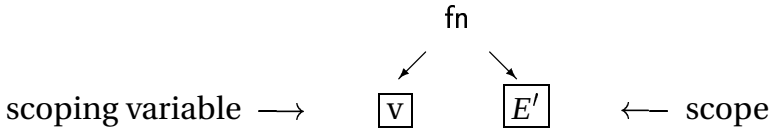
Motivation 7.3.5 The intended meaning of $\mathsf{fn} x.x + \underline{2}$ is the function which adds 2 to its argument. What about $\mathsf{fn} y.y + \underline{2}$? Well, it too should be a function which adds 2. The name of the variable used to form such a expression is not relevant to the intended meaning of the expression—the variables x and y are said to be *bound*. However, the expressions $x + \underline{2}$ and $y + \underline{2}$ are certainly different—the value of each expression is respectively 2 added to x and 2 added to y, so the values will only be the same if $x = y$. Here, the variables x and y are said to be *free*. Why do we need to study free and bound variables? Let us look at an example. Suppose that E_1 and E_2 are expressions. If one thinks of E_1 as a functional program, and the free occurrences of a variable x in E_1 as places at which new code could be executed, we might consider replacing the variable x by E_2. Such a replacement is called a *substitution*. We can try to substitute an expression E_2 for free occurrences of x in E_1 simply by replacing each free x with E_2; this will produce a new expression which will be denoted $E_1[E_2/x]$. For example, $(\mathsf{if}\, x\, \mathsf{then}\, \underline{4}\, \mathsf{else}\, \underline{5})[\,\underline{1} == \underline{2}\,/x]$ denotes the expression $\mathsf{if}\, \underline{1} == \underline{2}\, \mathsf{then}\, \underline{4}\, \mathsf{else}\, \underline{5}$.

But there is a problem lurking! Let us write $f \stackrel{\text{def}}{=} \mathsf{fn} x.E_1$. Given any expression E_2, the intended meaning of $f\,E_2$ is $E_1[E_2/x]$. Thus if E_1 is $x + y$, then $f\,E_2 = (x+y)[E_2/x] = E_2 + y$. So if $E \stackrel{\text{def}}{=} \mathsf{fn} x.x + y$, the intended meaning of E is "the function with adds y". We can try substituting the expression x for the occurrence of y in E. Now, $E[x/y]$ ought to be "the function which adds x". But in fact $E[x/y]$ is clearly the expression $\mathsf{fn} x.x + x$,

which is the function which doubles an integer input! The problem arises because when the variable x is substituted for the *free* variable y in $\mathsf{fn}\,x.x+y$, x becomes a *bound* variable.

Note that the expressions $\mathsf{fn}\,x.x+y$ and $\mathsf{fn}\,z.z+y$ can be regarded as "the same" in the sense that the intended meaning of *each* expression is the "function which adds" y. When we attempted to substitute x for the free y in $\mathsf{fn}\,x.x+y$, we noted that x would become bound. But if the intended *meaning* of $\mathsf{fn}\,z.z+y$ is the same as $\mathsf{fn}\,x.x+y$, what about substituting x for y in $\mathsf{fn}\,z.z+y$ to get $\mathsf{fn}\,z.z+x$? The latter expression is indeed what we were after—"the function which adds x". Informally we say that we *re-name* the bound variable x in $\mathsf{fn}\,x.x+y$ as a new variable z so that when x is substituted for y it does not become bound.

Definitions 7.3.6 We say that an expression E' is the **scope** of v in an expression of the form $\mathsf{fn}\,v.E'$. We call such an occurrence of v a **scoping** variable. The syntax tree for $\mathsf{fn}\,v.E'$ looks like this



In $\mathsf{let}\ v = E_1\ \mathsf{in}\ E_2$, the **scope** of v is E_2. Exercise: draw the tree. We also call such a v a **scoping** variable.

Now suppose that x is a variable which does occur in an expression E—of course x may occur more than once, possibly many times. Each *occurrence* of x (in E) is either free or bound. We say that an occurrence of x is **bound** in E if and only if the occurrence of x in E is in a *subexpression* of the form $\mathsf{fn}\,x.E'$ or $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$. A consequence of this definition is that an occurrence of x in E is bound just in case

(i) the occurrence is a scoping variable;

(ii) the occurrence occurs within the scope of a scoping occurrence of x.

If there is an occurrence of x in such E' or E_2 then we sometimes say that this bound occurrence of x has been **captured** by the scoping x. An occurrence of x in E is **free** iff the occurrence of x is not bound. Before reading on, take a look at Examples 7.3.7.

We shall write $var(E)$ for the set of all variables which occur in E. We can give a recursive definition of the set $var(E)$ which is obvious and omitted (cf the definition of $fvar(E)$ which follows). We write $fvar(E)$ for the set of variables which have free occurrences in E. We can define this recursively by the following (obvious!) clauses:

- $fvar(x) \stackrel{\text{def}}{=} \{x\}$;
- $fvar(\underline{c}) \stackrel{\text{def}}{=} \varnothing$;

- $fvar(E_1\ op\ E_2) \stackrel{\text{def}}{=} fvar(E_1) \cup fvar(E_2)$;
- $fvar(\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3) \stackrel{\text{def}}{=} fvar(E_1) \cup fvar(E_2) \cup fvar(E_3)$;
- $fvar(\mathsf{fn}x.E) \stackrel{\text{def}}{=} fvar(E) \setminus \{x\}$; occurrences of x in E are captured by the scope of $\mathsf{fn}x$, and hence are not free;
- $fvar(E_1 E_2) \stackrel{\text{def}}{=} fvar(E_1) \cup fvar(E_2)$;
- $fvar(\mathsf{nil}) \stackrel{\text{def}}{=} \varnothing$;
- $fvar(\mathsf{hd}(E)) \stackrel{\text{def}}{=} fvar(E)$;
- $fvar(\mathsf{tl}(E)) \stackrel{\text{def}}{=} fvar(E)$;
- $fvar(E_1 : E_2) \stackrel{\text{def}}{=} fvar(E_1) \cup fvar(E_2)$;
- $fvar(\mathsf{elist}(E)) \stackrel{\text{def}}{=} fvar(E)$; and
- $fvar(\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2) \stackrel{\text{def}}{=} fvar(E_1) \cup (fvar(E_2) \setminus \{x\})$.

We leave the (easy) recursive definition of the set $bvar(E)$ of the set of variables with bound occurrences in E to the reader.

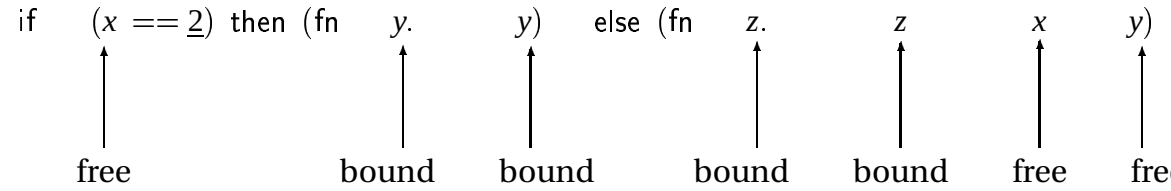
Examples 7.3.7 $u+v$ is the scope of u in $\mathsf{fn}\underline{x}.(\mathsf{fn}\underline{u}.u+v)\,z$. Example subexpressions are

$$z \qquad \text{and} \qquad \mathsf{fn}\underline{u}.u+v.$$

The underlined occurrences are scoping variables. If $N \stackrel{\text{def}}{=} \mathsf{fn}x.xxy\underline{x}zx$ then the underlined x is one of five occurrences of x in N.

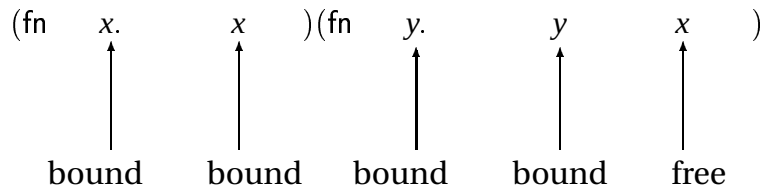
Warning: Note that a variable may occur both free and bound in a expression. Here are some examples:

(1)



Here, the set of free variables is $\{x,y\}$ and the set of bound variables is $\{y,z\}$. We could say that the second occurrence of z in the conditional has been captured by the scoping occurrence of z.

(2)



Here, the set of free variables is $\{x\}$ and the set of bound variables is $\{x,y\}$.

(3) In $\mathsf{fn}\, v.(\mathsf{let}\ z = z + x + y\ \mathsf{in}\ z + \underline{4} - v)$ the first and last v are bound, the first and third z are bound but the second is free, and the x and y are free. What are the (occurrences of) scoping variables?

7.3.2 Substitution of Terms

Definitions 7.3.8 Given E_1 and E_2 and v, we define the expression $E_1[E_2/v]$ to be "the" expression resulting from replacing all free occurrences of v in E_1 by the expression E_2, *renaming bound variables as necessary to avoid capture*. Note that we have written "the" above, as different expressions will result if we rename variables in different ways.

Examples 7.3.9 Informal examples are

$$\begin{array}{rcl}
(\mathsf{fn}\, x.x + y)[\underline{2}/y] & = & \mathsf{fn}\, x.x + \underline{2} \\
(\mathsf{fn}\, x.x + y)[x/y] & = & \mathsf{fn}\, x'.x' + x \\
(\mathsf{let}\ x = y + \underline{4}\ \mathsf{in}\ x + z + \underline{7})[u + v/z] & = & \mathsf{let}\ x = y + \underline{4}\ \mathsf{in}\ x + (u + v) + \underline{7} \\
(\mathsf{let}\ x = y + \underline{4}\ \mathsf{in}\ x + z + \underline{7})[u + y/z] & = & \mathsf{let}\ x = y + \underline{4}\ \mathsf{in}\ x + (u + y) + \underline{7} \\
(\mathsf{let}\ x = z + \underline{4} - x\ \mathsf{in}\ x + z + \underline{7})[x + y/z] & = & \mathsf{let}\ x' = (x + y) + \underline{4} - x\ \mathsf{in}\ x' + (x + y) + \underline{7} \\
((v_1 + \underline{2}) : (v_3 v_2))[\underline{10}/v_2] & = & (v_1 + \underline{2}) : (v_3\ \underline{10}) \\
(\mathsf{let}\ u = u\ \mathsf{in}\ u + \underline{7})[\underline{7}/u] & = & \mathsf{let}\ u = \underline{7}\ \mathsf{in}\ u + \underline{7}
\end{array}$$

In the second example, the substituted x will appear in the scope of the scoping x, so we rename (to x') any bound x to avoid capture. In the fifth example, the substituted x will appear in the scope of the scoping x, so we rename (to x') any bound x to avoid capture. In the final example, just the second occurrence of u in $\mathsf{let}\ u = u\ \mathsf{in}\ u + \underline{7}$ is free.

Remark 7.3.10 Let us return to the renaming problem. In the previous example, should $(\mathsf{fn}\, x.x + y)[x/y]$ be $\mathsf{fn}\, x'.x' + x$ or $\mathsf{fn}\, z.z + x$ or ...? We ought really to make a unique choice for the renamed variable. However, we have seen that as far as the semantics of expressions is concerned, the choice is not too critical: we saw that the two expressions $\mathsf{fn}\, x.x + y$ and $\mathsf{fn}\, z.z + y$ have the same intended meaning, namely that they both represent the function which adds y.

For these reasons, we shall *regard expressions which differ only in the names of their bound variables as equivalent*. In fact, to be precise, the idea that two expressions differ only in their bound variables is an *equivalence relation* on *Exp*. We should then work the rest of this course using α-equivalence classes of expressions instead of simple expressions. However, we will gloss over these technical complications in the rest of MC 308.

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

Table 7.4: Extending the Evaluation Relation

7.3.3 Extending the Operational Semantics

Definitions 7.3.11 An operational semantics for function abstractions and local declarations is given in Table 7.4. We only consider an eager semantics.

Examples 7.3.12

(1) Give the *set* of free variables, and *set* of bound variables, for each of the following expressions:

1. $\mathsf{fn}\, f.\mathsf{fn}\, x.\, f\,(f\,(x * \underline{7}))$
2. $\mathsf{let}\, l = u : v : \mathsf{nil}\, \mathsf{in}\, f\,(f\, l)$
3. $\mathsf{let}\ f = \mathsf{fn}\, x.x + y\, \mathsf{in}\, \mathsf{let}\, u = x : \mathsf{nil}\, \mathsf{in}\, (f(\mathsf{hd}(u)) - z)$

Answer

1. $\varnothing, \{x, f\}$.
2. $\{f, u, v\}, \{l\}$.
3. $\{x, y, z\}, \{x, u, f\}$.

(2) Prove that $(\mathsf{fn}\, z.z * \underline{2})\,\underline{3} \Downarrow^e \underline{6}$. To do this, we produce a deduction tree. First note that the program being evaluated is an application. So it *must* arise by the rule AA, hence we need to show that $\mathsf{fn}\, z.z * \underline{2} \Downarrow^e \mathsf{fn}\, x.E$ for some x and E, that $\underline{3} \Downarrow^e V'$ for some V', and that $E[V'/x] \Downarrow^e \underline{6}$. The first of these is easy, being an instance of VAL with $x = z$ and $E = z * \underline{2}$. The second must be also an instance of VAL with $V' = \underline{3}$, and the third, namely $\underline{3} * \underline{2} \Downarrow^e \underline{6}$, is also easy following from OP. Putting this altogether we get

$$\dfrac{\dfrac{}{\mathsf{fn}\, z.z * \underline{2} \Downarrow^e \mathsf{fn}\, z.z * \underline{2}}\,\text{VAL} \qquad \dfrac{}{\underline{3} \Downarrow^e \underline{3}}\,\text{VAL} \qquad \dfrac{\dfrac{}{\underline{3} \Downarrow^e \underline{3}}\,\text{VAL} \qquad \dfrac{}{\underline{2} \Downarrow^e \underline{2}}\,\text{VAL}}{(z * \underline{2})[\underline{3}/z] = \underline{3} * \underline{2} \Downarrow^e \underline{6}}\,\text{OP}}{(\mathsf{fn}\, z.z * \underline{2})\,\underline{3} \Downarrow^e \underline{6}}\,\text{AA}$$

(3) Prove that $\mathsf{hd}((\mathsf{fn}\, x.x + \underline{2})\,\underline{3} : \mathsf{nil}) \Downarrow^e \underline{5}$. To do this, we derive a deduction tree:

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

(4) Let $\Gamma \stackrel{\text{def}}{=} h :: ((X \to Y) \to Y) \to X \to \mathsf{int}, x :: X$. We show that

$$\Gamma \vdash \mathsf{let}\ f = \mathsf{fn}\,g.gx\ \mathsf{in}\ (h\,f)\,x :: \mathsf{int}$$

with the following deduction

$$\dfrac{\mathcal{D} \qquad \mathcal{D}_1}{\Gamma \vdash \mathsf{let}\ f = \mathsf{fn}\,g.gx\ \mathsf{in}\ (h\,f)\,x :: \mathsf{int}}$$

where $\mathcal{D}$ is

$$\dfrac{\dfrac{\dfrac{}{\Gamma, g :: X \to Y \vdash g :: X \to Y} \qquad \dfrac{}{\Gamma, g :: X \to Y \vdash x :: X}}{\Gamma, g :: X \to Y \vdash gx :: Y}}{\Gamma \vdash \mathsf{fn}\,g.gx :: (X \to Y) \to Y}$$

and where $\mathcal{D}_1$ is

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash h :: ((X \to Y) \to Y) \to X \to \mathsf{int}} \qquad \mathcal{D}}{\Gamma \vdash h(\mathsf{fn}\,g.gx)\ :: X \to \mathsf{int}} \qquad \dfrac{}{\Gamma \vdash x :: X}}{\Gamma \vdash (h\,(\mathsf{fn}\,g.gx))\,x :: \mathsf{int}}$$

8

Type Checking and Inference

8.1 Introduction

Motivation 8.1.1 A language is **strongly typed** if every legal expression has at least one type. A strongly typed language is **monomorphic** if every legal expression has a unique type (for example Pascal). A strongly typed language is **polymorphic** if a legal expression can have several types (for example Standard ML and Haskell). As an example, the list reversing function `rev` might have type

```
rev ::  [X] -> [X]
```

The idea is that `X` is a *type variable* which denotes *any type*; and `[X]` denotes the type of lists whose data elements have type `X`. In this chapter we shall study type inference in a language called $\mathbb{PFUN}$. We shall be able to study some of the formal properties of types and type inference. Common forms of polymorphism are

• **Overloading**: The same symbol is used to denote (finitely many) functions, implemented by *different* algorithms—context determines which is meant.

• **Parametric**: One expression belongs to a (usually infinite) family of *structurally related* types. A parametrically polymorphic procedure is given by a single algorithm which may be applied to arguments which possess different, but structurally similar, types. This minimizes duplication of code. Type expressions involve *parameters*, that is, type variables. Haskell enjoys parametric polymorphism. The function `rev :: [X] -> [X]` is an example of parametric polymorphism; all lists have the same structure, being either empty, or having a head and tail.

• **Implicit**: This is a particular form of parametric polymorphism, and we meet it later on.

8.2 The Types and Expressions of $\mathbb{PFUN}$

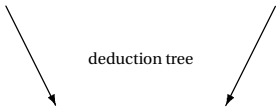
Definitions 8.2.1 Let us write *TyVar* for a countably infinite set of **type variables** $\{V_1, V_2, V_3, \ldots\}$. We shall often write X, Y, Z, U, W etc for type variables. The set *Type* of **types** of $\mathbb{PFUN}$ is inductively specified by the grammar

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

We shall write $TV(\sigma)$ for the set of type variables appearing in σ. The rules for deriving type assignments are exactly those in Figure 6.1. The expressions of $\mathbb{PFUN}$ are those of the extended language in the last chapter.

8.3 Type Assignment Examples

Motivation 8.3.1 Suppose that you are asked, given some Γ, E and σ, to prove that $\Gamma \vdash E :: \sigma$. This is another instance of *type checking*. So, you have to give a deduction of $\Gamma \vdash E :: \sigma$ using the rules in Figure 6.1. To figure out the deduction tree, suppose that the tree has the general form



$$\frac{}{\Gamma \vdash E :: \sigma} :: R$$

where $::R$ was the final rule used in the deduction. Which rule is $::R$? To see this, look at the expression E. You will then see that *only one of the rules in Figure 6.1 applies*, and this will allow you to work out what the hypotheses of the rule $::R$ must be. You can then continue to work backwards until the deduction tree is complete. In the examples, any typing rule $::?$ is written as just $?$ to save space.

Examples 8.3.2

(1) Prove that $\vdash \underline{T} : \mathsf{nil} :: [\mathsf{bool}]$.

We produce a deduction tree for this type assignment. The expression is a list, so this typing assertion must have been deduced using the rule CONS. It is clear what the (two) hypotheses of CONS must be; and it is also clear that the two hypotheses are deduced using instances of base rules (such as $\vdash \underline{T} :: \mathsf{bool}$, an instance of TRUE where $\Gamma = \varnothing$):

$$\dfrac{\dfrac{}{\vdash \underline{T} :: \mathsf{bool}}\,\text{TRUE} \qquad \dfrac{}{\vdash \mathsf{nil} :: [\mathsf{bool}]}\,\text{NIL}}{\vdash \underline{T} : \mathsf{nil} :: [\mathsf{bool}]}\,\text{CONS}$$

(2) Show that $\Gamma \vdash \mathsf{fn}\, x.(\underline{0} : x) :: [\mathsf{int}] \to [\mathsf{int}]$ for any context Γ.

We produce a deduction tree: note that the expression is a function, so the final rule used in the deduction must be ABS, where $E = \underline{0} : x$, and $\sigma = \tau = [\mathsf{int}]$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x :: [\mathsf{int}] \vdash \underline{0} :: \mathsf{int}}\,\text{INT} \qquad \dfrac{}{\Gamma, x :: [\mathsf{int}] \vdash x :: [\mathsf{int}]}\,\text{VAR}}{\Gamma, x :: [\mathsf{int}] \vdash \underline{0} : x :: [\mathsf{int}]}\,\text{CONS}}{\Gamma \vdash \mathsf{fn}\, x.(\underline{0} : x) :: [\mathsf{int}] \to [\mathsf{int}]}\,\text{ABS}$$

(3) Prove that $\underline{T} : \underline{2} : \mathsf{nil}$ is not typable in $\mathbb{PFUN}$ in any context Γ.

To do this, we try to derive a deduction tree, starting out with an arbitrary context Γ. The last rule applied must have been CONS, with $E_1 = \underline{T}$ and $E_2 = \underline{2} : \mathsf{nil}$ and so there has to be a type, say σ, for which the expression has type $[\sigma]$. We produce the deduction tree:

$$\dfrac{\dfrac{}{\Gamma \vdash \underline{T} :: \sigma}\,\text{TRUE} \qquad \dfrac{\dfrac{}{\Gamma \vdash \underline{2} :: \sigma}\,\text{INT} \qquad \dfrac{}{\Gamma \vdash \mathsf{nil} :: [\sigma]}\,\text{NIL}}{\Gamma \vdash \underline{2} : \mathsf{nil} :: [\sigma]}\,\text{CONS}}{\Gamma \vdash \underline{T} : \underline{2} : \mathsf{nil} :: [\sigma]}\,\text{CONS}$$

It follows that $\sigma = \mathsf{int}$ and $\sigma = \mathsf{bool}$ and this cannot be. So no typing for $\underline{T} : \underline{2} : \mathsf{nil}$ exists.

(4) Show that $\mathsf{hd}(y : \underline{3})$ is not typable in $\mathbb{PFUN}$ in any context Γ.

Working backwards we have:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash y :: \sigma}\,\text{VAR} \qquad \dfrac{}{\Gamma \vdash \underline{3} :: [\sigma]}\,\text{INT}}{\Gamma \vdash y : \underline{3} :: [\sigma]}\,\text{CONS}}{\Gamma \vdash \mathsf{hd}(y : \underline{3}) :: \sigma}\,\text{HD}$$

Looking at the rule INT (which must be used to type $\underline{3}$) we must have $\mathsf{int} = [\sigma]$, a contradiction. So the expression cannot be typable.

(5) Show that in $\mathbb{PFUN}$ we have $\vdash \mathsf{fn}\, x.(x : \mathsf{nil}) :: X \to [X]$ where X is a type variable.

To do this, we note that a unique rule from Figure 6.1 must be used to derive the typing assertion, and it has to be ABS. A careful inspection shows us that we have $\Gamma = \varnothing$, $E = x : \mathsf{nil}$, $\sigma = X$ and $\tau = [X]$. From this we can see that the hypothesis of ABS must be $x :: X \vdash x : \mathsf{nil} :: [X]$. The rest of the backward steps are equally easy, and we simply give the final tree:

$$\dfrac{\dfrac{\dfrac{}{x :: X \vdash x :: X}\,\text{VAR} \qquad \dfrac{}{x :: X \vdash \mathsf{nil} :: [X]}\,\text{NIL}}{x :: X \vdash x : \mathsf{nil} :: [X]}\,\text{CONS}}{\vdash \mathsf{fn}\, x.(x : \mathsf{nil}) :: X \to [X]}\,\text{ABS}$$

(6) Show that $\vdash \mathsf{fn}\, f.(f\,\mathsf{nil}, \underline{T}) :: ([X] \to Y) \to (Y, \mathsf{bool})$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f :: [X] \to Y \vdash f :: [X] \to Y}\,\text{VAR}}{\vdots}}{f :: [X] \to Y \vdash f\,\mathsf{nil} :: Y}\,\text{AP} \qquad \dfrac{\dfrac{}{f :: [X] \to Y \vdash \mathsf{nil} :: [X]}\,\text{NIL}}{f :: [X] \to Y \vdash \underline{T} :: \mathsf{bool}}\,\text{TRUE}}{f :: [X] \to Y \vdash (f\,\mathsf{nil}, \underline{T}) :: (Y, \mathsf{bool})}\,\text{PAIR}}{\vdash \mathsf{fn}\, f.(f\,\mathsf{nil}, \underline{T}) :: ([X] \to Y) \to (Y, \mathsf{bool})}\,\text{ABS}$$

(7) Show that $\vdash \mathsf{fn}\, f.\mathsf{fn}\, x.\, f\,(f\,x) :: (X \to X) \to X \to X$. Writing Γ for $f :: X \to X, x :: X$ we have

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f :: X \to X}\,\text{VAR} \quad \dfrac{}{\Gamma \vdash x :: X}\,\text{VAR}}{\Gamma \vdash f\,x :: X}\,\text{AP} \quad \dfrac{}{\Gamma \vdash f :: X \to X}\,\text{VAR}}{\Gamma \vdash f\,(f\,x) :: X}\,\text{AP}$$
$$\dfrac{\dfrac{}{f :: X \to X \vdash \mathsf{fn}\, x.\, f\,(f\,x) :: X \to X}\,\text{ABS}}{\vdash \mathsf{fn}\, f.\mathsf{fn}\, x.\, f\,(f\,x) :: (X \to X) \to X \to X}\,\text{ABS}$$

(8) Show that $(\mathsf{fn}\, f.\, fy)\, y$ is not typable for *any* context of the form $y :: \tau$. (Note that y is the only free variable).

We suppose, for a contradiction, that the expression is typeable. Let us call this type σ_1, say. We have:

$$\dfrac{\dfrac{\dfrac{}{y :: \tau, f :: \sigma_2 \vdash f :: \sigma_3 \to \sigma_1}\,\text{VAR} \quad \dfrac{}{y :: \tau, f :: \sigma_2 \vdash y :: \sigma_3}\,\text{VAR}}{y :: \tau, f :: \sigma_2 \vdash fy :: \sigma_1}\,\text{AP}}{y :: \tau \vdash \mathsf{fn}\, f.\, fy :: \sigma_2 \to \sigma_1}\,\text{ABS} \qquad \mathcal{D}}{y :: \tau \vdash (\mathsf{fn}\, f.\, fy)\, y :: \sigma_1}\,\text{AP}$$

where $\mathcal{D}$ is

$$\dfrac{}{y :: \tau \vdash y :: \sigma_2}\,\text{VAR}$$

You should be familiar with producing deduction trees by now! For example, suppose we had worked back to the hypothesis $y :: \tau, f :: \sigma_2 \vdash fy :: \sigma_1$ of ABS. fy is an application, so must have been deduced from rule AP. fy has type σ_1, hence rule AP tells us that f must have type $\sigma_3 \to \sigma_1$ and y has type σ_3 for some type σ_3. But the final rules at the top must be instances of VAR. So we must have $\sigma_2 = \sigma_3 \to \sigma_1$ and $\tau = \sigma_3$ and $\tau = \sigma_2$. Thus we deduce $\sigma_3 \to \sigma_1 = \sigma_3$ and this cannot be. So the expression is not typable.

(9) Show that given $I \stackrel{\text{def}}{=} \mathsf{len} :: [X] \to \mathsf{int}$ then $\mathsf{len}\, l = \mathsf{if}\ \mathsf{elist}(l)\ \mathsf{then}\ \underline{0}\ \mathsf{else}\ (\underline{1} + \mathsf{len}\,(\mathsf{tl}(l)))$ is a valid declaration. Thus we need to prove that

$$l :: [X] \vdash \mathsf{if}\ \mathsf{elist}(l)\ \mathsf{then}\ \underline{0}\ \mathsf{else}\ (\underline{1} + \mathsf{len}\,\mathsf{tl}(l)) :: \mathsf{int}$$

Exercise!

8.4 Type Substitutions

Motivation 8.4.1 It is easy to see that there are expressions of $\mathbb{PFUN}$ which are not typable. It is also possible for an expression to have exactly one type, for example $\vdash \underline{1} :: \sigma$ holds only for $\sigma = \mathsf{int}$. However, if an expression in which there are no free

variables is typable, it usually has many types. For example, $\vdash \mathsf{fn}\, x.x :: \sigma \to \sigma$ holds for any type σ. However, in $\mathbb{PFUN}$, of all the types that can be assigned to an expression, there is a most general one, in the sense that all other assigned types are instances of the most general type. We call this the *principal* type. The principal type of $\mathsf{fn}\, x.x$ is $X \to X$; any type $\sigma \to \sigma$ is obtained by taking X to be σ. We now make all this precise, and give a number of results about $\mathbb{PFUN}$ type inference:

Definitions 8.4.2 We call S a **type substitution** if it is a (possibly empty) finite set of (type-variable,type) pairs in which all the variables are distinct. We will write a typical S in the form $\langle X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n \rangle$. We write the empty type substitution as $\langle\rangle$. If τ is any type, we shall write $S\{\tau\}$ to denote the type τ in which any occurrence of X_i is changed to σ_i. Thus

$$\langle X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n \rangle \{\tau\} \stackrel{\text{def}}{=} \tau[\sigma_1, \ldots, \sigma_n / X_1, \ldots, X_n] \qquad \text{and} \qquad \langle\rangle\{\tau\} \stackrel{\text{def}}{=} \tau$$

We will define equality of type substitutions in a similar way to function equality, namely

$$S = S' \textit{ iff } \forall \tau. \quad S\{\tau\} = S'\{\tau\} \qquad (\dagger)$$

Note that there are no variable binding operations on type variables. We say that σ **generalises** σ' if there exists a type substitution S for which $\sigma' = S\{\sigma\}$, and say that σ' is an **instance** of σ. Given substitutions S_1 and S_2 there is a substitution $S_1 \cdot S_2$ whose action on any type τ is given by $(S_1 \cdot S_2)\{\tau\} = S_1\{S_2\{\tau\}\}$. We call $S_1 \cdot S_2$ a **composition** of S_1 and S_2. Note that this just describes how $S_1 \cdot S_2$ acts on a type, and does not give $S_1 \cdot S_2$ as an explicit set of pairs. To show that $S_1 \cdot S_2$ actually exists, we should describe it as such a set—see the examples below. If $S \stackrel{\text{def}}{=} \langle V \mapsto \sigma, X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n \rangle$ then we define S^V to be $\langle X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n \rangle$ and also $\langle\rangle^V$ to be $\langle\rangle$.

In $\mathbb{PFUN}$, if $\varnothing \vdash P :: \sigma$, the type σ assigned to the expression P is **principal** if σ generalises any other type which can be assigned to P. The principal type of $\mathsf{fn}\, x.x$ is $X \to X$. Note that the principal type is unique up to a consistent renaming of variables. Another principal type for $\mathsf{fn}\, x.x$ is $V \to V$.

Examples 8.4.3

(1) Define $S \stackrel{\text{def}}{=} \langle X \mapsto U, Y \mapsto \mathsf{bool} \rangle$. Let $\sigma \stackrel{\text{def}}{=} (X, Y \to Z)$ and $\Gamma \stackrel{\text{def}}{=} x :: X, y :: Y \to Z$. Then

$$S\{\sigma\} = (U, \mathsf{bool} \to Z) \quad \text{and} \quad S\{\Gamma\} = x :: S\{X\}, y :: S\{Y \to Z\} = x :: U, y :: \mathsf{bool} \to Z$$

(2) Note that $(X,Y) \to Z$ generalises $([\mathsf{bool}], Y) \to \mathsf{int}$ for

$$([\mathsf{bool}], Y) \to \mathsf{int} = S\{((X,Y) \to Z)\}$$

where $S \stackrel{\text{def}}{=} \langle X \mapsto [\mathsf{bool}], Z \mapsto \mathsf{int} \rangle$

(3) It follows from the definitions that (for example) $\langle X \mapsto X \rangle = \langle\rangle$. Think of some other examples of type substitutions which are equal, but not specified by the same sets.

(4) As mentioned above, the definition of composition of type substitutions does not describe $S_1 \cdot S_2$ as an explicit set of pairs. It is possible to calculate compositions explicitly, with a little thought. Consider $\langle X \mapsto \mathsf{int}, Y \mapsto X \rangle \cdot \langle Z \mapsto \mathsf{int} \rangle$. This substitution changes any occurrence of Z to int, then *at the same time* any X to int and any Y to X. Thus the composition is

$$\langle X \mapsto \mathsf{int}, Y \mapsto X, Z \mapsto \mathsf{int} \rangle$$

Now consider $\langle X \mapsto \mathsf{int}, Y \mapsto X \rangle \cdot \langle Y \mapsto \mathsf{int} \rangle$. Any Y will change to int. Then any X changes to int, but *there are no Ys* left. Thus the composition is

$$\langle X \mapsto \mathsf{int}, Y \mapsto \mathsf{int} \rangle$$

(5) Check that

$$\langle X \mapsto \mathsf{bool}, Y \mapsto X \rangle \cdot \langle Z \mapsto Y \rangle = \langle X \mapsto \mathsf{bool}, Y \mapsto X, Z \mapsto X \rangle$$

(6) Check that

$$\langle X \mapsto \mathsf{bool}, Y \mapsto U \rangle \cdot \langle Y \mapsto X, Z \mapsto Y \rangle = \langle Y \mapsto \mathsf{bool}, X \mapsto \mathsf{bool}, Z \mapsto U \rangle$$

(7) Check that

$$\langle X \mapsto \mathsf{int}, Y \mapsto X \rangle \cdot \langle X \mapsto (X, Y), U \mapsto (U \to Y) \rangle = \langle X \mapsto (\mathsf{int}, X), Y \mapsto X, U \mapsto (U \to X) \rangle$$

(8) As an exercise, try to write down a formula which gives

$$\langle Y_1 \mapsto \tau_1, \ldots, Y_m \mapsto \tau_m \rangle \cdot \langle X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n \rangle$$

as a set of pairs. Note that there might be several answers but each answer will be equal according to †. Note that this requires care, because some of the Y_js may well be X_is.

8.5 Local Polymorphism in $\mathbb{PFUN}$

Motivation 8.5.1 The LET rule permits different occurrences of x in E_2 to have different **implicit** types in a local declaration $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$. Thus, E_1 can be used polymorphically in the body E_2. This idea is best explained by example.

Example 8.5.2 We first note that $\vdash \mathsf{fn}\, x.x :: X \to X$:

$$\mathcal{D}(X) \left\{ \dfrac{\dfrac{}{x :: X \vdash x :: X}\ \text{VAR}}{\vdash \mathsf{fn}\, x.x :: X \to X}\ \text{ABS} \right.$$

Note that replacing X by any type σ in $\mathcal{D}(X)$ yields a deduction $\mathcal{D}(\sigma)$ for $\vdash \mathsf{fn}x.x :: \sigma \to \sigma$. Hence

$$\mathcal{D}_1\left\{\dfrac{\mathcal{D}(\mathsf{bool})^{(2)} \qquad \dfrac{}{\vdash \underline{T} :: \mathsf{bool}}\,\text{TRUE}}{\vdash (\mathsf{fn}x.x)\,\underline{T} :: \mathsf{bool}}\,\text{AP}\right.$$

and

$$\mathcal{D}_2\left\{\dfrac{\mathcal{D}([X])^{(3)} \qquad \dfrac{}{\vdash \mathsf{nil} :: [X]}\,\text{NIL}}{\vdash (\mathsf{fn}x.x)\,\mathsf{nil} :: [X]}\,\text{AP}\right.$$

Putting things together we get

$$\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\vdash ((\mathsf{fn}x.x)\,\underline{T}, (\mathsf{fn}x.x)\,\mathsf{nil}) :: (\mathsf{bool}, [X])}\,\text{PAIR}$$

Note that $(f\,\underline{T}, f\,\mathsf{nil})[(\mathsf{fn}x.x)/f] \quad = \quad ((\mathsf{fn}x.x)\,\underline{T}, (\mathsf{fn}x.x)\,\mathsf{nil})$. So we have

$$\dfrac{\mathcal{D}(Y) \qquad \dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\vdash (f\,\underline{T}, f\,\mathsf{nil})[(\mathsf{fn}x.x)/f] :: (\mathsf{bool}, [X])}}{\vdash \mathsf{let}\ f = (\mathsf{fn}x.x)\ \mathsf{in}\ (f\,\underline{T}, f\,\mathsf{nil}) :: (\mathsf{bool}, [X])}\,\text{LET}$$

If we look at the above deduction of

$$\vdash \mathsf{let}\ \underbrace{f}_{(1)} = (\mathsf{fn}x.x)\ \mathsf{in}\ (\underbrace{f}_{(2)}\,\underline{T}, \underbrace{f}_{(3)}\,\mathsf{nil}) :: (\mathsf{bool}, [X])$$

then we can observe that the occurrence of f labelled (2) has implicit type $\mathsf{bool} \to \mathsf{bool}$ and that labelled (3) has implicit type $[X] \to [X]$. Now, the principal type of $\mathsf{fn}x.x$ is $Y \to Y$ and both of the implicit types of f are substitution instances of this principal type, with $S = \langle Y \mapsto \mathsf{bool}\rangle$ and $S = \langle Y \mapsto [X]\rangle$, respectively.

Motivation 8.5.3 We can summarize the last example by noting that *Variables which are bound in local declarations (such as f above) can have polymorphic instances in the body of the declaration.*

It is only possible for *bound* variables to possess polymorphic instances. Now, $\mathbb{PFUN}$ has one other variable binding operation, that found in function abstractions $\mathsf{fn}x.E$. Can such bound variables have polymorphic instances within the scope of $\mathsf{fn}x$ abstractions? The answer is in fact no. An example illustrates this.

Examples 8.5.4

(1) Show that the implicit type of f in $\vdash \mathsf{let}\ f = \mathsf{fn}x.x\ \mathsf{in}\ f\,\underline{3} :: \mathsf{int}$ is $\mathsf{int} \to \mathsf{int}$.

The deduction tree must look like

$$\dfrac{\dfrac{T_1}{\vdash \mathsf{fn}x.x :: \sigma} \qquad \dfrac{T_2}{\vdash (\mathsf{fn}x.x)\,\underline{3} :: \mathsf{int}}}{\vdash \mathsf{let}\ f = \mathsf{fn}x.x\ \mathsf{in}\ f\,\underline{3} :: \mathsf{int}}\ \text{LET}$$

(for some σ) and it is easy to produce the remainder of the deduction tree T_2 to obtain $\vdash \mathsf{fn}x.x :: \mathsf{int} \to \mathsf{int}$. This is the implicit type of f (and of course we can take σ to be this type—what is T_1?).

(2) $\mathsf{fn}\, f.(f\,\underline{T}, f\,\mathsf{nil})$ is not typable (in the empty context) in $\mathbb{PFUN}$.

To see this, we derive a possible deduction tree.

$$\dfrac{\dfrac{\mathcal{D} \qquad \dfrac{\dfrac{}{f :: \sigma_2 \vdash f :: \sigma_7 \to \sigma_5}\ \text{VAR} \qquad \dfrac{}{f :: \sigma_2 \vdash \mathsf{nil} :: \sigma_7 = [\sigma_8]}\ \text{NIL}}{f :: \sigma_2 \vdash f\,\mathsf{nil} :: \sigma_5}\ \text{AP}}{f :: \sigma_2 \vdash (f\,\underline{T}, f\,\mathsf{nil}) :: \sigma_3 = (\sigma_4, \sigma_5)}}{\vdash \mathsf{fn}\, f.(f\,\underline{T}, f\,\mathsf{nil}) :: \sigma_1 = \sigma_2 \to \sigma_3}\ \text{ABS}$$

where $\mathcal{D}$ is

$$\dfrac{\dfrac{}{f :: \sigma_2 \vdash f :: \sigma_6 \to \sigma_4}\ \text{VAR} \qquad \dfrac{}{f :: \sigma_2 \vdash \underline{T} :: \sigma_6 = \mathsf{bool}}\ \text{TRUE}}{f :: \sigma_2 \vdash f\,\underline{T} :: \sigma_4}\ \text{AP}$$

We conclude from the two instances of VAR that $\sigma_2 = [\sigma_8] \to \sigma_5$ and $\sigma_2 = \mathsf{bool} \to \sigma_4$ so that $[\sigma_8] = \mathsf{bool}$, a contradiction. (Also $\sigma_5 = \sigma_4$ but this does not tell us anything useful).

8.6 A Type Inference Algorithm

Motivation 8.6.1 Given a program expression P with no free variables, how do we calculate or infer its principal type, or show that it does not have one? There is an algorithm (due to Hindley; Damas-Milner) which will compute this. This algorithm bears some resemblance to that given in Chapter 3, but is complicated by the fact that we now have type variables and higher order functions. In order to illustrate it, we shall restrict attention to a fragment of $\mathbb{PFUN}$. Let us give the definition of the restricted language, and then the algorithm itself.

Definitions 8.6.2 The types and expressions are given by

$$\begin{array}{lcl} \sigma & ::= & \mathsf{int} \mid X \mid \sigma \to \sigma \\ E & ::= & \underline{n} \mid E\ \mathit{iop}\ E \mid \mathsf{fn}x.E \mid E\,E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E \end{array}$$

$$
\begin{array}{rcl}
MGU(\sigma,\sigma) & = & \langle\rangle \qquad \text{here } \sigma \text{ is any type} \\
MGU(X,Y) & = & \langle X \mapsto Y \rangle \qquad \text{here } X \text{ and } Y \text{ are distinct variables} \\
MGU(X,\sigma) & = & \begin{cases} \text{here } \sigma \text{ is either } \mathsf{int} \text{ or a function type} \\ \langle X \mapsto \sigma \rangle & \text{if } X \notin TV(\sigma) \\ \mathit{FAIL} & \text{otherwise} \end{cases} \\
MGU(\sigma,X) & = & \begin{cases} \text{here } \sigma \text{ is either } \mathsf{int} \text{ or a function type} \\ \langle X \mapsto \sigma \rangle & \text{if } X \notin TV(\sigma) \\ \mathit{FAIL} & \text{otherwise} \end{cases} \\
MGU(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) & = & S_2 \cdot S_1 \\
 & & \text{where} \\
 & & \quad \sigma_1,\sigma_2,\tau_1,\tau_2 \text{ are any types} \\
 & & \quad S_1 \stackrel{\text{def}}{=} MGU(\sigma_1,\tau_1) \\
 & & \quad S_2 \stackrel{\text{def}}{=} MGU(S_1\{\sigma_2\}, S_1\{\tau_2\}) \\
 & & \quad \mathit{FAIL} \text{ otherwise} \\
MGU(\mathsf{int}, \sigma \to \tau) & = & \mathit{FAIL} \qquad \text{here } \sigma,\tau \text{ are any types} \\
MGU(\sigma \to \tau, \mathsf{int}) & = & \mathit{FAIL} \qquad \text{here } \sigma,\tau \text{ are any types}
\end{array}
$$

Table 8.1: The Most General Unifier Algorithm

Suppose that σ and τ are any two types. We say that these types are **unifiable** if there exists a type substitution S for which $S\{\sigma\} = S\{\tau\}$. Also, a **most general unifer** is a unifier for which, given another unifer S', there exists T for which $S' = T \cdot S$. We can define a function *MGU* which given σ and τ will return a most general unifier if there is one, or will otherwise *FAIL*. We define this by specifying the possible cases for the input in Table 8.1.

A **typing** for the judgement

$$x_1 :: \sigma_1, \ldots, x_n :: \sigma_n \vdash E \qquad \dagger$$

is a pair (S,τ) for which

$$x_1 :: S\{\sigma_1\}, \ldots, x_n :: S\{\sigma_n\} \vdash E :: \tau$$

may be deduced using the $\mathbb{PFUN}$ type assignment rules. Such a typing is said to be **principal** if given any other (S',τ')there is some T for which $S' = T \cdot S$ and $\tau' = T\{\tau\}$.

There is a type inference function Φ which given any input of the form † will either return a principal typing, or *FAIL* if there is none. To define it, we need a little more notation. Given a context $\Gamma = x_1 :: \sigma_1, \ldots, x_n :: \sigma_n$ let us write (by abusing notation) $TV(\Gamma)$ for the set

$$TV(\sigma_1) \cup \ldots \cup TV(\sigma_n)$$

We shall also write $S\{\Gamma\}$ to mean

$$x_1 :: S\{\sigma_1\}, \ldots, x_n :: S\{\sigma_n\}$$

and we define $S\{\varnothing\} \stackrel{\text{def}}{=} \varnothing$. The function Φ is given in Table 8.2.

Examples 8.6.3

(1) We claimed that the principal type of $\mathsf{fn}\, x.x$ is $X \to X$. We have

$$\Phi(\varnothing \vdash \mathsf{fn}\, x.x) = (S^V, S\{V\} \to \tau)$$

where

$$(S, \tau) = \Phi(x :: V \vdash x) = (\langle\rangle, V).$$

Thus $\Phi(\varnothing \vdash \mathsf{fn}\, x.x) = (\langle\rangle^V, \langle\rangle\{V\} \to V) = (\langle\rangle, V \to V)$. So, up to a renaming of type variables, the principal type is $V \to V$.

(2) We calculate $\Phi(x :: X \vdash \mathsf{fn}\, f.f\,x)$. This is $(S^V, S\{V\} \to \tau)$ where

$$(S, \tau) = \Phi(x :: X, f :: V \vdash f\,x) = ({S_3}^U \cdot S_2 \cdot S_1, S_3\{U\})$$

where

$$(S_1, \tau_1) = \Phi(x :: X, f :: V \vdash f) = (\langle\rangle, V)$$

and

$$(S_2, \tau_2) = \Phi(x :: X, f :: V \vdash x) = (\langle\rangle, X)$$

and

$$S_3 = MGU(V, X \to U) = \langle V \mapsto (X \to U)\rangle \qquad U \notin \{\langle\rangle\{V\}, X\} = \{V, X\}$$

Therefore $(S, \tau) = (\langle V \mapsto (X \to U)\rangle, U)$ and so $\Phi(x :: X \vdash \mathsf{fn}\, f.f\,x) = (\langle\rangle, (X \to U) \to U)$.

$$\Phi(x_1 :: \sigma_1, \ldots x_n :: \sigma_n \vdash x_i) \quad = \quad (\langle\rangle, \sigma_i)$$

$$\Phi(x_1 :: \sigma_1, \ldots x_n :: \sigma_n \vdash y) \quad = \quad \mathit{FAIL} \quad \text{whenever } \forall i. y \neq x_i$$

$$\Phi(\Gamma \vdash \underline{n}) \quad = \quad (\langle\rangle, \mathsf{int})$$

$$\begin{aligned}\Phi(\Gamma \vdash E_1 \mathit{iop}\, E_2) \quad = \quad & (S_4 \cdot S_3 \cdot S_2 \cdot S_1, S_4\{\tau_2\}) \\ & \text{where} \\ & (S_1, \tau_1) = \Phi(\Gamma \vdash E_1) \\ & S_2 = MGU(\tau_1, \mathsf{int}) \\ & (S_3, \tau_2) = \Phi((S_2 \cdot S_1)\Gamma \vdash E_2) \\ & S_4 = MGU(\tau_2, \mathsf{int})\end{aligned}$$

$$\begin{aligned}\Phi(\Gamma \vdash \mathsf{fn}\, x.E) \quad = \quad & (S^V, S\{V\} \to \tau) \\ & \text{where} \\ & (S, \tau) = \Phi(\Gamma, x{:}V \vdash E) \\ & V \notin TV(\Gamma)\end{aligned}$$

$$\begin{aligned}\Phi(\Gamma \vdash E_1 E_2) \quad = \quad & ({S_3}^V \cdot S_2 \cdot S_1, S_3\{V\}) \\ & \text{where} \\ & (S_1, \tau_1) = \Phi(\Gamma \vdash E_1) \\ & (S_2, \tau_2) = \Phi(S_1\{\Gamma\} \vdash E_2) \\ & S_3 = MGU(S_2\{\tau_1\}, \tau_2 \to V) \\ & V \notin TV(S_2\{\tau_1\}) \text{ or } TV(\tau_2)\end{aligned}$$

Note the clause $V \notin TV(S_2\{\tau_1\})$ or $TV(\tau_2)$. When computing Φ, we should first calculate S_2, τ_1 and τ_2, and then, knowing what type variables they contain, choose V accordingly. In practise, however, we can select the variable V *first*, and then when computing S_2, τ_1 and τ_2, make sure there are no variable clashes. This does need care!!

$$\begin{aligned}\Phi(\Gamma \vdash \mathsf{let}\, x = E_1 \,\mathsf{in}\, E_2) \quad = \quad & (S_2 \cdot S_1, \tau_2) \\ & \text{where} \\ & (S_1, \tau_1) = \Phi(\Gamma \vdash E_1) \\ & (S_2, \tau_2) = \Phi(S_1\{\Gamma\} \vdash E_2[E_1/x])\end{aligned}$$

Table 8.2: The Type Inference Algorithm for $\mathbb{PFUN}$

(3)

$$\Phi(\varnothing \vdash \mathsf{let}\ x = \underline{4}\ \mathsf{in}\ \mathsf{fn}\ f.\mathsf{fn}\, g.\, f\,(g\,x)) = (S_2 \cdot S_1, \tau_2)$$

where
$(S_1, \tau_1) = \Phi(\varnothing \vdash \underline{4}) = (\langle\rangle, \mathsf{int})$
$(S_2, \tau_2) = \Phi(S_1\{\varnothing\} \vdash \mathsf{fn}\ f.\mathsf{fn}\, g.\, f\,(g\,\underline{4})) = (S^V, S\{V\} \to \tau)$

where

$$(S, \tau) = \Phi(f :: V \vdash \mathsf{fn}\, g.\, f\,(g\,\underline{4})) = ({S'}^{V'}, S'\{V'\} \to \tau') \qquad V' \notin \{V\}$$

where

$$(S', \tau') = \Phi(f :: V, g :: V' \vdash f\,(g\,\underline{4})) = ({A_3}^U \cdot A_2 \cdot A_1, A_3\{U\})$$

where

$$(A_1, \sigma_1) = \Phi(f :: V, g :: V' \vdash f) = (\langle\rangle, V)$$

and

$$(A_2, \sigma_2) = \Phi(f :: V, g :: V' \vdash g\,\underline{4}) = ({B_3}^{U'} \cdot B_2 \cdot B_1, B_3\{U'\})$$

and

$$A_3 = MGU(A_2\{\sigma_1\}, \sigma_2 \to U) \qquad U \notin TV(A_2\{\sigma_1\}) \cup TV(\sigma_2)$$

where

$$(B_1, \rho_1) = \Phi(f :: V, g :: V' \vdash g) = (\langle\rangle, V')$$

and

$$(B_2, \rho_2) = \Phi(f :: V, g :: V' \vdash \underline{4}) = (\langle\rangle, \mathsf{int})$$

and

$$B_3 = MGU(V', \mathsf{int} \to U') = \langle V' \mapsto (\mathsf{int} \to U')\rangle \quad \text{where } U' \notin \{V', \mathsf{int}\}$$

Therefore

$$(A_2, \sigma_2) = (\langle V' \mapsto (\mathsf{int} \to U') \rangle, U')$$

and

$$A_3 = MGU(V, U' \to U) = \langle V \mapsto (U' \to U) \rangle \quad \text{where } U \notin \{V, U'\}$$

Therefore

$$(S', \tau') = (\langle V \mapsto (U' \to U), V' \mapsto (\mathsf{int} \to U') \rangle, U)$$

and

$$(S, \tau) = (\langle V \mapsto (U' \to U) \rangle, (\mathsf{int} \to U') \to U)$$

Therefore

$$(S_2 \cdot S_1, \tau_2) = (\langle\rangle, (U' \to U) \to (\mathsf{int} \to U') \to U)$$

and thus the principal type of $\mathsf{let}\ x = \underline{4}\ \mathsf{in}\ \mathsf{fn}\ f.\mathsf{fn}\ g.\ f\ (g\ x)$ is $(U' \to U) \to (\mathsf{int} \to U') \to U$

9

The SECD Machine

9.1 Why Introduce the SECD Machine?

Motivation 9.1.1 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 could try defining a transition relation in a similar style to that for $\mathbb{IMP}$. Given a program P, it would be fairly easy for a human to give the complete transition sequence for P. However, this does require a careful scrutiny of the rules which define $\rightsquigarrow$: It is one thing to observe the rules and find, through a process of inspection, the unique P' for which $P \rightsquigarrow P'$. It is quite another to take P and *effectively compute* P'.

For example, while with practice seeing that

$$(\underline{3}+\underline{2}) \leq \underline{6} \qquad \rightsquigarrow \qquad \underline{5} \leq \underline{6}$$

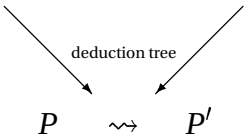
is easy (for humans!) one must not forget that proving this actually involves producing a deduction tree

$$\frac{\dfrac{}{\underline{3}+\underline{2} \rightsquigarrow \underline{5}}}{(\underline{3}+\underline{2}) \leq \underline{6} \rightsquigarrow \underline{5} \leq \underline{6}} \qquad (*)$$

Ultimately, 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 5). Thus establishing $P \mapsto P'$ will not require the construction of a deduction tree, as is the case with $\rightsquigarrow$ (which we illustrated with (*)):



An evaluation semantics, $\Downarrow^e$, is very much an opposite to the notion of a rewrite 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 9.1.

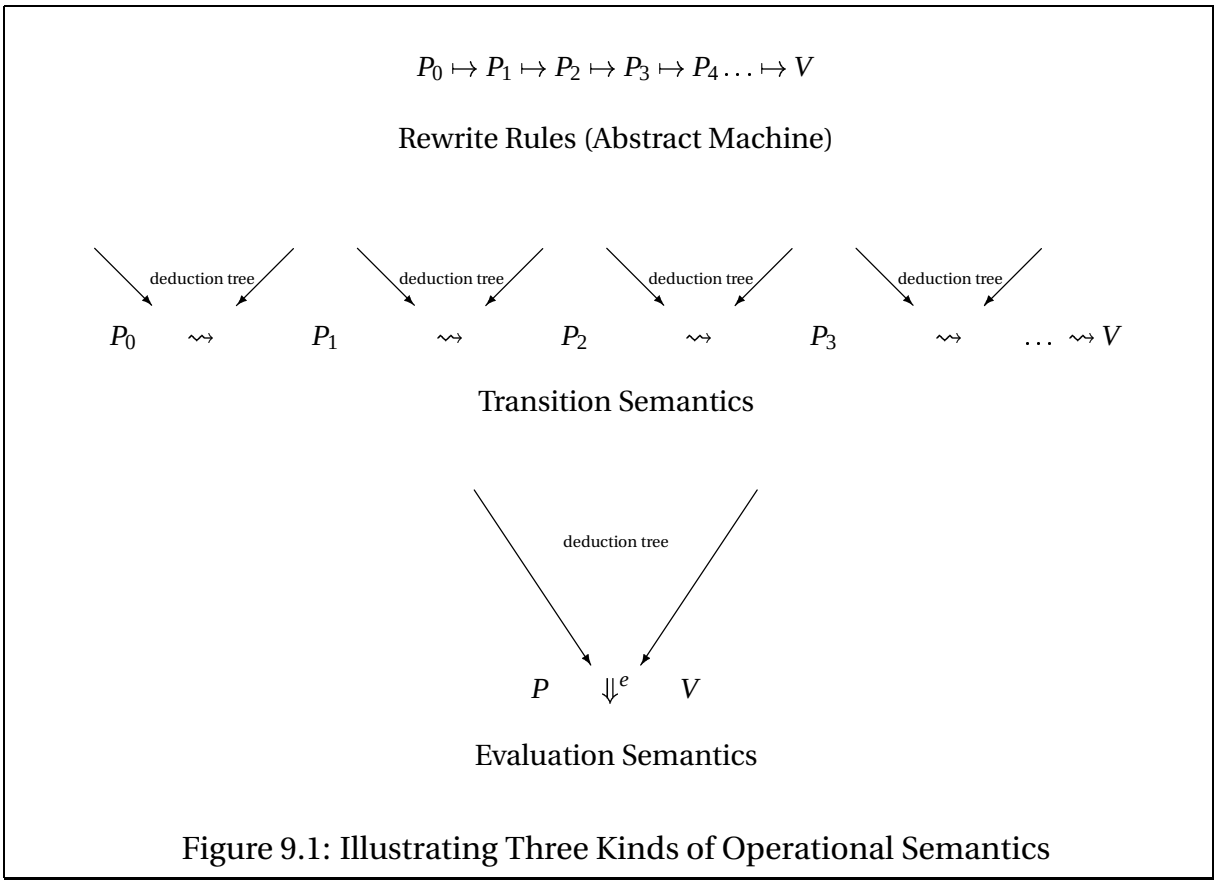


Figure 9.1: Illustrating Three Kinds of Operational Semantics

We will define a "machine", the *SECD machine*, which will "mechanically compute" certain programs to values, using rewrite rules. Landin invented the SECD machine. Originally, 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. There are lazy evaluation strategies for SECD machines, but such machines are slow.

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

$$E ::= x \mid \underline{c} \mid EE \mid \mathsf{fn}x.E \mid E\ op\ E$$

and the definition of program is just as in Chapter 6 but using this restricted set of terms. 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 $\mathbb{FUN}^e$.

⋙ **NOTE 9.1.2** ***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. Note also that*** E ***is used to denote SECD environments, and thus we denote expressions by*** M**.**

9.2 The Definition of the SECD Machine

Motivation 9.2.1 Before we outline the structure of the SECD machine, we introduce the notion of a closure. Consider $(\mathsf{fn}x.\mathsf{fn}y.(x+y))\,\underline{3}\,\underline{5} \rightsquigarrow (\mathsf{fn}y.(\underline{3}+y))\,\underline{5}$. The transition involves the substitution of $\underline{3}$ for the free variable x in $\mathsf{fn}y.(x+y)$. The SECD machine implements substitution via an environment which records the values of variables. The SECD machine represents $(\mathsf{fn}x.\mathsf{fn}y.(x+y))\,\underline{3}$, that is $(\mathsf{fn}y.(x+y))[\underline{3}/x]$, as a *closure*, which is a triple consisting of the bound variable, the scope, and the current environment:

$$\mathsf{CLO}(\quad y \quad , \quad x+y \quad , \quad x=\underline{3} \quad)$$

bound variable function scope environment

A closure stores data representing a function (plus current environment). When the SECD machine applies this particular function value to the argument $\underline{5}$, it restores the environment to $x=\underline{3}$, adds the binding $y=\underline{5}$, and evaluates $x+y$ in this updated environment.

The SECD machine has a typical configuration (S,E,C,D) consisting of four components:

(i) The **stack** S is a (possibly empty) list consisting of constants and closures. The empty list is denoted by $-$.

(ii) Let the symbol a denote either a constant $\underline{c}$ or a closure. The **environment** E takes the form $x_1 = a_1 : \ldots : x_n = a_n$, meaning that the variables $x_1, \ldots, x_n$ currently have the values $a_1, \ldots, a_n$ respectively. The environment may be empty $(-)$.

(iii) The **control** C is a list of commands. A command is either a term of the restricted language, an operator *op*, or the word APP.

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

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

It is essentially a list of triples $(S_1, E_1, C_1), (S_2, E_2, C_2), \ldots, (S_n, E_n, C_n)$ and serves as the function call stack.

Definitions 9.2.2 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	$-$

Note that the SECD machine described here is an **interpreter**. The program is executed directly, and is not compiled, as was the case for the CSS machine.h

If the control is non-empty, then its first command triggers a configuration rewrite, whereby the SECD machine changes to a new configuration. The rewrites are deterministic, and are determined by the element at the head of the Control list. Here are the possible rewrites:

A constant is pushed onto the stack:

S	S
E	E
C	$\underline{c} : C$
D	D

$\xrightarrow{\text{cst}}$

S	$\underline{c} : S$
E	E
C	C
D	D

The value of a variable is taken from the environment and pushed onto the stack. If the variable is x and E contains $x = a$ then a is pushed:

S	S
E	E
C	$x : C$
D	D

$\xrightarrow{\text{var}}$

S	$a : S$
E	E
C	C
D	D

An operator term is replaced by code to compute the arguments:

S	S
E	E
C	$M_1\ op\ M_2 : C$
D	D

$\xrightarrow{\text{optm}}$

S	S
E	E
C	$M_2 : M_1 : op : C$
D	D

An operator *op* is computed:

S	$\underline{c} : \underline{c'} : S$
E	E
C	$op : C$
D	D

$\xrightarrow{\text{op}}$

S	$\underline{c\ op\ c'} : S$
E	E
C	C
D	D

An abstraction is converted to a closure and then pushed onto the stack:

S	S
E	E
C	$\mathsf{fn}\, x.M : C$
D	D

$\xrightarrow{\text{abs}}$

S	$\mathsf{CLO}(x,M,E) : S$
E	E
C	C
D	D

A function application is replaced by code to compute the argument and the function with an explicit APP instruction:

S	S
E	E
C	$M_1 M_2 : C$
D	D

$\xrightarrow{\text{fapp}}$

S	S
E	E
C	$M_2 : M_1 : \mathsf{APP} : C$
D	D

The closure $\mathsf{CLO}(x,M,E')$ is called by creating a new configuration to evaluate M in the environment E', extended with a binding for the argument. The old configuration is saved in the dump:

S	$\mathsf{CLO}(x,M,E') : a : S$
E	E
C	$\mathsf{APP} : C$
D	D

$\xrightarrow{\text{clo}}$

S	$-$
E	$x = a : E'$
C	M
D	(S,E,C,D)

The function call terminates in a configuration where the Control is empty but the Dump is not. To return from the function, the machine restores the configuration (S,E,C,D) from the Dump, then pushes a onto the stack:

S	a
E	E'
C	$-$
D	(S,E,C,D)

$\xrightarrow{\text{res}}$

S	$a:S$
E	E
C	C
D	D

The result of the evaluation, say a, is obtained from a **final configuration** where the Control and Dump are empty, and a is the sole value on the stack:

S	a
E	$-$
C	$-$
D	$-$

9.3 Example Evaluations

Examples 9.3.1

(1) Use the SECD machine to calculate $(\mathsf{fn}y.y+\underline{2})\,\underline{4}$.

S	$-$
E	$-$
C	$(\mathsf{fn}y.y+\underline{2})\,\underline{4}$
D	$-$

$\xrightarrow{\text{fapp}}$

S	$-$
E	$-$
C	$\underline{4}:\mathsf{fn}y.y+\underline{2}:\mathsf{APP}$
D	$-$

$\xrightarrow{\text{cst}}$

S	$\underline{4}$
E	$-$
C	$\mathsf{fn}y.y+\underline{2}:\mathsf{APP}$
D	$-$

$\xrightarrow{\text{abs}}$

S	$\mathsf{CLO}(y,y+\underline{2},-):\underline{4}$
E	$-$
C	APP
D	$-$

$\xrightarrow{\text{clo}}$

S	$-$
E	$y=\underline{4}$
C	$y+\underline{2}$
D	$(-,-,-,-)$

$\xrightarrow{\text{optm}}$

S	$-$
E	$y=\underline{4}$
C	$\underline{2}:y:+$
D	$(-,-,-,-)$

$\xrightarrow{\text{cst}}$

S	$\underline{2}$
E	$y=\underline{4}$
C	$y:+$
D	$(-,-,-,-)$

$\xrightarrow{\text{var}}$

S	$\underline{4}:\underline{2}$
E	$y=\underline{4}$
C	$+$
D	$(-,-,-,-)$

$\xrightarrow{\text{op}}$

S	$\underline{6}$
E	$y=\underline{4}$
C	$-$
D	$(-,-,-,-)$

$\xrightarrow{\text{res}}$

S	$\underline{6}$
E	$-$
C	$-$
D	$-$

Hence $\mathsf{fn}y.(y+\underline{2})\,\underline{4} \Downarrow^e \underline{6}$.

(2) Use the SECD machine to calculate $(\mathsf{fn}x.x)\,\underline{3} \leq \underline{5}$.

S	$-$
E	$-$
C	$(\mathsf{fn}x.x)\,\underline{3} \leq \underline{5}$
D	$-$

$\xrightarrow{\text{optm}}$

S	$-$
E	$-$
C	$\underline{5}:(\mathsf{fn}x.x)\,\underline{3}:\leq$
D	$-$

$\xrightarrow{\text{cst}}$

S	$\underline{5}$
E	$-$
C	$(\mathsf{fn}x.x)\,\underline{3}:\leq$
D	$-$

$\xrightarrow{\text{fapp}}$

S	$\underline{5}$
E	–
C	$\underline{3} : \mathsf{fn}\,x.x : \mathsf{APP} :\leq$
D	–

$\overset{\text{cst}}{\longmapsto}$

S	$\underline{3} : \underline{5}$
E	–
C	$\mathsf{fn}\,x.x : \mathsf{APP} :\leq$
D	–

$\overset{\text{abs}}{\longmapsto}$

S	$\mathsf{CLO}(x,x,-) : \underline{3} : \underline{5}$
E	–
C	$\mathsf{APP} :\leq$
D	–

$\overset{\text{clo}}{\longmapsto}$

S	–
E	$x = \underline{3}$
C	x
D	$(\underline{5}, -, \leq, -)$

$\overset{\text{var}}{\longmapsto}$

S	$\underline{3}$
E	$x = \underline{3}$
C	–
D	$(\underline{5}, -, \leq, -)$

$\overset{\text{res}}{\longmapsto}$

S	$\underline{3} : \underline{5}$
E	–
C	$\leq$
D	–

$\overset{\text{op}}{\longmapsto}$

S	$\underline{F}$
E	–
C	–
D	–

Hence $(\mathsf{fn}\,x.x)\,\underline{3} \leq \underline{5} \Downarrow^e \underline{F}$.

A

Correctness of the CSS Machine

A.1 A proof of Correctness

Motivation A.1.1 In the appendix we give a proof of correctness for the *interpreted* CSS machine. It works just like the machine defined earlier, but programs are executed directly and are not compiled. **THE PROOF IS NOT EXAMINABLE**.

Definitions A.1.2 We make slight changes to the previous definitions. A CSS **code** C is a list which is produced by the following grammars:

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

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

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

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

The CSS re-writes are defined in Table A.1, where each rule R is written

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

Motivation A.1.3 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 4, the result matches that of the CSS machine, and vice versa. We make this precise in the following theorem:

Theorem A.1.4 For all $n \in \mathbb{Z}$, $b \in \mathbb{B}$, $P_1 :: \mathsf{int}$, $P_2 :: \mathsf{bool}$, $P_3 :: \mathsf{cmd}$ and $s, s_1, s_2 \in \mathit{States}$ we have

$$(P_1, s) \Downarrow (\underline{n}, s) \quad \mathit{iff} \quad \boxed{P_1}\boxed{-}\boxed{s} \longmapsto^t \boxed{-}\boxed{\underline{n}}\boxed{s}$$

$$(P_2, s) \Downarrow (\underline{b}, s) \quad \mathit{iff} \quad \boxed{P_2}\boxed{-}\boxed{s} \longmapsto^t \boxed{-}\boxed{\underline{b}}\boxed{s}$$

$$(P_3, s_1) \Downarrow (\mathsf{skip}, s_2) \quad \mathit{iff} \quad \boxed{P_3}\boxed{-}\boxed{s_1} \longmapsto^t \boxed{-}\boxed{-}\boxed{s_2}$$

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

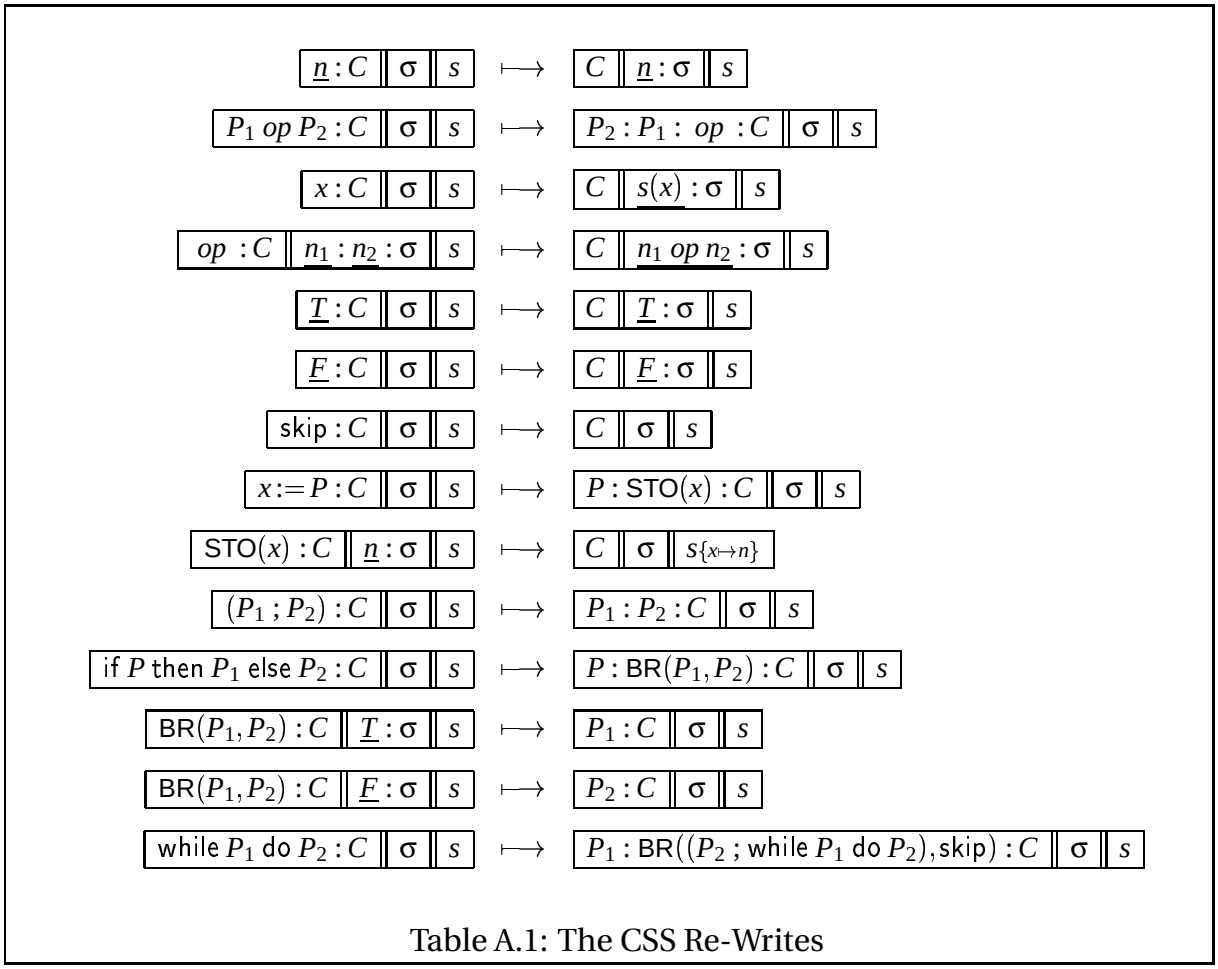


Table A.1: The CSS Re-Writes

Motivation A.1.5 The proof method for Theorem A.1.4 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 $k \in \mathbb{N}$,

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

If it is not immediately clear to you how Mathematical Induction will be used, then look ahead to page 98. We shall need a few preliminary results before we can prove the theorem.

Lemma A.1.6 The CSS machine re-writes are deterministic, that is each CSS configuration re-writes to a unique CSS configuration:

More precisely, if

$$\boxed{C \,\|\, \sigma \,\|\, s} \longmapsto \boxed{C_1 \,\|\, \sigma_1 \,\|\, s_1} \quad \textit{and} \quad \boxed{C \,\|\, \sigma \,\|\, s} \longmapsto \boxed{C_2 \,\|\, \sigma_2 \,\|\, s_2}$$

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

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

Lemma A.1.7 Given any sequence of CSS re-writes, we can (uniformly) extend both the code and stack of each configuration, without affecting the execution of the original code and stack:

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

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

implies

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

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

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

for all appropriate codes, stacks and states

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

implies

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

(*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,

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

implies

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

holds for each $k \leq k_0$, then

for all codes, stacks and states,

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

implies

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

Let us choose arbitrary codes, stacks and states for which

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

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

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

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

implies

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

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

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

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

We need to prove that

$$\boxed{\underline{n} : C_1 : C_3}\boxed{\sigma_1 : \sigma_3}\boxed{s_1} \longmapsto^{k_0+1} \boxed{C_2 : C_3}\boxed{\sigma_2 : \sigma_3}\boxed{s_2} \qquad (1)$$

By Lemma A.1.6 we must have[1]

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

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

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

But

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

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

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

$$\boxed{\mathsf{BR}(P_1,P_2):C_1}\boxed{\underline{T}:\sigma_1}\boxed{s_1} \longmapsto^{k_0+1} \boxed{C_2}\boxed{\sigma_2}\boxed{s_2}$$

We need to prove that

$$\boxed{\mathsf{BR}(P_1,P_2):C_1:C_3}\boxed{\underline{T}:\sigma_1:\sigma_3}\boxed{s_1} \longmapsto^{k_0+1} \boxed{C_2:C_3}\boxed{\sigma_2:\sigma_3}\boxed{s_2} \tag{4}$$

Now

$$\boxed{\mathsf{BR}(P_1,P_2):C_1}\boxed{\underline{T}:\sigma_1}\boxed{s_1} \longmapsto^1 \boxed{P_1:C_1}\boxed{\sigma_1}\boxed{s_1}$$

and so by induction we have

$$\boxed{P_1:C_1:C_3}\boxed{\sigma_1:\sigma_3}\boxed{s_1} \longmapsto^{k_0} \boxed{C_2:C_3}\boxed{\sigma_2:\sigma_3}\boxed{s_2} \tag{5}$$

But

$$\boxed{\mathsf{BR}(P_1,P_2):C_1:C_3}\boxed{\underline{T}:\sigma_1:\sigma_3}\boxed{s_1} \longmapsto^1 \boxed{P_1:C_1:C_3}\boxed{\sigma_1:\sigma_3}\boxed{s_1} \tag{6}$$

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

Lemma A.1.8 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

for all appropriate codes, stacks and states, if

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

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

$$\boxed{C_1}\boxed{\sigma}\boxed{s} \longmapsto^{k_1} \boxed{-}\boxed{\sigma'}\boxed{s'}$$

$$\boxed{C_2}\boxed{\sigma'}\boxed{s'} \longmapsto^{k_2} \boxed{-}\boxed{\sigma''}\boxed{s''}$$

where $k_1+k_2=k$.

[1] All we are saying here is that any sequence of re-write steps must have a unique form. We often use determinism of $\longmapsto$ in the next few pages, without always quoting Lemma A.1.6.

[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 $\underline{T}$.

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

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

and then consider the possible forms that C_1 can take.

(*Case* C_1 *is* while P_1 do $P_2 : C_1$):

We suppose that

$$\boxed{\text{while } P_1 \text{ do } P_2 : C_1 : C_2}\boxed{\sigma}\boxed{s} \longmapsto^{k_0+1} \boxed{-}\boxed{\sigma''}\boxed{s''}$$

and hence by Lemma A.1.6

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

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

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

and

$$\boxed{C_2}\boxed{\sigma'}\boxed{s'} \longmapsto^{k_2} \boxed{-}\boxed{\sigma''}\boxed{s''} \tag{2}$$

But

$$\boxed{\text{while } P_1 \text{ do } P_2 : C_1}\boxed{\sigma}\boxed{s} \longmapsto^{1} \boxed{P_1 : \mathsf{BR}((P_2 \text{ ; while } P_1 \text{ do } P_2), \text{skip}) : C_1}\boxed{\sigma}\boxed{s} \tag{3}$$

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

Lemma A.1.9 For all appropriate codes, stacks, states and natural numbers,

$$P :: \text{int } \textit{and } \boxed{P}\boxed{\sigma}\boxed{s} \longmapsto^{k} \boxed{-}\boxed{\sigma'}\boxed{s'} \quad \textit{implies}$$

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

and

$$P :: \text{bool } \textit{and } \boxed{P}\boxed{\sigma}\boxed{s} \longmapsto^{k} \boxed{-}\boxed{\sigma'}\boxed{s'} \quad \textit{implies}$$

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

Proof A lengthy, trivial Structural Induction. □

Proving Theorem A.1.4

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* $\Downarrow_{\text{OP}_1}$): 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

$$\begin{array}{rll} \boxed{P_1\ op\ P_2 \| - \| s} & \longmapsto & \boxed{P_2 : P_1 :\ op \| - \| s} \\ \text{by Lemma A.1.7 and inductive hypotheses} & \longmapsto^t & \boxed{P_1 :\ op \| \underline{n_2} \| s} \\ \text{by Lemma A.1.7 and inductive hypotheses} & \longmapsto^t & \boxed{op \| \underline{n_1} : \underline{n_2} \| s} \\ & \longmapsto & \boxed{- \| \underline{n_1\ op\ n_2} \| s} \end{array}$$

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!! 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* $\phi(k_0+1)$): 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 A.1.8 and A.1.9 we have, noting $P_2 :: \mathsf{int}$, that

$$\boxed{P_2 \;\|\; - \;\|\; s} \longmapsto^{k_1} \boxed{- \;\|\; \underline{n_2} \;\|\; s}$$
$$\boxed{P_1 :\ op \;\|\; \underline{n_2} \;\|\; s} \longmapsto^{k_2} \boxed{- \;\|\; \underline{n} \;\|\; s}$$

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

$$\boxed{P_1 \;\|\; \underline{n_2} \;\|\; s} \longmapsto^{k_{21}} \boxed{- \;\|\; \underline{n_1} : \underline{n_2} \;\|\; s}$$
$$\boxed{op \;\|\; \underline{n_1} : \underline{n_2} \;\|\; s} \longmapsto^{k_{22}} \boxed{- \;\|\; \underline{n} \;\|\; s} \qquad (1)$$

where $k_{21} + k_{22} = k_2$. So as $k_1 \leq k_0$, by Induction we deduce that $(P_2\,, s) \Downarrow (\underline{n_2}\,, s)$, and from Lemma A.1.9 that

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

Also, as $k_{21} \leq k_0$, 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 A.1.6 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,

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

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

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

for all $k \leq k_0$, then

$$\text{for all } P :: \mathsf{cmd}, s, s', \quad \boxed{P \;\|\; - \;\|\; s} \longmapsto^{k_0+1} \boxed{- \;\|\; \sigma \;\|\; s'} \quad \textit{implies} \quad \sigma = - \ \textit{and}\ (P\,, s) \Downarrow (\mathsf{skip}\,, s')$$

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

$$\boxed{P}\boxed{-}\boxed{s} \longmapsto^{k_0+1} \boxed{-}\boxed{\sigma}\boxed{s'}$$

We consider cases for P:

(*Case* P is $x := P$): Using Lemma A.1.6, we must have

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

and so by Lemmas A.1.8 and A.1.9 (and the typing rules)

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

where $k_1 + k_2 = k_0$. By determinism for (1) we have $\sigma = -$ and $s_{\{x \mapsto c\}} = s'$. By the first right to left implication for integer expressions (proved above) we have $(P, s) \Downarrow (\underline{c}, s)$. Hence $(x := P, s) \Downarrow (\mathsf{skip}, s_{\{x \mapsto c\}})$, and as $s_{\{x \mapsto c\}} = s'$ 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. □

A.2 CSS Executions

Examples A.2.1

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

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

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

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

$$
\begin{array}{lll}
\boxed{\mathsf{if}\ x \geq \underline{0}\ \mathsf{then}\ x := x - \underline{1}\ \mathsf{else}\ \mathsf{skip}} \,\|\, \boxed{-} \,\|\, \boxed{s} & \longmapsto & \boxed{x \geq \underline{0} : \mathsf{BR}(x := x - \underline{1}, \mathsf{skip})} \,\|\, \boxed{-} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{\underline{0} : x \geq : \mathsf{BR}(x := x - \underline{1}, \mathsf{skip})} \,\|\, \boxed{-} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{x : \geq : \mathsf{BR}(x := x - \underline{1}, \mathsf{skip})} \,\|\, \boxed{\underline{0}} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{\geq : \mathsf{BR}(x := x - \underline{1}, \mathsf{skip})} \,\|\, \boxed{\underline{1} : \underline{0}} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{\mathsf{BR}(x := x - \underline{1}, \mathsf{skip})} \,\|\, \boxed{\underline{T}} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{x := x - \underline{1}} \,\|\, \boxed{-} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{x - \underline{1} : \mathsf{STO}(x)} \,\|\, \boxed{-} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{\underline{1} : x : - : \mathsf{STO}(x)} \,\|\, \boxed{-} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{x : - : \mathsf{STO}(x)} \,\|\, \boxed{\underline{1}} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{- : \mathsf{STO}(x)} \,\|\, \boxed{\underline{1} : \underline{1}} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{\mathsf{STO}(x)} \,\|\, \boxed{\underline{0}} \,\|\, \boxed{s} \\
& \longmapsto & \boxed{-} \,\|\, \boxed{-} \,\|\, \boxed{s\{x \mapsto 0\}}
\end{array}
$$

Notation Index

Subject Index