University of Leicester

FUNCTIONAL PROGRAMMING THEORY

Course Notes

for lectures 18 to 36 of

MC 208

Dr. R. L. Crole

Department

of

Mathematics and Computer Science

Preface

These notes are to accompany the second half of the course MC 208. They contain all of the core material for this part of the course. For more motivation and background, as well as 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 in the notes. If you have any other (constructive) comments, please tell me about them.

Books recommended for the second eighteen lectures of MC 208 are

• "Semantics with Applications" by H. R. Neilson and F. Nielson. Wiley 1992.

• "The Semantics of Programming Languages: An Introduction Using Structured Operational Semantics" by Matthew Hennessy. Wiley 1990.

• "The Syntax and Semantics of Programming Languages" by David Watt. Prentice Hall Computer Science, 1991.

• "The Structure of Typed Programming Languages" by David Schmidt. MIT Press 1994.

• "Semantics of Programming Languages" by Carl Gunter. MIT Press 1992.

• "The Formal Semantics of Programming Languages" by Glynn Winskel. MIT Press 1993.

• "Functional Programming and Parallel Graph Reduction" by Rinus Plasmeijer and Marko van Eekelen. Addison Wesley 1993.

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

These notes contain some material based on lectures given by L. Paulson and A. M. Pitts of Cambridge University, and some material based on the above books.

Dr. R. L. Crole November 24, 2009

Contents

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 Discrete Structures (MC150).

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**."

• $\Longleftrightarrow$ means "if and only if". In proofs of theorems which are of the form

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

we shall write

($\Rightarrow$) ... proof that *statement 1* implies *statement 2* ...

($\Leftarrow$) ... proof that *statement 2* implies *statement 1* ...

• 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.*

α	alpha
β	beta
γ	gamma (lower case)
Γ	gamma (upper case)
δ	delta (lower case)
Δ	delta (upper case)
ϵ	epsilon
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

1.2 A Review of Sets

Definitions 1.2.1 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:

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

Recall that the **empty** set, $\varnothing$, is the set with no elements, and that we say a set S is a **subset** of a set A, written $S \subseteq A$, if any element of S is an element of A. Thus given sets A and S we could write this definition of subset as

$$S \subseteq A \qquad \Longleftrightarrow \qquad \forall x\, (x \in S \Longrightarrow x \in A). \qquad\qquad (*)$$

Note that $\Longleftrightarrow$ stands for "if and only if" and is used to give definitional equivalences. We could read $(*)$ as $S \subseteq A$ "is by definition the same as" for every x, $x \in S \Longrightarrow x \in A$. 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.

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 $A \times B$, 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 1.2.2

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

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

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

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

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

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

1.3 Relations

Motivation 1.3.1 The idea here is to see how we can formalise the the notion of relationships. Some examples of relationships are

- Ron *is_the_father_of* Roy;
- $0 \leq 5$;
- London *is_south_of* Leicester;

In each case, we have a pair of objects (for example 0 and 5) which are related in some way. Note that the order in which the objects are written down is important: $0 \leq 5$, but not $5 \leq 0$. Let us look for a general framework into which all of our examples fit.

Definitions 1.3.2 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 1.3.4. 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 1.3.3 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 Ron *is_the_father_of* 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 1.3.4 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 1.3.5 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 1.3.6 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 1.3.4.

(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 1.3.7 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 1.3.8 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 1.3.9 We can define an equivalence relation $\sim$ on the set $\mathbb{Z}$ of integers by setting

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

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

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

1.4 Inductively Defined Sets

Definitions 1.4.1 Let us first introduce some notation. Consider

statement 1 implies *statement 2*.

It is sometimes convenient to write this as

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

Consider

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 "$x \leq 4 \Longrightarrow x \leq 6$" as

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

The usefulness of this notation will soon become clear.

Motivation 1.4.2 As motivation for this section, consider the following:

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

(a) $0 \in E$, and

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

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

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

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

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

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

Definitions 1.4.3 A **rule** R for inductively defining a set denoted by I is a pair (H, c) where H is any finite set, and c is an element. Note that H might be $\varnothing$, in which case we say that R is a **base** rule. If H is non-empty we say R is an **inductive** rule. In the case that H is non-empty we might write $H = \{ h_1, \ldots, h_k \}$ where $1 \leq k$. We can write down a base rule $R = (\varnothing, c)$ for inductively defining the set I using the following notation

Base

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

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

Inductive

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

Note that the order of the statements h_1 `in` I h_2 `in` I ... h_k `in` I appearing above the line is irrelevant: the h_i are elements of the *set* H. You may like to think of the h_i as *hypotheses* and c as a *conclusion*. The notation h_i `in` I is meant to suggest that h_i is an element of the set I. (For the reason we have not written $h_i \in I$ see Proposition 1.4.7).

[1] So if $\mathcal{E}$ is the collection of sets satisfying (a) and (b), then E is the least element of $\mathcal{E}$ with respect to the subset ordering.

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

Inductively Defined Sets

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

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

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

Note that a base rule corresponds to the "base clause" and an inductive rule corresponds to the "inductive clause" as described in Motivation 1.4.2.

Example 1.4.4 A set[3] $\mathcal{R}$ of rules for defining the set E of even numbers is $\mathcal{R} = \{1, 2\}$ where

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

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

(1) $\quad 0 \in E$

(2) $\quad \forall e,\ e \in E \Longrightarrow e + 2 \in E.$

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

⋙ **Warning 1.4.5** ***Note that rule (2) is, strictly speaking, a rule schema, that is e is acting as a variable. There is a "rule" for each instantiation of e—hence the "$\forall e$" in the statement of closure of E above. A rule schema is a template for a collection of rules.***

Definitions 1.4.6 If I is inductively defined by a set of rules $\mathcal{R}$, a **deduction** of $x \texttt{ in } I$ is given by a list

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

where

(i) y_1 is a conclusion of a base rule;

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

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

(ii) for any $1 \leq i \leq m$, y_i is the conclusion of some rule R for which the hypotheses of R form a subset of $\{ y_1, \ldots, y_{i-1} \}$ (the subset can be empty—in this case R will be a base rule); and

(iii) $y_m = x$.

Note that $(*)$ is a *list*—the order of the y_i is crucial. Note also that in (ii) we sometimes say that y_i `in` I has been **deduced** using the rule R. The subset condition in (ii) simply ensures that the hypotheses appearing in the rule have already been deduced.

A **labelled** deduction of x `in` I looks like

$$\begin{array}{ll} y_1 \texttt{ in } I & (R_1) \\ y_2 \texttt{ in } I & (R_2) \\ \quad \ldots & \\ y_m \texttt{ in } I & (R_m) \end{array}$$

in which the sequence of y_i `in` I's is a deduction of x `in` I, and each R_i is the rule from $\mathcal{R}$ which has been used to deduce y_i `in` I.

Proposition 1.4.7 Suppose that I is inductively defined by a set of rules $\mathcal{R}$. Then

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

that is

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

Proof Non-examinable. Ask if you want details. □

⋙ **Warning 1.4.8** ***IC** means that the elements of the **I**nductively **defined set** I are **C**onstructed **by "applying" the rules in** $\mathcal{R}$—$x \in I$ if **there exists a deduction of** x `in` I. **IL** captures precisely the idea that I **is the L**east **set satisfying the rules, that is, there can be no elements of** I **other than those constructed by the rules**—$x \in I$ only if **there exists a deduction of** x `in` I. **Here, least refers to the subset ordering** $\subseteq$ **on sets.***

Examples 1.4.9

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

$$\frac{}{0 \texttt{ in } I}\,(a) \qquad \frac{i \texttt{ in } I}{i+3 \texttt{ in } I}\,(b) \qquad \frac{i \texttt{ in } I}{i-3 \texttt{ in } I}\,(c)$$

and informally you should think of the symbol i as a variable, that is, (b) and (c) are rule schemas. For example I being closed under (b) means that if i is *any* element of I $(i \in I)$,

so too is $i+3 \in I$. A deduction that $9 \in I$ is given by $0 \texttt{ in } I, 3 \texttt{ in } I, 6 \texttt{ in } I, 9 \texttt{ in } I$, and a labelled version of this deduction would be

$$\begin{array}{ll} 0 \texttt{ in } I & (a) \\ 3 \texttt{ in } I & (b) \\ 6 \texttt{ in } I & (b) \\ 9 \texttt{ in } I & (b) \end{array}$$

(2) 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[4]

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

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

As an example, let $\Sigma = \{a, b, c\}$. One possible labelled deduction that $abac \in \Sigma^*$ is

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

If we compare this labelled deduction with the general definition in Definitions 1.4.6, we see that $m = 6$, and $y_1 = a$, $y_2 = b$, etc to $y_6 = abac$. We have

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

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

$$\{a, c\} \subseteq \{a, b, ab, c\} = \{y_1, \ldots, y_{5-1}\}.$$

(iii) $y_m = y_6 = abac$.

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

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

$$\dfrac{\dfrac{\dfrac{}{a \texttt{ in } \Sigma^*}(1) \quad \dfrac{}{b \texttt{ in } \Sigma^*}(1)}{ab \texttt{ in } \Sigma^*}(2) \quad \dfrac{\dfrac{}{a \texttt{ in } \Sigma^*}(1) \quad \dfrac{}{c \texttt{ in } \Sigma^*}(1)}{ac \texttt{ in } \Sigma^*}(2)}{abac \texttt{ in } \Sigma^*}(2)$$

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

$$\frac{}{P \texttt{ in } Prpn}\,[P \in Var]\ (V) \qquad \frac{\phi \texttt{ in } Prpn \quad \psi \texttt{ in } Prpn}{\phi \wedge \psi \texttt{ in } Prpn}\,(\wedge)$$

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

Clause **IC** of the definition of an inductively defined set says that *Prpn* is closed under each of these rules. Thus the first rule says that any propositional variable P is a proposition. Also (for example) rule $\wedge$ tells us that if $\phi \in Prpn$ and $\psi \in Prpn$ then $\phi \wedge \psi \in Prpn$. Clause **IL** captures formally the requirement that propositions can only arise through applications of the above rules. Finally, an *example* of a labelled deduction that

$$((P \to Q) \vee (Q \to P)) \wedge R$$

is a proposition is

$P \texttt{ in } Prpn$	(V)
$Q \texttt{ in } Prpn$	(V)
$P \to Q \texttt{ in } Prpn$	$(\to)$
$Q \to P \texttt{ in } Prpn$	$(\to)$
$R \texttt{ in } Prpn$	(V)
$(P \to Q) \vee (Q \to P) \texttt{ in } Prpn$	$(\vee)$
$((P \to Q) \vee (Q \to P)) \wedge R \texttt{ in } Prpn$	$(\wedge)$

1.5 Closures of Relations

Motivation 1.5.1 Suppose that R is a binary relation on a set A. Recall (MC150) the reflexive closure of R is the *smallest*[5] binary relation on A, say R^r, which is reflexive and *contains* R. Intuitively, we could obtain R^r from R by adding relationships to R until we obtain a reflexive relation. We then stop adding relationships, because R^r is required to be smallest.

[5] with respect to the subset order, $\subseteq$.

For example, if $A \stackrel{\text{def}}{=} \{\, \alpha, \beta, \gamma \,\}$, and $R \stackrel{\text{def}}{=} \{\, (\alpha, \gamma), (\alpha, \alpha) \,\}$, the smallest reflexive relation on A which contains R must be

$$R^r \stackrel{\text{def}}{=} \{\, (\alpha, \gamma), (\alpha, \alpha), (\beta, \beta), (\gamma, \gamma) \,\}$$

where we have added the relationships (β, β) and (γ, γ) to R.

The symmetric or transitive closures of R can also be constructed by adding relationships to R to produce a symmetric or transitive relation. For example, if R contains (a, b) and (b, c), but not (a, c), we must add the latter relationship (amongst others) to R, to obtain R^t.

We can formulate the definitions of reflexive, symmetric and transitive closures as inductively defined sets—see MC150. The basic idea that an inductively defined set is the smallest set satisfying some rules is the key: for each kind of closure on R, we write down rules which say exactly how relationships are added to R to form the closure. However, the inductive definitions are difficult to work with, so we give alternative, but equivalent definitions.

Definitions 1.5.2 Let R be a binary relation on a set A. Recall that we write

$$R^n \stackrel{\text{def}}{=} \{\, (a, a') \mid \exists a_1, \ldots, a_n \in A \text{ such that } a = a_1 \; R \; a_2 \; R \; a_3 \ldots a_{n-1} \; R \; a_n = a' \,\}$$

So, informally, $a \; R^n \; a'$ just when there is a finite sequence of n relationships from a to a'.

Then

(i) $R^r \stackrel{\text{def}}{=} R \cup \{\, (a, a) \mid a \in A \,\}$;

(ii) $R^s \stackrel{\text{def}}{=} R \cup R^{op}$;

(iii) $R^t \stackrel{\text{def}}{=} \bigcup_{n=1}^{\infty} R^n$ [6]

Example 1.5.3 Let $A \stackrel{\text{def}}{=} \{\, \alpha, \beta, \gamma, \delta \,\}$ and $R \stackrel{\text{def}}{=} \{\, (\alpha, \beta), (\beta, \gamma), (\gamma, \delta) \,\}$. Then

• $R^r = R \cup \{\, (x, x) \mid x \in A \,\} = \{\, (\alpha, \beta), (\beta, \gamma), (\gamma, \delta)(\alpha, \alpha), (\beta, \beta), (\gamma, \gamma), (\delta, \delta) \,\}$.

• $R^t = \{\, (\alpha, \beta), (\beta, \gamma), (\gamma, \delta)(\alpha, \gamma), (\beta, \delta), (\alpha, \delta) \,\}$ where for example we can deduce that $(\alpha, \delta) \in R^t$ as follows: $\alpha \; R^t \; \delta \iff \exists m . \alpha \; R^m \; \delta$. But we can see that $\alpha \; R \; \beta \; R \; \gamma \; R \; \delta$, and so $\alpha \; R^3 \; \delta$, and hence that $\alpha \; R^t \; \delta$ as required.

1.6 Principles of Induction

Motivation 1.6.1 In this section we see how inductive techniques of proof which the reader has met before fit into the framework of inductively defined sets. A **property** is a statement which is either true or false. We shall write $Prop(x)$ to denote a property of x.

[6] If X_i is a set for each $i \geq 1$, then there is a set $Y \stackrel{\text{def}}{=} \{\, y \mid y \in X_1 \text{ or } y \in X_2 \text{ or } y \in X_3 \text{ or } \ldots \,\}$, that is, $y \in Y \iff \exists m . y \in X_m$ for any y. We usually denote Y by $\bigcup_{i=1}^{\infty} X_i$. Why?

For example, if $Prop(x) \stackrel{\text{def}}{=} x \geq 2$, then $Prop(3)$ is true and $Prop(0)$ is false. If $Prop(a)$ is *true* then we often say that $Prop(a)$ **holds**.

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

$$\frac{}{0 \texttt{ in } \mathbb{N}}\,(zero) \qquad \frac{n \texttt{ in } \mathbb{N}}{n+1 \texttt{ in } \mathbb{N}}\,(add1)$$

Suppose we wish to show that $Prop(n)$ holds for all $n \in \mathbb{N}$. Let

$$S \stackrel{\text{def}}{=} \{\ n \mid n \in \mathbb{N} \text{ and } Prop(n) \text{ holds}\ \}.$$

If we can show that $S = \mathbb{N}$, then certainly $Prop(n)$ holds for all $n \in \mathbb{N}$—for if $n \in \mathbb{N}$, then $n \in S$ and so $Prop(n)$ holds. Now, $S \subseteq \mathbb{N}$ by definition, so we can prove $S = \mathbb{N}$ by showing that S is closed under the rules *zero* and *add1* (for then $\mathbb{N} \subseteq S$ by **IL** and we already have $S \subseteq \mathbb{N}$) and this amounts to precisely what one needs to verify for Mathematical Induction:

• S is closed under *zero* iff $0 \in S$ iff $Prop(0)$; and

• S is closed under *add1*, iff for every natural number n, $n \in S$ implies $n+1 \in S$, iff for every natural number n, $Prop(n)$ implies $Prop(n+1)$.

Next we shall see how the Principle of Structural Induction for the propositions of first order logic fits into our general framework of inductively defined sets. Recall that this says in order to prove that a property $Prop(\phi)$ holds for all propositions ϕ it is enough to show that

• $Prop(R)$ holds for each propositional variable R;

• if $Prop(\phi)$ and $Prop(\psi)$ hold for any ϕ and ψ, then so do $Prop(\phi \wedge \psi)$, $Prop(\phi \vee \psi)$, $Prop(\phi \rightarrow \psi)$ and $Prop(\neg\phi)$.

Now, we have specified the collection $Prpn$ of propositions as an inductively defined set. If we put

$$S \stackrel{\text{def}}{=} \{\ \phi \mid \phi \in Prpn \text{ and } Prop(\phi)\ \}$$

then $S \subseteq Prpn$ by definition. If also S is closed under the rules defining $Prpn$, then $Prpn \subseteq S$ by property **IL**, and so $Prpn = S$. But then for any proposition ϕ we must have $\phi \in S$, and so $Prop(\phi)$. Thus: *showing S is closed under the rules defining $Prpn$ will prove that $Prop(\phi)$ holds for all ϕ.* Let us examine one (typical) part of proving that S is closed under the rules for defining $Prpn$. Take the rule

$$\frac{\phi \texttt{ in } Prpn \quad \psi \texttt{ in } Prpn}{\phi \wedge \psi \texttt{ in } Prpn}$$

Showing S is closed under this rule amounts to showing that for any ϕ and ψ, if $\phi \in S$ and $\psi \in S$, then $\phi \wedge \psi \in S$. But this is exactly proving that if $Prop(\phi)$ and $Prop(\psi)$ hold, then so does $Prop(\phi \wedge \psi)$. And this is just one of the steps which we check when applying the

Rule Induction

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

$$\text{for all } i \in I, \quad Prop(i).$$

Then all we need to do is

• for every base rule $\frac{}{b \text{ in } I} \in \mathcal{R}$ prove that (if $b \in I$ then) $Prop(b)$ holds; and

• for every inductive rule $\frac{h_1 \text{ in } I, \ldots, h_k \text{ in } I}{c \text{ in } I} \in \mathcal{R}$ prove that if $h_1 \in I$ and ... and $h_k \in I$, and $Prop(h_1)$ and $Prop(h_2)$ and ... and $Prop(h_k)$ all hold, so does[a] $Prop(c)$; that is

$$(h_j \in I \ \ \& \ \ Prop(h_j) \text{ where } 1 \leq j \leq k) \quad \Longrightarrow \quad Prop(c)$$

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

[a] and of course $c \in I$ follows!

Figure 1.2: Rule Induction

Principle of Structural Induction—the remaining steps correspond to showing the closure of S under the remaining rules used to define *Prpn*. Thus the Principle of Structural Induction is a consequence of the properties of the inductively defined set *Prpn*.

Definitions 1.6.2 We present a useful inductive principle which subsumes the two principles given above—we call it **Rule Induction**. See Figure 1.2.

To see that Rule Induction works, write

$$S \stackrel{\text{def}}{=} \{\ i \ \mid \ i \in I \text{ and } Prop(i) \text{ holds } \}.$$

Notice that checking the bulleted (•) instructions above amounts to verifying that S *is* closed under $\mathcal{R}$. Thus property **IL** of I tells us that $I \subseteq S$. Also, $S \subseteq I$ by definition. Hence $S = I$. So if i is *any* element of I, then $i \in S$, and so $Prop(i)$ holds.

Examples 1.6.3

(1) Let $\Sigma = \{ a, b, c \}$ and let a set[7] S of words be defined inductively by the rules

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

[7] 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

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

We prove that for every $w \in S$, $Prop(w)$ 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 $Prop(b)$ holds.

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

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

(*Rule (4)*): Suppose $w, w' \in S$ 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 for every $w \in S$, $Prop(w)$.

1.7 Recursively Defined Functions

Definitions 1.7.1 Let I be inductively defined by a set of rules $\mathcal{R}$, and A any set. A function $f\colon I \to A$ can be defined by

• specifying an element $f(b) \in A$ for every base rule $\frac{}{b \text{ in } I} \in \mathcal{R}$; 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 $\frac{h_1 \text{ in } I, \ldots, h_k \text{ in } I}{c \text{ in } I} \in \mathcal{R}$,

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 1.7.2

(1) The factorial function $F\colon \mathbb{N} \to \mathbb{N}$ 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.

2

Untyped Functional Languages

2.1 Introduction

Motivation 2.1.1 The first half of this course taught you to program in a typed, lazy, functional programming language, namely Haskell. Our aim now is to give a thorough account of the operational theory of languages like Haskell. We shall see how the syntax of programs can be rigorously defined, how to identify programs with different code but similar computational behaviour, and how to define a formal execution mechanism for a simple language. We shall also see how we can do simple formal reasoning about the behaviours of programs.

In this chapter, we shall first consider the theory of an untyped, lazy language called $\mathbb{UL}$. Apart from being untyped, it resembles Haskell. Let us proceed ...

2.2 The Syntax of $\mathbb{UL}$

Motivation 2.2.1 When studying Haskell, you wrote programs which made use of a particular language, and a particular way of putting words of the language together. Failure to do this meant your program would not run. We need a way to make the construction and structure of programs precise. Let us look at a simple program

$$\mathsf{sum}(l) = \underbrace{\mathsf{if}\ \ \mathsf{elist}(l)\ \ \mathsf{then}\ \ \underline{0}\ \ \mathsf{else}\ \ \mathsf{hd}(l) + \mathsf{sum}(\mathsf{tl}(l))}_{\text{body}}$$

and think about the structure of the program body. It has the form

$$\mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ N$$

where, for example, B is $\mathsf{elist}(l)$. The conditional expression requires three arguments, B, M and N, and to make this clear it is helpful to write the conditional as

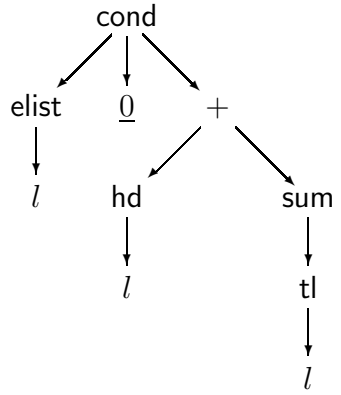
$$\mathsf{cond}(\mathsf{elist}(l)\ ,\ \underline{0}\ ,\ \mathsf{hd}(l) + \mathsf{sum}(\mathsf{tl}(l)))$$

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.

In fact, all of the programs we shall meet in the remainder of this course arise formally as "constructors applied to arguments". We can make the idea precise by thinking about finite trees, whose nodes are constructors and whose sub-trees are "sub-programs". For example, the body of the program written above is secretly just notation for the following 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 a constructors which take no arguments!!. We call constructors which take no arguments and which are not variables, such as $\underline{0}$ or the Boolean $\underline{T}$, *constants*. Let us move on to the precise definitions.

Definitions 2.2.2 We begin the formal definition of $\mathbb{UL}$. Let *Var* be a fixed, countably infinite, set of **variables**, for which we assume there is a specified enumeration (list). Thus

$$Var \stackrel{\text{def}}{=} \{ v_0, v_1, v_2, \dots \}.$$

We often denote variables by the letters x, y, z, u, v etc, but may on occasion use other letters. Let *Cst* be a set of **constants** where

$$Cst \stackrel{\text{def}}{=} \{ \underline{c} \mid c \in \mathbb{Z} \cup \mathbb{B} \}$$

and $\mathbb{Z}$ is the set of integers and $\mathbb{B} \stackrel{\text{def}}{=} \{ T, F \}$ is the set of Booleans. Let *Opr* be a set of **operators** given by

$$Opr \stackrel{\text{def}}{=} \{ =, \leq, \geq, <, >, +, -, * \}.$$

It is assumed that the intended meaning of these operators is the meaning you are familiar with from Haskell.

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

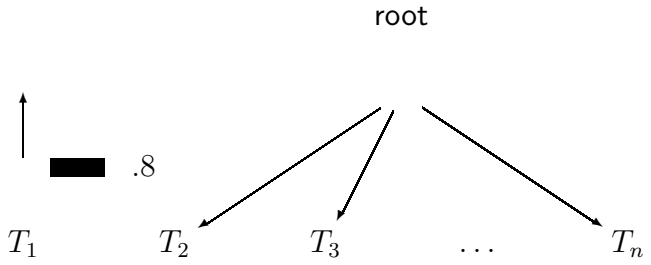
$$m \leq n = \begin{cases} T & \text{if } m \text{ is less than or equal to } n \\ F & \text{otherwise} \end{cases}$$

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

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

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

We now define the set $Term^{\mathbb{UL}}$ of **terms**. *For the time being, you can think of a term informally as a program, but we warned that they are not quite the same thing.* A term is in fact a finite tree, and the set of terms, $Term^{\mathbb{UL}}$, is a subset of the collection of all finite trees. We shall adopt the following notation for finite trees: If T_1, T_2, T_3 and so on to T_n is a (finite) sequence of finite trees, then we shall write $\mathsf{root}(T_1, T_2, T_3, \ldots, T_n)$ for the finite tree which has the form



and whose root is denoted by the symbol root.

The set of program **constructors** (recall Motivation 2.2.1) is given by

$$\{\, x, \underline{c}, op, \mathsf{cond}, \lambda, \mathsf{ap}, \mathsf{rec}, \mathsf{nil}, \mathsf{hd}, \mathsf{tl}, \mathsf{cons}, \mathsf{elist} \,\}$$

We say that $\underline{c}$ is a **nullary** constructor because it takes no arguments, cond is a **ternary** constructor because it takes three arguments, and so on in the obvious way.

The set $Term^{\mathbb{UL}}$ is inductively defined by the rules in Figure 2.1, where x can be *any* variable, $\underline{c}$ can be *any* constant, and op can be *any* operator.

Motivation 2.2.3 We have now given a very precise definition of the syntax which we shall use to denote terms. However, terms will be very difficult to read. We therefore introduce notational abbreviations (syntactic sugar) which apply to the formal definitions of terms:

Remark 2.2.4 We shall adopt the following notational abbreviations:

- We write $M\ op\ N$ for $op(M, N)$;
- if M then N else L for $\mathsf{cond}(M, N, L)$;
- $\lambda x.M$ for $\lambda(x, M)$;
- $M\,N$ for $\mathsf{ap}(M, N)$;
- $\mathsf{rec}\,x.M$ for $\mathsf{rec}(x, M)$; and

$$\frac{}{x \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{VAR} \qquad \frac{}{\underline{c} \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{CONST} \qquad \frac{M \ \texttt{in} \ Term^{\mathbb{UL}} \quad N \ \texttt{in} \ Term^{\mathbb{UL}}}{op(M,N) \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{OP}$$

$$\frac{M \ \texttt{in} \ Term^{\mathbb{UL}} \quad N \ \texttt{in} \ Term^{\mathbb{UL}} \quad L \ \texttt{in} \ Term^{\mathbb{UL}}}{\mathsf{cond}(M,N,L) \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{COND}$$

$$\frac{M \ \texttt{in} \ Term^{\mathbb{UL}}}{\lambda(x,M) \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{ABS} \qquad \frac{M \ \texttt{in} \ Term^{\mathbb{UL}} \quad N \ \texttt{in} \ Term^{\mathbb{UL}}}{\mathsf{ap}(M,N) \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{AP}$$

$$\frac{M \ \texttt{in} \ Term^{\mathbb{UL}}}{\mathsf{rec}(x,M) \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{REC} \qquad \frac{}{\mathsf{nil} \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{NIL} \qquad \frac{M \ \texttt{in} \ Term^{\mathbb{UL}}}{\mathsf{hd}(M) \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{HD}$$

$$\frac{M \ \texttt{in} \ Term^{\mathbb{UL}}}{\mathsf{tl}(M) \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{TL} \qquad \frac{M \ \texttt{in} \ Term^{\mathbb{UL}} \quad N \ \texttt{in} \ Term^{\mathbb{UL}}}{\mathsf{cons}(M,N) \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{CONS}$$

$$\frac{M \ \texttt{in} \ Term^{\mathbb{UL}}}{\mathsf{elist}(M) \ \texttt{in} \ Term^{\mathbb{UL}}}\ \text{ELIST}$$

Figure 2.1: Rules for Generating the Inductively Defined Set $Term^{\mathbb{UL}}$

• $M : N$ for $\mathsf{cons}(M, N)$.

We shall often prove propositions of the form $\forall M \in \mathit{Term}^{\mathbb{UL}}.\mathit{Prop}(M)$ by using Rule Induction. Sometimes we say instead that the proof uses "induction on the (finite tree) structure of M". Why is this?

Example 2.2.5 Let us give an example deduction tree[1] for a simple term. We shall show that $\mathsf{cond}(=(x, \underline{4}), \underline{0}, +(x, \underline{5}))$ is a term, that is,

$$\mathsf{cond}(=(x, \underline{4}), \underline{0}, +(x, \underline{5})) \in \mathit{Term}^{\mathbb{UL}}.$$

The formal tree looks like this:

$$\dfrac{\dfrac{\dfrac{}{x \ *}\,\text{VAR} \quad \dfrac{}{\underline{4} \ *}\,\text{CONST}}{=(x, \underline{4}) \ *}\,\text{OP} \quad \dfrac{}{\underline{0} \ *}\,\text{CONST} \quad \dfrac{\dfrac{}{x \ *}\,\text{VAR} \quad \dfrac{}{\underline{5} \ *}\,\text{CONST}}{+(x, \underline{5}) \ *}\,\text{OP}}{\mathsf{cond}(=(x, \underline{4}), \underline{0}, +(x, \underline{5})) \ *}\,\text{COND}$$

and the corresponding tree using informal notation looks like:

$$\dfrac{\dfrac{\dfrac{}{x \ *}\,\text{VAR} \quad \dfrac{}{\underline{4} \ *}\,\text{CONST}}{x = \underline{4} \ *}\,\text{OP} \quad \dfrac{}{\underline{0} \ *}\,\text{CONST} \quad \dfrac{\dfrac{}{x \ *}\,\text{VAR} \quad \dfrac{}{\underline{5} \ *}\,\text{CONST}}{x + \underline{5} \ *}\,\text{OP}}{\mathsf{if}\ x = \underline{4}\ \mathsf{then}\ \underline{0}\ \mathsf{else}\ x + \underline{5} \ *}\,\text{COND}$$

where $*$ denotes $\mathtt{in}\ \mathit{Term}^{\mathbb{UL}}$ to save space.

Definitions 2.2.6 Note that each "rule" in Figure 2.1 is really a rule schema. The rule COND means that if M, N and L are *any* terms, then so too is $\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ L$. We can also give the definition of the set of terms using a grammar, which uses less space than the full set of rules, and is clearer to read:

M	::=	x	variables
	|	$\underline{c}$	constant
	|	$M\ op\ M$	operator
	|	$\mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M$	conditional
	|	$\lambda x.M$	function term
	|	$M\,M$	function application
	|	$\mathsf{rec}\,x.M$	recursive term
	|	nil	empty list
	|	$\mathsf{hd}(M)$	head of list
	|	$\mathsf{tl}(M)$	tail of list
	|	$M : M$	cons for lists
	|	$\mathsf{elist}(M)$	test for empty list

Remember that the grammar is nothing other than a shorthand description of the rules in Figure 2.1. It reads: "any term is either a variable x, or a constant $\underline{c}$, or $M\ op\ N$ provided that M and N are already terms ...".

[1] See page 11

Motivation 2.2.7 The intended meanings of most of the terms are just what you would expect from Haskell, except for $\mathsf{rec}\, x.M$. In order to explain its meaning, if P and P' are two programs, we shall write $P \rightsquigarrow P'$ to mean that P "computes in one step" to P' (this notation will be defined properly on page 36). We shall also write $M[N/v]$ to mean "M where v is replaced by N". For example,

$$(\underline{2}+\underline{5})+\underline{1} \rightsquigarrow \underline{7}+\underline{1} \rightsquigarrow \underline{8} \qquad \text{and} \qquad (x+y)[\underline{4}/y] = x+\underline{4}.$$

`\x->M` in Haskell corresponds to $\lambda x.M$ in $\mathbb{UL}$, and is code for the program which is a function whose effect is to map x to M. More carefully, if $f \stackrel{\text{def}}{=} \lambda x.M$, then $f\, a \rightsquigarrow M[a/x]$. Thus $\lambda x.x+\underline{2}$ is a program whose intended meaning is the function which "adds 2", and we can write (for example) $(\lambda x.x+\underline{2})\,\underline{4} \rightsquigarrow \underline{4}+\underline{2}$.

Now, $\mathsf{rec}\, x.M$ is a recursive program on x, which is specified by the code in M. We ilustrate by example. Write $R \stackrel{\text{def}}{=} \mathsf{rec}\, x.M$. The program R "computes in one step" to $M[R/x]$. Thus if we take $M \stackrel{\text{def}}{=} \underline{0} : x$, then

$$R \rightsquigarrow (\underline{0} : x)[R/x] \equiv \underline{0} : R \rightsquigarrow \underline{0} : (\underline{0} : R) \rightsquigarrow \ldots$$

and so R is a *program which recursively evaluates to an infinite list of zeros.* We call each step in the computation of R an **unfolding**.

Remark 2.2.8 We shall adopt a few conventions to make terms more readable:

• In general, we shall write our "formal" syntax in an informal manner, using brackets "(" and ")" to disambiguate where appropriate—recall that in Haskell one can add such brackets to structure programs. For example, the term $\mathsf{ap}(\lambda(x, M), N)$ (which is unambiguous) will not be written $\lambda x.M\,N$ according to the abbreviation in Remark 2.2.4 (which is ambiguous) but will be written $(\lambda x.M)\,N$.

• We also *drop* brackets on other occasions. For example, we take λ-term $\lambda x.M$ to mean $\lambda x.(M)$. Thus we can write $\lambda x.\lambda y.y+\underline{2}$ instead of the more clumsy $\lambda x.(\lambda y.(y+\underline{2}))$. A similar convention applies to $\mathsf{rec}\, x.M$. We call M the **body** of $\lambda x.M$ and $\mathsf{rec}\, x.M$.

• $M_1M_2M_3 \ldots M_n$ is shorthand for $(\ldots((M_1M_2)M_3)\ldots M_n)$. We say that the application constructor (ap) **associates** to the left. For example, $M_1M_2M_3$ is short for $(M_1\, M_2)\, M_3$ (which is in turn a shorthand notation for the tree denoted by $\mathsf{ap}(\mathsf{ap}(M_1, M_2), M_3)$.

• We shall write $M : N : L$ for $M : (N : L)$ and say that the cons constructor (cons) associates to the right.

• The *op* constructors associate to the left. Thus the term $\underline{3}$ *op* $\underline{10}$ *op* $\underline{5}$ is shorthand for ($\underline{3}$ *op* $\underline{10}$) *op* $\underline{5}$.

• We take $\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ L$ to mean $\mathsf{if}\ (M)\ \mathsf{then}\ (N)\ \mathsf{else}\ (L)$.

Examples 2.2.9 Examples of actual terms, that is, elements of $Term^{\mathbb{UL}}$ are

(1) x;

(2) $\mathsf{hd}(\mathsf{cons}(\underline{2}, \underline{4}))$;

(3) $\mathsf{ap}(x, \mathsf{ap}(x, y))$;

(4) $\lambda(x, \lambda(y, \mathsf{ap}(\mathsf{ap}(y, x), \lambda(z, z))))$;

(5) $\mathsf{rec}(x, op(x, \underline{2}))$; and

(6) $\mathsf{cons}(x, \mathsf{cons}(\underline{2}, \underline{3}))$.

These would normally be written as

(1) x;

(2) $\mathsf{hd}(\underline{2} : \underline{4})$;

(3) $x\,(x\,y)$;

(4) $\lambda x.\lambda y.y\,x\,(\lambda z.z)$;

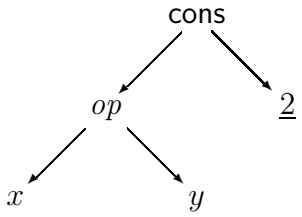
(5) $\mathsf{rec}\,x.x\ op\ \underline{2}$; and

(6) $x : \underline{2} : \underline{3}$.

⋙ **Warning 2.2.10** ***There are many terms which do not represent "commonsense" programs. By commonsense, we mean "well typed" programs. For example,*** $\mathsf{hd}(\underline{2} + \underline{3})$, $\underline{T} - \underline{3}$ ***and*** $\underline{4} * (\underline{T} : \underline{F})$ ***are all terms. In Chapter 4 we shall show how to add typing information, which will prevent such terms ever arising.***

2.3 Free and Bound Variables

Motivation 2.3.1 We have seen the definition of a term as a finite tree which is formed using certain rules. It is often convenient to refer to "parts" of a term. More precisely, such "parts" are subtrees of terms. We refer to such subtrees as subterms. For example, $x\ op\ y$ is a subterm of $(x\ op\ y) : \underline{2}$, and this fact looks transparent if we draw the tree which $(x\ op\ y) : \underline{2}$ denotes:



We now make this idea precise:

Definitions 2.3.2 We shall use the symbol $\equiv$ to mean **syntactic identity**. Two objects are syntactically identical iff they "symbolically the same". Thus, for example $2+2 \equiv 2+2$, but $2 + 2 \not\equiv 4$.

We shall define the notion of a subterm of a term. We shall specify a binary relation $\lhd$ (meaning "is a **subterm** of") on $Term^{\mathbb{UL}}$ by the following clauses:

- $M \lhd M$ for all terms M;
- $S \lhd M \; op \; N \iff S \lhd M$ or $S \lhd N$;
- $S \lhd \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ L \iff S \lhd M$ or $S \lhd N$ or $S \lhd L$;
- $S \lhd \lambda x.M \iff S \equiv x$ or $S \lhd M$;
- $S \lhd M\,N \iff S \lhd M$ or $S \lhd N$;
- $S \lhd \mathsf{rec}\,x.M \iff S \equiv x$ or $S \lhd M$;
- $S \lhd \mathsf{hd}(M) \iff S \lhd M$;
- $S \lhd \mathsf{tl}(M) \iff S \lhd M$;
- $S \lhd M : N \iff S \lhd M$ or $S \lhd N$; and
- $S \lhd \mathsf{elist}(M) \iff S \lhd M$.

We say that a variable x **occurs** in a term M if $x \lhd M$. We say that a term M lies in the **scope** of λy or $\mathsf{rec}\,y$ in a term of the form $\lambda y.M$ or $\mathsf{rec}\,y.M$ respectively.

Example 2.3.3 $u + \underline{2}$ is the scope of λu in $\lambda x.(\lambda u.u + \underline{2})\,z$. Example subterms are

$$z \lhd \lambda x.(\lambda u.u + \underline{2})\,z \text{ and } \lambda u.u + \underline{2} \lhd \lambda x.(\lambda u.u + \underline{2})\,z.$$

If $N \stackrel{\text{def}}{=} \lambda x.xxy\underline{x}zx$ then the underlined x is the *fourth* occurrence of x in N. x occurs in N five times.

Motivation 2.3.4 The intended meaning of $\lambda x.x + \underline{2}$ is the function which adds 2 to its argument. What about $\lambda y.y + \underline{2}$? Well, it too should be a function which adds 2. The name of the variable used to form such a term is not relevant to the intended meaning of the term—the variables x and y are said to be *bound*. However, the terms $x + \underline{2}$ and $y + \underline{2}$ are certainly different—the value of each term 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*. Let us give the full definitions:

Definitions 2.3.5 One reason for defining the notion of a subterm is so that we can give a formal definition of *free* and *bound* variables. Suppose that x is a variable which does occur in a term M—of course x may occur more than once, possibly many times. Each *occurrence* of x (in M) is either free or bound. We say that an occurrence of x is **bound** in M if the occurrence of x in M is in a subterm of the form $\lambda x.N$ or $\mathsf{rec}\,x.N$—this means that whenever $\lambda\,x$ or $\mathsf{rec}\,x$ appear in a term, only those occurrences of x which appear in the *scopes*[2] of $\lambda\,x$ or $\mathsf{rec}\,x$ are bound (as well as the occurrence of x immediately after the

[2]See Definitions 2.3.2

λ or rec !!). If there is an occurrence of x in such N then we say that occurrence of x has been **captured** by (the scope of) λx or $\mathsf{rec}\, x$ to mean that the occurrence of x is bound by the respective λx or $\mathsf{rec}\, x$. An occurrence of x in M is **free** iff the occurrence of x is not bound. Before reading on, take a look at Examples 2.3.6.

We shall write $var(M)$ for the set of all variables which occur in M, that is

$$var(M) \stackrel{\text{def}}{=} \{\, x \mid x \in Var \text{ and } x \lhd M \,\}.$$

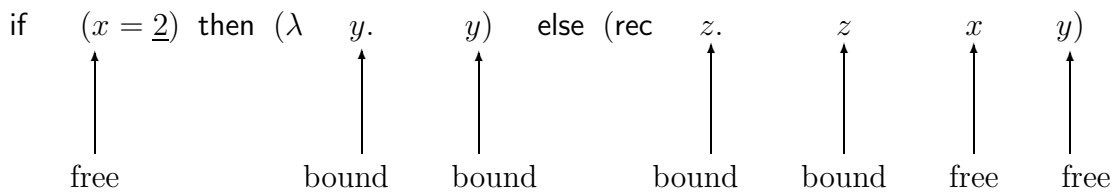
We can give a recursive definition of the set $var(M)$ which is obvious and omitted (cf the definition of $fvar(M)$ which follows). We write $fvar(M)$ for the set of variables which have free occurrences in M. We can define this recursively by the following (obvious!) clauses:

- $fvar(x) \stackrel{\text{def}}{=} \{\, x \,\}$;
- $fvar(\underline{c}) \stackrel{\text{def}}{=} \varnothing$;
- $fvar(M \ op \ N) \stackrel{\text{def}}{=} fvar(M) \cup fvar(N)$;
- $fvar(\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ L) \stackrel{\text{def}}{=} fvar(M) \cup fvar(N) \cup fvar(L)$;
- $fvar(\lambda x.M) \stackrel{\text{def}}{=} fvar(M) \setminus \{\, x \,\}$; occurrences of x in M are captured by the scope of λx, and hence are not free;
- $fvar(M\, N) \stackrel{\text{def}}{=} fvar(M) \cup fvar(N)$;
- $fvar(\mathsf{rec}\, x.M) \stackrel{\text{def}}{=} fvar(M) \setminus \{\, x \,\}$; occurrences of x in M are captured by the scope of $\mathsf{rec}\ x$, and hence are not free;
- $fvar(\mathsf{nil}) \stackrel{\text{def}}{=} \varnothing$;
- $fvar(\mathsf{hd}(M)) \stackrel{\text{def}}{=} fvar(M)$;
- $fvar(\mathsf{tl}(M)) \stackrel{\text{def}}{=} fvar(M)$;
- $fvar(M : N) \stackrel{\text{def}}{=} fvar(M) \cup fvar(N)$; and
- $fvar(\mathsf{elist}(M)) \stackrel{\text{def}}{=} fvar(M)$

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

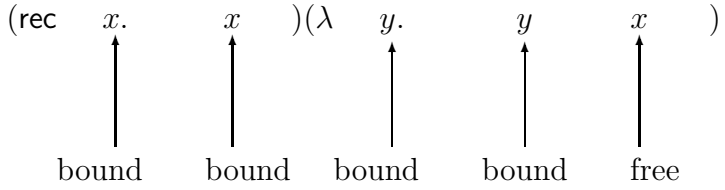
Examples 2.3.6 Warning: Note that a variable may occur both free and bound in a term. Here are two examples:

(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 $\mathsf{rec}\, z$.

(2)



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

2.4 Substitution of Terms

Motivation 2.4.1 Suppose that M and N are terms. If one thinks of M as a functional program, and the free occurrences of a variable x in M as places at which new code could be executed, we might consider replacing the variable x by N. Such a replacement is called a *substitution*. We substitute a term N for free occurrences of x in M simply by replacing each free x with N; this will produce a new term which will be denoted $M[N/x]$. For example, (if x then $\underline{4}$ else $\underline{5}$)[$\underline{1} = \underline{2}$ $/x$] denotes the term if $\underline{1} = \underline{2}$ then $\underline{4}$ else $\underline{5}$.

But things are not entirely straightforward! Suppose that $f \stackrel{\text{def}}{=} \lambda x.L$. Given any term N, the intended meaning of $f\,N$ is $L[N/x]$. Thus if L is y, then $f\,N = y[N/x] = y$. So if $M \stackrel{\text{def}}{=} \lambda x.y$, the intended meaning of M is "the function with constant value y". Now, y occurs freely in M, and x is a term, so we can try substituting the term x for the free occurrence of y, giving a new term denoted by $M[x/y]$. Now, $M[x/y]$ ought to be "the function with constant value x". But in fact $M[x/y]$ is clearly the term $\lambda x.x$, which is the identity function! The problem arises because when the variable x is substituted for the *free* variable y in $\lambda x.y$, x becomes captured by the scope of the abstraction λx.

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

Examples 2.4.2 Informal examples are

$$(\lambda x.x + y)[\underline{2}/y] = \lambda x.x + \underline{2} \qquad \text{and} \qquad (\lambda x.x + y)[x/y] = \lambda u.u + x.$$

In the second example, the substituted x will appear in the scope of λx, so we rename (to u) the bound x's to avoid capture.

Remark 2.4.3 We have now introduced a minor problem, which we shall deal with below. In the previous example, should $(\lambda x.x + y)[x/y]$ be $\lambda u.u + x$ or $\lambda z.z + x$ or ...? We can make a unique choice by appealing to the fixed enumeration (list) of the variables in *Var* (recall page 17). This is made clear in the following definition.

Definitions 2.4.4 We now give a formal definition of **substitution** of terms. In light of Discussion 2.4.1, we shall re-name any free variables which would become bound when the substitution takes place.

Given terms M and N, and a variable x, we shall define a new term denoted by $M[N/x]$, which is the term M with free occurrences of x replaced by N, *by recursion on the finite tree structure of* M:

- $x[N/x] \stackrel{\text{def}}{=} N$ (if $M \equiv x$);
- $y[N/x] \stackrel{\text{def}}{=} y$ where $x \neq y$ (if $M \equiv y$);
- $\underline{c}[N/x] \stackrel{\text{def}}{=} \underline{c}$ (if $M \equiv \underline{c}$);
- $(L\ op\ L')[N/x] \stackrel{\text{def}}{=} L[N/x]\ op\ L'[N/x]$ (if $M \equiv L\ op\ L'$ for some L and L');
- $(\mathsf{if}\ L\ \mathsf{then}\ L'\ \mathsf{else}\ L'')[N/x] \stackrel{\text{def}}{=} \mathsf{if}\ L[N/x]\ \mathsf{then}\ L'[N/x]\ \mathsf{else}\ L''[N/x]$ (if $M \equiv \ldots$ etc etc);
- $(L\,L')[N/x] \stackrel{\text{def}}{=} L[N/x]\,L'[N/x]$;
- $(\lambda x.L)[N/x] \stackrel{\text{def}}{=} \lambda x.L$; and
- $(\lambda y.L)[N/x] \stackrel{\text{def}}{=} \lambda y.L[N/x]$ if $x \neq y$ and $x \notin \mathit{fvar}(L)$ or $y \notin \mathit{fvar}(N)$;
- $(\lambda y.L)[N/x] \stackrel{\text{def}}{=} \lambda z.L[z/y][N/x]$ if $x \neq y$ and $x \in \mathit{fvar}(L)$ and $y \in \mathit{fvar}(N)$, where z is chosen as the first variable in (the fixed enumeration of) Var for which $z \notin \mathit{var}(N) \cup \mathit{var}(L)$. So occurrences of y in $\lambda y.L$ will be renamed to the variable z to ensure that occurrences of y in N will not be captured upon substitution;
- $(\mathsf{rec}\,x.L)[N/x] \stackrel{\text{def}}{=} \mathsf{rec}\,x.L$; and
- $(\mathsf{rec}\,y.L)[N/x] \stackrel{\text{def}}{=} \mathsf{rec}\,y.L[N/x]$ if $x \neq y$ and $x \notin \mathit{fvar}(L)$ or $y \notin \mathit{fvar}(N)$;
- $(\mathsf{rec}\,y.L)[N/x] \stackrel{\text{def}}{=} \mathsf{rec}\,z.L[z/y][N/x]$ if $x \neq y$ and $x \in \mathit{fvar}(L)$ and $y \in \mathit{fvar}(N)$, where z is chosen as the first variable in (the fixed enumeration of) Var for which $z \notin \mathit{var}(N) \cup \mathit{var}(L)$. So occurrences of y in $\mathsf{rec}\,y.L$ will be renamed to the variable z to ensure that occurrences of y in N will not be captured upon substitution;
- $\mathsf{nil}[N/x] \stackrel{\text{def}}{=} \mathsf{nil}$;
- $\mathsf{hd}(L)[N/x] \stackrel{\text{def}}{=} \mathsf{hd}(L[N/x])$;
- $\mathsf{tl}(L)[N/x] \stackrel{\text{def}}{=} \mathsf{tl}(L[N/x])$;
- $(L : L')[N/x] \stackrel{\text{def}}{=} L[N/x] : L'[N/x]$;
- $\mathsf{elist}(L)[N/x] \stackrel{\text{def}}{=} \mathsf{elist}(L[N/x])$.

Examples 2.4.5

(1)

$$\begin{array}{rcl}((v_1 + \underline{2}) : (v_3 v_2))[\underline{10}/v_2] & = & (v_1 + \underline{2})[\underline{10}/v_2] : (v_3 v_2)[\underline{10}/v_2] \\ & = & (v_1[\underline{10}/v_2] + \underline{2}[\underline{10}/v_2]) : (v_3[\underline{10}/v_2] v_2[\underline{10}/v_2]) \\ & = & (v_1 + \underline{2}) : (v_3\ \underline{10})\end{array}$$

Note that in the first example, we wrote down each of the recursive steps. It's not too difficult (!) to write the result of the substitution straight down, or at least miss out some of the steps, as in the next example:
(2)

$$\begin{array}{rcl}(\mathsf{rec}\, v_3.v_6\, v_3 : \mathsf{nil})[v_3\, v_1/v_6] & =_* & \mathsf{rec}\, v_2.(v_6\, v_3 : \mathsf{nil})[v_2/v_3][v_3\, v_1/v_6] \\ & = & \mathsf{rec}\, v_2.(v_6\, v_2 : \mathsf{nil})[v_3\, v_1/v_6] \\ & = & \mathsf{rec}\, v_2.((v_3\, v_1)\, v_2 : \mathsf{nil})\end{array}$$

where at * note that $v_6 \in \mathit{fvar}(v_6\, v_3 : \mathsf{nil})$ and $v_3 \in \mathit{fvar}(v_3\, v_1)$, so we have to rename v_3 to avoid capture. We rename v_3 to be the first variable in Var not appearing in

$$\mathit{fvar}(v_6\, v_3 : \mathsf{nil}) \cup \mathit{fvar}(v_3\, v_1) = \{\, v_1, v_3, v_6 \,\}$$

which is v_2.

Motivation 2.4.6 We have claimed that for any two terms M and N, and variable x, there is a term $M[N/x]$ which is specified by the previous definition. We should, of course, prove that $M[N/x]$ *is* a term. While this can be done, the proof is a subtle induction, and we omit it.

2.5 α-Equivalence

Motivation 2.5.1 We have seen that the two terms $\lambda x.y$ and $\lambda z.y$ have the same intended meaning, namely that they both represent the function with constant value y. You will also note the the definition of substitution is a little unwieldy due to the clauses which involve a renaming of bound variables. Whenever a renaming takes place we have to choose "the first variable in the enumeration $v_0, v_1, \ldots$ which does not appear in the terms involved in the substitution". Now, if we were to implement substitution, we would have to be explicit about *what* we renamed variables to, when avoiding capture. But, in fact, as regards the overall meaning of $\mathbb{UL}$ terms, it does not really matter what we rename variables to, *provided we choose a fresh variable.* Thus the computational meaning of both $\lambda u.u + x$ and $\lambda z.z + x$ in Remark 2.4.3 is the same—they are both functions which add x.

For these reasons, we shall *regard terms which differ only in the names of their bound variables as equivalent.* We have to give a proper definition of what it means for two terms to be "equal" if they "differ only in the names of their bound variables".

Definitions 2.5.2 To do this we shall define an equivalence[3] relation, denoted by $\sim_\alpha$, on the set $\mathit{Term}^{\mathbb{UL}}$ of terms. So formally $\sim_\alpha$ is a set (of pairs), and in particular a subset of $\mathit{Term}^{\mathbb{UL}} \times \mathit{Term}^{\mathbb{UL}}$. We define it inductively by the rules[4] in Figure 2.2.

[3] See Definitions 1.3.8.
[4] We shall write $M \sim_\alpha M'$ instead of (M, M') `in` $\sim_\alpha$.

$$\frac{}{M \sim_\alpha M}\ \text{REF} \qquad \frac{M \sim_\alpha M'}{M' \sim_\alpha M}\ \text{SYM} \qquad \frac{M \sim_\alpha M' \quad M' \sim_\alpha M''}{M \sim_\alpha M''}\ \text{TRAN}$$

$$\frac{M \sim_\alpha M' \quad N \sim_\alpha N'}{M\ op\ N \sim_\alpha M'\ op\ N'} \qquad \frac{M \sim_\alpha M' \quad N \sim_\alpha N' \quad L \sim_\alpha L'}{\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ L \sim_\alpha \mathsf{if}\ M'\ \mathsf{then}\ N'\ \mathsf{else}\ L'}$$

$$\frac{}{\lambda v.M \sim_\alpha \lambda v'.M[v'/v]}\ (1) \qquad \frac{M \sim_\alpha M'}{\lambda x.M \sim_\alpha \lambda x.M'} \qquad \frac{M \sim_\alpha M' \quad N \sim_\alpha N'}{M\,N \sim_\alpha M'\,N'}$$

$$\frac{}{\mathsf{rec}\,v.M \sim_\alpha \mathsf{rec}\,v'.M[v'/v]}\ (2) \qquad \frac{M \sim_\alpha M'}{\mathsf{rec}\,x.M \sim_\alpha \mathsf{rec}\,x.M'}$$

$$\frac{M \sim_\alpha M'}{\mathsf{hd}(M) \sim_\alpha \mathsf{hd}(M')} \qquad \frac{M \sim_\alpha M'}{\mathsf{tl}(M) \sim_\alpha \mathsf{tl}(M')} \qquad \frac{M \sim_\alpha M' \quad N \sim_\alpha N'}{M : N \sim_\alpha M' : N'}$$

$$\frac{M \sim_\alpha M'}{\mathsf{elist}(M) \sim_\alpha \mathsf{elist}(M')}$$

In (1) *and* (2), v' *may be any variable different from* v *and which does not occur in* M

Figure 2.2: Rules for Generating the α-Equivalence Relation $M \sim_\alpha M'$

The formal definition of two terms differing only in their bound variables is of course that the terms are α-equivalent. We wish to consider a term as being "equal" to all other α-equivalent terms, and we can do this by considering α-equivalence classes.

We define the set $Exp^{\mathbb{UL}}$ of **expressions** to be the set of α-equivalence classes of terms:

$$Exp^{\mathbb{UL}} \quad \stackrel{\text{def}}{=} \quad Term^{\mathbb{UL}}/\sim_\alpha = \{\ \overline{M}\ \mid\ M \in Term^{\mathbb{UL}}\ \}.^5$$

Example 2.5.3 We have

$$\overline{\lambda u.u + x} = \{\ M\ \mid\ \lambda u.u + x \sim_\alpha M\ \} = \{\ \lambda u.u + x, \lambda z.z + x, \dots\} = \overline{\lambda z.z + x} = \dots$$

Check this!! Rule (1) gives us (for example) $\lambda u.u + x \sim_\alpha \lambda z.z + x$ taking M to be $u + x$, v to be u and v' to be z.

⋙ **Warning 2.5.4** ***The formal definition of α-equivalence amounts to saying that two terms are α-equivalent if one can be transformed to the other by a sequence of changes of bound variables. The definition in Figure 2.2 makes this intuitive idea watertight. Instead of writing $\overline{M}$ for an expression, we adopt the convention that we simply write M, that is we shall denote an α-equivalence class by a representative. We shall "treat" expressions as though they are terms, but whenever we give a definition involving expressions, we must not forget that expressions are in fact α-equivalence classes and that we have to check that the definition is well-defined.***

If $M, N \in Term^{\mathbb{UL}}$ ***and*** $M \sim_\alpha N$***, then of course*** $\overline{M} = \overline{N}$***. For example*** $\lambda x.x \sim_\alpha \lambda z.z$ ***and so*** $\overline{\lambda x.x} = \overline{\lambda z.z}$***. Following the above convention, we can simply write*** $\lambda x.x = \lambda z.z$***. And magically, the convention also allows us to write***

$$(\lambda x.x + y)[x/y] = \lambda u.u + x = \lambda z.z + x.$$

Motivation 2.5.5 Finally, what about substitution of expressions? What expression is $M[N/x]$ when M and N are expresions, rather than terms? In practice, we can just "forget" that M and N are α-equivalence classes, and take the "expression" $M[N/x]$ to be the equivalence class of the term $M'[N'/x]$ where $M \sim_\alpha M'$ and $N \sim_\alpha N'$. Thus, it turns out that because we are dealing with α-equivalence classes, when we rename variables to avoid capture, we can choose any new name we like. And this avoids the hassle of a specific choice of variable, as we had on page 26. It is actually quite tricky to prove that this all works out, and we omit to do this. We look at one example:

Example 2.5.6 Dealing with α-equivalence classes we have

$$(\lambda x.(x + y))\,[\mathsf{rec}\ z.xz/y] = \lambda u.(u + \mathsf{rec}\ z.xz).$$

[5] Recall that $\overline{M} = \{\ M'\ \mid\ M \sim_\alpha M', M' \in Term^{\mathbb{UL}}\ \}$. See Definitions 1.3.8.

But (for example)

$$\lambda x.x + y = \lambda w.w + y \quad \text{and} \quad \mathsf{rec}\, z.xz = \mathsf{rec}\, v.xv$$

and so we ought to have

$$\lambda u.u + \mathsf{rec}\, z.xz = \lambda w.w + \mathsf{rec}\, v.xv. \qquad (*)$$

It is "easy to see" that $(*)$ holds via a renaming of bound variables. Here is how we could give a formal derivation:

$$\dfrac{\dfrac{\dfrac{}{u \sim_\alpha u}\,\text{REF} \quad \dfrac{}{\mathsf{rec}\, z.xz \sim_\alpha \mathsf{rec}\, v.xv}\,(2)}{u + \mathsf{rec}\, z.xz \sim_\alpha u + \mathsf{rec}\, v.xv}}{\lambda u.(u + \mathsf{rec}\, z.xz) \sim_\alpha \lambda u.(u + \mathsf{rec}\, v.xv)} \quad \dfrac{}{\lambda u.(u + \mathsf{rec}\, z.xz) \sim_\alpha \lambda w.(w + \mathsf{rec}\, v.xv)}\,(1)$$
$$\lambda u.(u + \mathsf{rec}\, z.xz) \sim_\alpha \lambda w.(w + \mathsf{rec}\, v.xv)\,\text{TRAN}$$

2.6 Terms with Contexts for $\mathbb{UL}$

Motivation 2.6.1 We will shortly use the concept of expressions to give an abstract definition of a (functional) program. Before we do this, we need one further technical device. It is very convenient, when dealing with expressions, to keep track of the free variables appearing in an expression. We will do this by defining judgements of the form $\Gamma \vdash M$ where Γ is a set of variables, M is a term, and the free variables of M all appear in Γ. An example is

$$\underbrace{\{\, x, y, z \,\}}_{\text{set of variables}} \quad \vdash \quad \underbrace{x + y}_{\text{term}}$$

For clarity, we usually drop the curly braces from the set of variables, writing this example as $x, y, z \vdash x + y$.

Definitions 2.6.2 We shall define a relation $\vdash$ between finite sets of variables and terms. More formally, $\vdash$ is a relation[6] between $\mathcal{P}_{fin}(\mathit{Var})$ and $\mathit{Term}^{\mathbb{UL}}$. We often write Γ for a typical element of $\mathcal{P}_{fin}(\mathit{Var})$, and it will be convenient to write Γ, x for $\Gamma \cup \{\, x \,\}$ and Γ, Γ' for $\Gamma \cup \Gamma'$. We define $\vdash$ inductively by the rules in Figure 2.3, where instead of writing (Γ, M) `in` $\vdash$, we use the more readable $\Gamma \vdash M$.

We define the set of **terms whose free variables appear in a context** Γ, denoted by $\mathit{Term}^{\mathbb{UL}}(\Gamma)$, and the set of **expressions whose free variables appear in** Γ, denoted by $\mathit{Exp}^{\mathbb{UL}}(\Gamma)$, by

$$\mathit{Term}^{\mathbb{UL}}(\Gamma) \stackrel{\text{def}}{=} \{\, M \mid \Gamma \vdash M \,\} \qquad \text{and} \qquad \mathit{Exp}^{\mathbb{UL}}(\Gamma) \stackrel{\text{def}}{=} \mathit{Term}^{\mathbb{UL}}(\Gamma)/\sim_\alpha$$

[6] So $\vdash$ is a set of pairs, each pair being of the form (Γ, M).

$$\frac{}{\Gamma \vdash x}\,[x \in \Gamma] \qquad \frac{}{\Gamma \vdash \underline{c}} \qquad \frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash M \; op \; N}$$

$$\frac{\Gamma \vdash M \quad \Gamma \vdash N \quad \Gamma \vdash L}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ L} \qquad \frac{\Gamma, x \vdash M}{\Gamma \vdash \lambda x.M} \qquad \frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash M\,N} \qquad \frac{\Gamma, x \vdash M}{\Gamma \vdash \mathsf{rec}\,x.M}$$

$$\frac{}{\Gamma \vdash \mathsf{nil}} \qquad \frac{\Gamma \vdash M}{\Gamma \vdash \mathsf{hd}(M)} \qquad \frac{\Gamma \vdash M}{\Gamma \vdash \mathsf{tl}(M)} \qquad \frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash M : N} \qquad \frac{\Gamma \vdash M}{\Gamma \vdash \mathsf{elist}(M)}$$

Figure 2.3: Rules for Generating the Relation $\Gamma \vdash M$

where you should note that the equivalence relation $\sim_\alpha$ on $Term^{\mathbb{UL}}$ induces an equivalence relation (also written $\sim_\alpha$) on $Term^{\mathbb{UL}}(\Gamma)$. Note that we write $\vdash M$ when Γ is empty, that is, $\varnothing \vdash M$. If $\vdash M$ we say that M is **closed**.

Proposition 2.6.3 If $\Gamma \vdash M$, then $fvar(M) \subseteq \Gamma$.

Proof We use Rule Induction for the inductively defined set $\vdash$. We wish to show that $\forall(\Gamma, M) \in \vdash.\ Prop((\Gamma, M))$ where $Prop((\Gamma, M)) \stackrel{\text{def}}{=} fvar(M) \subseteq \Gamma$. Thus all we need to do is verify property closure of the rules in Figure 2.3. Let us give one example.

(*Closure under the rule*):

$$\frac{\Gamma, x \vdash M}{\Gamma \vdash \lambda x.M}$$

The inductive hypothesis is $Prop((\Gamma \cup \{\, x \,\}, M))$, that is $fvar(M) \subseteq \Gamma \cup \{\, x \,\}$. We have to prove that $Prop((\Gamma, \lambda x.M))$, that is $fvar(\lambda x.M) \subseteq \Gamma$. We calculate

$$\begin{array}{lcll} fvar(\lambda x.M) & \stackrel{\text{def}}{=} & fvar(M) \setminus \{\, x \,\} & \\ & \subseteq & (\Gamma \cup \{\, x \,\}) \setminus \{\, x \,\} & \text{using the inductive hypothesis} \\ & = & \Gamma. & \end{array}$$

The rest of the proof, checking property closure for the other rules, is left as an exercise. □

2.7 Programs and Values for $\mathbb{UL}$

Motivation 2.7.1 A *program* will be a closed expression. A program is closed so that it is a "self contained" expression, into which no further data need be input. A program is required to be an expression, so that programs which differ only in their bound variables are equal.

We shall soon give rules which tell us how a program can be "evaluated" or "computed" to a value. A value will be a program that is as "fully evaluated as possible" according to a particular kind of evaluation or computation strategy. For example, $(\lambda x.x + \underline{2})\underline{3}$ is a program which computes to the value $\underline{5}$, and we can write this as

$$(\lambda x.x + \underline{2})\underline{3} \quad \Downarrow \quad \underline{5}$$

reading $\Downarrow$ as "evaluates to". Note that $\underline{5}$ is a value, but it is also a very trivial program—it *is* an expression with no free variables!! Functions, that is programs of the form $\lambda x.M$, will also be regarded as values. The idea is that the body M of the function will not be evaluated until an argument has been passed to the function. Finally, lists of the form $P : Q$, where P and Q are programs, are also values. This may seem odd at first sight—think of some examples. As we shall soon see in more detail, $\mathbb{UL}$ is a *lazy* language, meaning that "program fragments are only evaluated if they are used". Thus the head or tail of a list will only be evaluated *if* "extracted" by a hd or tl function. So $(\underline{3} + \underline{4}) : \mathsf{nil}$ is a value; it does not evaluate to $\underline{7} : \mathsf{nil}$.

Definitions 2.7.2 We define the set $Prog^{\mathbb{UL}}$ of **programs** to be the set of closed expressions, that is, we put

$$Prog^{\mathbb{UL}} \stackrel{\text{def}}{=} Exp^{\mathbb{UL}}(\varnothing).$$

We shall often denote a program, that is an element of $Prog^{\mathbb{UL}}$, by P or Q, though not exclusively. A **value** is a program which is *represented*[7] by a term given by the grammar

$$V ::= \underline{c} \mid \lambda x.M \mid \mathsf{nil} \mid P : Q$$

where M is any element of $Term^{\mathbb{UL}}(x)$ (that is $x \vdash M$) and P and Q are (representatives for) programs. We shall write $Val^{\mathbb{UL}}$ for the set of values.

2.8 An Evaluation Relation for $\mathbb{UL}$

Definitions 2.8.1 We shall define an **evaluation relation** between programs (elements of $Prog^{\mathbb{UL}}$) and values (elements of $Val^{\mathbb{UL}}$), which will take the form $P \Downarrow V$. So, formally, $\Downarrow \subseteq Prog^{\mathbb{UL}} \times Val^{\mathbb{UL}}$. It is defined by the rules[8] in Figure 2.4. Note that in rule AP, $\lambda x.M$ and Q must be programs, and so

$$M \in Exp^{\mathbb{UL}}(\varnothing, x) \text{ and } Q \in Exp^{\mathbb{UL}}(\varnothing).$$

It can be proved that $M[Q/x] \in Exp^{\mathbb{UL}}(\varnothing)$, that is, $M[Q/x]$ is indeed a program. Hence rule AP makes sense. A similar argument applies to rule REC.

Motivation 2.8.2 We refer to the definition of $\Downarrow$ as a **structured operational semantics** for $\mathbb{UL}$. The word *semantics* refers to the fact that the rules defining $\Downarrow$ give

[7] Up to α-equivalence. See Definitions 1.3.8 and 2.5.2.

[8] In the figure, we write $P \Downarrow V$ instead of (P, V) `in` $\Downarrow$.

$$\frac{}{V \Downarrow V}\ \text{VAL} \qquad \frac{P \Downarrow \underline{m} \quad Q \Downarrow \underline{n}}{P\ op\ Q \Downarrow \underline{m\ op\ n}}\ \text{OP}$$

$$\frac{P \Downarrow \underline{T} \quad Q \Downarrow V}{\mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \Downarrow V}\ \text{COND}_1 \qquad \frac{P \Downarrow \underline{F} \quad Q' \Downarrow V}{\mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \Downarrow V}\ \text{COND}_2$$

$$\frac{P \Downarrow \lambda x.M \quad M[Q/x] \Downarrow V}{P\,Q \Downarrow V}\ \text{AP} \qquad \frac{M[\mathsf{rec}\,x.M/x] \Downarrow V}{\mathsf{rec}\,x.M \Downarrow V}\ \text{REC}$$

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

$$\frac{P \Downarrow \mathsf{nil}}{\mathsf{elist}(P) \Downarrow \underline{T}}\ \text{ELIST}_1 \qquad \frac{P \Downarrow P' : Q}{\mathsf{elist}(P) \Downarrow \underline{F}}\ \text{ELIST}_2$$

Figure 2.4: Rules for Generating the Evaluation Relation $P \Downarrow V$ of $\mathbb{UL}$

a "meaning" to programs P. This "meaning" arises by showing how programs compute to values, which is specified in a "computational" or "operational" manner—hence the adjective *operational*. Finally, *structured* refers to the finite tree structure of P: whenever we have $P \Downarrow V$, we can see which rules might have been used to deduce $P \Downarrow V$ by looking at the outermost constructor of P. See Warning 2.8.3.

⋙ **Warning 2.8.3** ***We re-emphasise the comments above about structured semantics. Suppose that*** $P \Downarrow V$***. This judgement must be derived using the rules in Figure 2.4—we know this, because the set*** $\Downarrow$ ***is inductively defined, and its elements such as*** (P, V) ***only arise by application of the rules—see Warning 1.4.8. But the program*** P ***has a unique syntactic term structure—in particular, a unique outermost term constructor—so that*** P ***is either a constant (*** $P = \underline{c}$***), an operator term (*** $P = M\ op\ N$***), a conditional, and so on. This tells us which rule(s) could be used to deduce*** $P \Downarrow V$***. For example, if we know that*** $\mathsf{hd}(P) \Downarrow V$***, this must have been deduced through the rule*** HD***, and hence there must exist programs*** P' ***and*** Q ***for which*** $P \Downarrow P' : Q$ ***and*** $P' \Downarrow V$***.***

Motivation 2.8.4 You should note that the rules in Figure 2.4 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

$$P \equiv C(M_1, M_2, \ldots, M_n)$$

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

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

The same idea applies to lists. Consider $H : T$, that is $\mathsf{cons}(H,T)$ where cons is the program constructor. We regard this as a fully evaluated program—very lazy!! We only compute the subterms H or T 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 $\mathsf{cons}(P',Q)$, but then we only bother (lazy) to evaluate P' to a value, say V. Thus $\mathsf{hd}(P)$ evaluates to V, and there is no need to evaluate Q. Now look at rules HD and TL.

We have seen that if P is a program, V is a value, and $P \Downarrow V$, the latter means that "the program P evaluates to the value V". But what would happen if there was another value V' for which $P \Downarrow V'$? This would mean that one program could compute to two different values. In fact, thankfully, this cannot happen! The relation $\Downarrow$ is *deterministic*, meaning that a program can only compute to one value. We prove this formally by showing that for each of the rules used to generate $\Downarrow$, if the programs in the hypothesis of the rule are deterministic, so is the program in the conclusion of the rule.

Theorem 2.8.5 The relation $\Downarrow$ is **deterministic**: For any program P and values V and V', if $P \Downarrow V$ and $P \Downarrow V'$, then[9] $V = V'$.

Proof We shall show that $\forall P \Downarrow V.\ \mathit{Prop}((P,V))$[10] where

$$\mathit{Prop}((P,V)) \stackrel{\text{def}}{=} \forall V'\ (P \Downarrow V' \Longrightarrow V = V').$$

To do this we apply rule induction: we have to verify property closure for the rules in Figure 2.4.

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

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

H2 for all V', if $Q' \Downarrow V'$ then $V = V'$.

We have to prove that

C for any V', if $\quad \mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \Downarrow V' \quad$ then $V = V'$.

[9] Do not forget that $=$ here denotes equality of the two α-equivalence classes represented by V and V'.
[10] Remember that this is shorthand for $\forall (P,V) \in \Downarrow.\ \mathit{Prop}((P,V))$

Pick an arbitrary V' for which $\mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \Downarrow V'$—(*). Now (*) could be deduced from an application of either $\textsc{cond}_1$ or $\textsc{cond}_2$. If it were the former, then $P \Downarrow \underline{T}$. So using **H1**, we would have $\underline{F} = \underline{T}$, a contradiction. Hence (*) must be a conclusion to an instance of $\textsc{cond}_2$, say

$$\frac{P \Downarrow \underline{F} \quad Q' \Downarrow V'}{\mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \Downarrow V'}$$

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

(*Closure under* REC): The inductive hypothesis is

H for all V', if $M[\mathsf{rec}\, x.M/x] \Downarrow V'$ then $V = V'$.

We have to prove that

C for any V', $\mathsf{rec}\, x.M \Downarrow V'$ implies $V = V'$.

Pick an arbitrary V' for which $\mathsf{rec}\, x.M \Downarrow V'$. This last relation *must* arise through the rule REC, and so $M[\mathsf{rec}\, x.M/x] \Downarrow V'$. That $V = V'$ then follows from **H**. □

Examples 2.8.6 Prove that $(\lambda z.z * \underline{2})\,\underline{3} \Downarrow \underline{6}$. 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 AP, hence we need to show that $\lambda z.z * \underline{2} \Downarrow \lambda x.M$ for some x and M, and that $M[\underline{3}/x] \Downarrow \underline{6}$. The first of these is easy, being an instance of VAL with $x \equiv z$ and $M \equiv z * \underline{2}$. The second, namely $\underline{3} * \underline{2} \Downarrow \underline{6}$, is also easy following from OP. Putting this altogether we get

$$\dfrac{\dfrac{}{\lambda z.z * \underline{2} \Downarrow \lambda z.z * \underline{2}}\,\textsc{val} \qquad \dfrac{\dfrac{}{\underline{3} \Downarrow \underline{3}}\,\textsc{val} \qquad \dfrac{}{\underline{2} \Downarrow \underline{2}}\,\textsc{val}}{(z * \underline{2})[\underline{3}/z] \equiv \underline{3} * \underline{2} \Downarrow \underline{6}}\,\textsc{op}}{(\lambda z.z * \underline{2})\,\underline{3} \Downarrow \underline{6}}\,\textsc{ap}$$

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

$$\dfrac{T \qquad \dfrac{\dfrac{}{\lambda x.x + \underline{2} \Downarrow \lambda x.x + \underline{2}}\,\textsc{val} \qquad \dfrac{\dfrac{}{\underline{3} \Downarrow \underline{3}}\,\textsc{val} \qquad \dfrac{}{\underline{2} \Downarrow \underline{2}}\,\textsc{val}}{(x + \underline{2})[\underline{3}/x] \equiv \underline{3} + \underline{2} \Downarrow \underline{5}}\,\textsc{op}}{(\lambda x.x + \underline{2})\,\underline{3} \Downarrow \underline{5}}\,\textsc{ap}}{\mathsf{hd}((\lambda x.x + \underline{2})\,\underline{3} : \mathsf{nil}) \Downarrow \underline{5}}\,\textsc{hd}$$

where T is the tree

$$\frac{}{(\lambda x.x + \underline{2})\,\underline{3} : \mathsf{nil} \Downarrow (\lambda x.x + \underline{2})\,\underline{3} : \mathsf{nil}}\,\textsc{val}$$

$$\frac{P \rightsquigarrow P'}{P\ op\ Q \rightsquigarrow P'\ op\ Q}\ \mathrm{OP}_1 \qquad \frac{Q \rightsquigarrow Q'}{\underline{n}\ op\ Q \rightsquigarrow \underline{n}\ op\ Q'}\ \mathrm{OP}_2 \qquad \frac{}{\underline{n}\ op\ \underline{m} \rightsquigarrow \underline{n\ op\ m}}\ \mathrm{OP}_3$$

$$\frac{P \rightsquigarrow P'}{\mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \rightsquigarrow \mathsf{if}\ P'\ \mathsf{then}\ Q\ \mathsf{else}\ Q'}\ \mathrm{COND}$$

$$\frac{}{\mathsf{if}\ \underline{T}\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightsquigarrow P}\ \mathrm{COND}_1 \qquad \frac{}{\mathsf{if}\ \underline{F}\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightsquigarrow Q}\ \mathrm{COND}_2$$

$$\frac{P \rightsquigarrow P'}{P\,Q \rightsquigarrow P'\,Q}\ \mathrm{AP}_1 \qquad \frac{}{(\lambda x.M)\,Q \rightsquigarrow M[Q/x]}\ \mathrm{AP}_2 \qquad \frac{}{\mathsf{rec}\,x.M \rightsquigarrow M[\mathsf{rec}\,x.M/x]}\ \mathrm{REC}$$

$$\frac{P \rightsquigarrow P'}{\mathsf{hd}(P) \rightsquigarrow \mathsf{hd}(P')}\ \mathrm{HD}_1 \qquad \frac{}{\mathsf{hd}(P : Q) \rightsquigarrow P}\ \mathrm{HD}_2$$

$$\frac{P \rightsquigarrow P'}{\mathsf{tl}(P) \rightsquigarrow \mathsf{tl}(P')}\ \mathrm{TL}_1 \qquad \frac{}{\mathsf{tl}(P : Q) \rightsquigarrow Q}\ \mathrm{TL}_2$$

$$\frac{P \rightsquigarrow P'}{\mathsf{elist}(P) \rightsquigarrow \mathsf{elist}(P')}\ \mathrm{ELIST}_1 \qquad \frac{}{\mathsf{elist}(\mathsf{nil}) \rightsquigarrow \underline{T}}\ \mathrm{ELIST}_2 \qquad \frac{}{\mathsf{elist}(P : Q) \rightsquigarrow \underline{F}}\ \mathrm{ELIST}_3$$

Figure 2.5: Rules for Generating the Transition Relation $P \rightsquigarrow Q$ in $\mathbb{UL}$

2.9 A Transition Relation for $\mathbb{UL}$

Motivation 2.9.1 Not all programs compute to values! Some programs P loop or *diverge*, by which we mean that there does not exist a value V for which $P \Downarrow V$. An example is $\mathsf{rec}\,x.x$. However, when a program P does compute to a value, how can we calculate that value? The rules for deriving the relation $P \Downarrow V$ do not lend themselves to direct calculation. To overcome this problem, we shall define a new relation between programs, written $P \rightsquigarrow Q$. The intuitive idea is that if P and Q are related by $\rightsquigarrow$, then P "computes in one step" to Q. For example,

$$(\lambda x.x + \underline{2})\underline{3} \rightsquigarrow \underline{3} + \underline{2} \qquad \text{and} \qquad \underline{3} + \underline{2} \rightsquigarrow \underline{5}.$$

Now we give the full definition:

Definitions 2.9.2 We shall define a **transition relation** between programs, that is a binary relation on $Prog^{\mathbb{UL}}$. It takes the form $P \rightsquigarrow Q$ and is inductively defined by the rules in Figure 2.5. If $P \rightsquigarrow Q$ we say that P **computes in one step** to Q.

Motivation 2.9.3 Note again that $\rightsquigarrow$ is **lazy**. In order to compute $P\,Q$, we have to (deterministically!) apply rule AP_1 until P reduces to a value $\lambda x.M$ and then apply rule AP_2 which substitutes the function argument Q straight into M without first evaluating Q to a value:

$$P\,Q \rightsquigarrow_{\text{AP}_1} P'\,Q \rightsquigarrow \ \ldots \ \rightsquigarrow_{\text{AP}_1} (\lambda x.M)\,Q \rightsquigarrow_{\text{AP}_2} M[Q/x]$$

Some programs cannot compute in one step to another program. An example is $\underline{2} + \underline{T}$ for which there is no program Q with

$$(\underline{2} + \underline{T}) \rightsquigarrow Q.$$

One can see this by inspecting the rules for generating $\rightsquigarrow$. We say that such programs are **terminal**. However, when a program P is not terminal, there is a *unique* program Q for which $P \rightsquigarrow Q$. Thus, the relation $\rightsquigarrow$ is, like $\Downarrow$, *deterministic*.

Theorem 2.9.4 The relation $\rightsquigarrow$ is **deterministic**: If P, Q and Q' are any programs, then if $P \rightsquigarrow Q$ and $P \rightsquigarrow Q'$ we have $Q = Q'$.

Proof We want to prove that $\forall P \rightsquigarrow Q.\ \mathit{Prop}((P, Q))$ where

$$\mathit{Prop}((P, Q)) \stackrel{\text{def}}{=} \forall Q'(P \rightsquigarrow Q' \Longrightarrow Q = Q').$$

So by rule induction, we show property closure for the rules in Figure 2.5. This routine exercise is left to the reader. □

2.10 Relating Evaluation and Transition Relations in $\mathbb{UL}$

Motivation 2.10.1 We need to find a connection between $\Downarrow$ and $\rightsquigarrow$. Consider

$$\begin{array}{lcl} \mathsf{hd}((\lambda x.x + \underline{2})\,\underline{3} : \mathsf{nil}) & \rightsquigarrow & (\lambda x.x + \underline{2})\,\underline{3} \\ & \rightsquigarrow & \underline{3} + \underline{2} \\ & \rightsquigarrow & \underline{5} \end{array}$$

and (see Examples 2.8.6)

$$\mathsf{hd}((\lambda x.x + \underline{2})\,\underline{3} : \mathsf{nil}) \ \Downarrow \ \underline{5}$$

It appears that a program will compute to a value if there is a sequence of one-step transitions from the program to the value. This suggests that $\Downarrow$ might be the transitive closure of $\rightsquigarrow$. In fact $\Downarrow$ is (more-or-less) the reflexive transitive closure $\rightsquigarrow^*$—reflexivity arises from the fact that for any value V, we have $V \Downarrow V$.

Theorem 2.10.2 For every program P and value V in $\mathbb{UL}$, we have

$$P \Downarrow V \Longleftrightarrow P \rightsquigarrow^* V.$$

Proof

($\Rightarrow$) We use Rule Induction for $\Downarrow$ to prove $\forall P \Downarrow V, P \rightsquigarrow^* V$. The details are an exercise.

($\Leftarrow$) We can show that

$$X \stackrel{\text{def}}{=} \{\ (P,Q)\ \mid\ \forall V.\ (Q \Downarrow V \Longrightarrow P \Downarrow V)\ \}$$

is closed under the rules in Figure 2.5 which define $\rightsquigarrow$.

(*Closure under* HD_1): Suppose $(P,P') \in X$—(*). We have to prove $(\mathsf{hd}(P), \mathsf{hd}(P')) \in X$, that is

$$\forall V.\ (\mathsf{hd}(P') \Downarrow V \Longrightarrow \mathsf{hd}(P) \Downarrow V) \qquad (\dagger)$$

Pick an arbitrary value V and suppose that $\mathsf{hd}(P') \Downarrow V$. Then from rule HD of Figure 2.4 we know that there must be programs Q and Q' for which $P' \Downarrow Q : Q'$—(**) and $Q \Downarrow V$. Now, $Q : Q'$ is a value in $\mathbb{UL}$, so using supposition (*), and (**), we have $P \Downarrow Q : Q'$. Hence

$$\frac{P \Downarrow Q : Q' \quad Q \Downarrow V}{\mathsf{hd}(P) \Downarrow V}$$

As V was arbitrary, ($\dagger$) holds.

We can show closure under the other rules similarly, and the details are omitted. Hence by **IL** for $\rightsquigarrow$, we have $\rightsquigarrow \subseteq X$, that is for any P and Q,

$$P \rightsquigarrow Q \Longrightarrow \forall V\ (Q \Downarrow V \Longrightarrow P \Downarrow V).$$

Note that X is in fact a reflexive and transitive relation between programs—exercise: check this!! But, by definition, $\rightsquigarrow^*$ is the smallest such relation which contains $\rightsquigarrow$. Hence $\rightsquigarrow^* \subseteq X$, and so for any P and Q,

$$P \rightsquigarrow^* Q \Longrightarrow \forall V\ (Q \Downarrow V \Longrightarrow P \Downarrow V).$$

If we take $Q \stackrel{\text{def}}{=} V$, and note that $V \Downarrow V$, then we have

$$P \rightsquigarrow^* V \Longrightarrow P \Downarrow V$$

as required. □

Definitions 2.10.3 Suppose that P is a program. Either P is terminal or it is not; let us consider a non-terminal P. We say that P has a **finite transition sequence** if there is a transition sequence of the form

$$P \rightsquigarrow P_1 \rightsquigarrow P_2 \rightsquigarrow \ldots \rightsquigarrow P_m$$

for some natural number $m \geq 1$ for which P_m is terminal. In such a case we say that P is **convergent**, and call

$$P \rightsquigarrow P_1 \rightsquigarrow P_2 \rightsquigarrow \ldots \rightsquigarrow P_m$$

the **full transition sequence** of P.

If such m does not exist, we say that P has an **infinite transition sequence**, which must be of the form

$$P \rightsquigarrow P_1 \rightsquigarrow P_2 \rightsquigarrow \ldots \rightsquigarrow P_n \rightsquigarrow \ldots$$

where each P_n is non-terminal. We say that P is **divergent** or **loops**.

It is easy to see from the definition of $\rightsquigarrow$ that a value V is terminal. Note also that appealing to Theorem 2.9.4, any transition sequence must be a *unique*.

Example 2.10.4 Let $M \stackrel{\text{def}}{=} \mathsf{if}\ x = \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x + f\,(x - \underline{1})$, $F \stackrel{\text{def}}{=} \lambda x.M$ and $R \stackrel{\text{def}}{=} \mathsf{rec}\, f.F$. We give the full transition sequence of $R\,\underline{2}$ in $\mathbb{UL}$.

$$\begin{array}{rcl}
R\,\underline{2} & \rightsquigarrow & F[R/f]\,\underline{2} \ \equiv\ (\lambda x.M[R/f])\,\underline{2} \\
& \rightsquigarrow & M[R/f][\underline{2}/x] \ \equiv\ \mathsf{if}\ \underline{2} = \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2} + R\,(\underline{2} - \underline{1}) \\
& \rightsquigarrow & \mathsf{if}\ \underline{F}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2} + R\,(\underline{2} - \underline{1}) \\
& \rightsquigarrow & \underline{2} + R\,(\underline{2} - \underline{1}) \\
& \rightsquigarrow_{(1)} & \underline{2} + (\lambda x.M[R/f])\,(\underline{2} - \underline{1}) \\
& \rightsquigarrow & \underline{2} + M[R/f][\underline{2} - \underline{1}/x] \\
& \equiv & \underline{2} + (\mathsf{if}\ (\underline{2} - \underline{1}) = \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ (\underline{2} - \underline{1}) + R\,((\underline{2} - \underline{1}) - \underline{1})) \\
& \rightsquigarrow & \underline{2} + (\mathsf{if}\ \underline{1} = \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ (\underline{2} - \underline{1}) + R\,((\underline{2} - \underline{1}) - \underline{1})) \\
& \rightsquigarrow & \underline{2} + (\mathsf{if}\ \underline{T}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ (\underline{2} - \underline{1}) + R\,((\underline{2} - \underline{1}) - \underline{1})) \\
& \rightsquigarrow & \underline{2} + \underline{1} \\
& \rightsquigarrow & \underline{3}
\end{array}$$

It is not too difficult to verify each of the transition steps. For example, the step $\rightsquigarrow_{(1)}$ is valid because:

$$\dfrac{\dfrac{\dfrac{}{R \rightsquigarrow F[R/f]}\,\text{REC}}{R\,(\underline{2} - \underline{1}) \rightsquigarrow (\lambda x.M[R/f])\,(\underline{2} - \underline{1})}\,\text{AP}_1}{\underline{2} + R\,(\underline{2} - \underline{1}) \rightsquigarrow_{(1)} \underline{2} + (\lambda x.M[R/f])\,(\underline{2} - \underline{1})}\,\text{OP}_2$$

where of course $F[R/f] = \lambda x.M[R/f]$.

2.11 The Syntax, Programs and Values of $\mathbb{UE}$

Motivation 2.11.1 We have seen that the language $\mathbb{UL}$ has lazy functions and lists, in that subterms of such programs are only evaluated if the values of the subterms are explicitly required in later computations. The language $\mathbb{UE}$ has the same syntax as $\mathbb{UL}$ in which one may write programs, but a different operational semantics, that is, a new definition of $\Downarrow$ and $\rightsquigarrow$. The operational semantics of $\mathbb{UE}$ is *eager* which, roughly, means that subterms of programs are always computed to values before the program itself is evaluated.

Consider $P\,Q$. Let us write this as the finite tree $\mathsf{ap}(P,Q)$ (as originally defined) where ap is the program constructor. In order to evaluate $\mathsf{ap}(P,Q)$ eagerly, we *must* evaluate P to a function, say $\lambda x.M$, *and* Q to a value, say V'. The next step of the computation is to evaluate $M[V'/x]$. If now $M[V'/x]$ evaluates to a value, say V, then so too does $\mathsf{ap}(P,Q)$. Now look at rule AP, and see how it captures our intended operational semantics!

Look at the operational rules for lists and notice that they capture eager evaluation.

Definitions 2.11.2 In order to specify $\mathbb{UE}$, we have to define the sets of **terms**, **expressions**, **terms whose free variables appear in a context** Γ, **expressions whose free variables appear in a context** Γ, and **programs**. These are all exactly the same as for $\mathbb{UL}$:

$$\begin{array}{rcl} \mathit{Term}^{\mathbb{UE}} & \stackrel{\text{def}}{=} & \mathit{Term}^{\mathbb{UL}} \\ \mathit{Exp}^{\mathbb{UE}} & \stackrel{\text{def}}{=} & \mathit{Exp}^{\mathbb{UL}} \\ \mathit{Term}^{\mathbb{UE}}(\Gamma) & \stackrel{\text{def}}{=} & \mathit{Term}^{\mathbb{UL}}(\Gamma) \\ \mathit{Exp}^{\mathbb{UE}}(\Gamma) & \stackrel{\text{def}}{=} & \mathit{Exp}^{\mathbb{UL}}(\Gamma) \\ \mathit{Prog}^{\mathbb{UE}} & \stackrel{\text{def}}{=} & \mathit{Prog}^{\mathbb{UL}} \end{array}$$

The values of $\mathbb{UE}$ are different. The set of values $\mathit{Val}^{\mathbb{UE}}$ in $\mathbb{UE}$ consists of those programs in $\mathbb{UE}$ which are *represented*[11] by terms which appear in the grammar

$$V ::= \underline{c} \mid \lambda x.M \mid \mathsf{nil} \mid V : V'.$$

The crucial difference of $\mathbb{UE}$ to $\mathbb{UL}$ is that lists consist of values, rather than programs.

2.12 Evaluation and Transition Relations for $\mathbb{UE}$

Motivation 2.12.1 We specify a structured operational semantics for $\mathbb{UE}$ just as we did for $\mathbb{UL}$. The difference is that the rules enforce an *eager* computation strategy. The evaluation relation $\Downarrow$ is **deterministic** which means that (up to α-equivalence) a (non looping) program must evaluate to a unique value. So Theorem 2.8.5 holds for $\mathbb{UE}$. We omit the proof, but you should be aware of the statement of the theorem.

Definitions 2.12.2 The evaluation relation $\Downarrow$ for $\mathbb{UE}$ is generated by modifying the rules in Figure 2.4. The new rules are given in Figure 2.6.

Motivation 2.12.3 We shall give a transition relation for $\mathbb{UE}$. The motivation is the same as that for $\mathbb{UL}$. The difference here is that the computation strategy is *eager*. For example, in $\mathbb{UL}$ we have

$$(\lambda x.x * \underline{2})(\underline{3} + \underline{10}) \rightsquigarrow (\underline{3} + \underline{10}) * \underline{2}$$

but in $\mathbb{UE}$ we have to compute the argument first

$$(\lambda x.x * \underline{2})(\underline{3} + \underline{10}) \rightsquigarrow (\lambda x.x * \underline{2})\underline{13} \rightsquigarrow \underline{13} * \underline{2}.$$

[11] That is, representatives for α-equivalence—see Definitions 1.3.8 and Warning 2.5.4.

$$\frac{}{V \Downarrow V}\ \text{VAL} \qquad \frac{P \Downarrow \underline{m} \quad Q \Downarrow \underline{n}}{P\ op\ Q \Downarrow \underline{m\ op\ n}}\ \text{OP}$$

$$\frac{P \Downarrow \underline{T} \quad Q \Downarrow V}{\mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \Downarrow V}\ \text{COND}_1 \qquad \frac{P \Downarrow \underline{F} \quad Q' \Downarrow V}{\mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \Downarrow V}\ \text{COND}_2$$

$$\frac{P \Downarrow \lambda x.M \quad Q \Downarrow V' \quad M[V'/x] \Downarrow V}{P\,Q \Downarrow V}\ \text{AP} \qquad \frac{M[\mathsf{rec}\,x.M/x] \Downarrow V}{\mathsf{rec}\,x.M \Downarrow V}\ \text{REC}$$

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

$$\frac{P \Downarrow V \quad Q \Downarrow V'}{P : Q \Downarrow V : V'}\ \text{CONS} \qquad \frac{P \Downarrow \mathsf{nil}}{\mathsf{elist}(P) \Downarrow \underline{T}}\ \text{ELIST}_1 \qquad \frac{P \Downarrow V : V'}{\mathsf{elist}(P) \Downarrow \underline{F}}\ \text{ELIST}_2$$

Figure 2.6: Rules for Generating the Evaluation Relation $P \Downarrow V$ of $\mathbb{UE}$

Just as for $\mathbb{UL}$, the transition relation is deterministic. So Theorem 2.9.4 holds for $\mathbb{UE}$. Again, we omit the proof, but you should be aware of the theorem.

Definitions 2.12.4 We define a **transition relation** between programs, that is a binary relation on $Prog^{\mathbb{UE}}$. The transition relation $\rightsquigarrow$ for $\mathbb{UE}$ is generated by modifying the rules in Figure 2.5. The new rules are given in Figure 2.7. If $P \rightsquigarrow Q$ we say that P **computes in one step** to Q.

Motivation 2.12.5 The connection between $\rightsquigarrow$ and $\Downarrow$ is the same for $\mathbb{UE}$ as for $\mathbb{UL}$. Roughly, a program can compute in a finite number of transition steps to a value iff it evaluates to the value. This is made precise in the following theorem, whose routine proof is omitted:

Theorem 2.12.6 For every program P and value V in $\mathbb{UE}$, we have

$$P \Downarrow V \iff P \rightsquigarrow^* V.$$

2.13 Further Examples and Comments

Examples 2.13.1

(1) We have seen that a number of programs will loop in $\mathbb{UE}$ but converge to a value in $\mathbb{UL}$. One reason that eager languages *are* used is that they are easier to implement than their lazy counterparts. We do not go into the details here.

$$\frac{P \rightsquigarrow P'}{P\ op\ Q \rightsquigarrow P'\ op\ Q}\ \mathrm{OP}_1 \qquad \frac{Q \rightsquigarrow Q'}{\underline{n}\ op\ Q \rightsquigarrow \underline{n}\ op\ Q'}\ \mathrm{OP}_2 \qquad \frac{}{\underline{n}\ op\ \underline{m} \rightsquigarrow \underline{n\ op\ m}}\ \mathrm{OP}_3$$

$$\frac{P \rightsquigarrow P'}{\mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \rightsquigarrow \mathsf{if}\ P'\ \mathsf{then}\ Q\ \mathsf{else}\ Q'}\ \mathrm{COND}$$

$$\frac{}{\mathsf{if}\ \underline{T}\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightsquigarrow P}\ \mathrm{COND}_1 \qquad \frac{}{\mathsf{if}\ \underline{F}\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightsquigarrow Q}\ \mathrm{COND}_2$$

$$\frac{P \rightsquigarrow P'}{P\,Q \rightsquigarrow P'\,Q}\ \mathrm{AP}_1 \qquad \frac{Q \rightsquigarrow Q'}{(\lambda x.M)\,Q \rightsquigarrow (\lambda x.M)\,Q'}\ \mathrm{AP}_2 \qquad \frac{}{(\lambda x.M)\,V \rightsquigarrow M[V/x]}\ \mathrm{AP}_3$$

$$\frac{}{\mathsf{rec}\,x.M \rightsquigarrow M[\mathsf{rec}\,x.M/x]}\ \mathrm{REC}$$

$$\frac{P \rightsquigarrow P'}{\mathsf{hd}(P) \rightsquigarrow \mathsf{hd}(P')}\ \mathrm{HD}_1 \qquad \frac{}{\mathsf{hd}(V : V') \rightsquigarrow V}\ \mathrm{HD}_2$$

$$\frac{P \rightsquigarrow P'}{\mathsf{tl}(P) \rightsquigarrow \mathsf{tl}(P')}\ \mathrm{TL}_1 \qquad \frac{}{\mathsf{tl}(V : V') \rightsquigarrow V'}\ \mathrm{TL}_2$$

$$\frac{P \rightsquigarrow P'}{P : Q \rightsquigarrow P' : Q}\ \mathrm{CONS}_1 \qquad \frac{Q \rightsquigarrow Q'}{V : Q \rightsquigarrow V : Q'}\ \mathrm{CONS}_2$$

$$\frac{P \rightsquigarrow P'}{\mathsf{elist}(P) \rightsquigarrow \mathsf{elist}(P')}\ \mathrm{ELIST}_1 \qquad \frac{}{\mathsf{elist}(\mathsf{nil}) \rightsquigarrow \underline{T}}\ \mathrm{ELIST}_2 \qquad \frac{}{\mathsf{elist}(V : V') \rightsquigarrow \underline{F}}\ \mathrm{ELIST}_3$$

Figure 2.7: Rules for Generating the Transition Relation $P \rightsquigarrow Q$ in $\mathbb{UE}$

(2) Let $M \stackrel{\text{def}}{=} \mathsf{if}\ x = \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x + f\,(x - \underline{1})$, $F \stackrel{\text{def}}{=} \lambda x.M$ and $R \stackrel{\text{def}}{=} \mathsf{rec}\, f.F$. We give the full transition sequence of $R\,\underline{2}$ in $\mathbb{UE}$. Compare Example 2.10.4.

$$\begin{array}{rcl}
R\,\underline{2} & \rightsquigarrow & F[R/f]\,\underline{2} \ \equiv\ (\lambda x.M[R/f])\,\underline{2} \\
& \rightsquigarrow & M[R/f][\underline{2}/x] \ \equiv\ \mathsf{if}\ \underline{2} = \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2} + R\,(\underline{2} - \underline{1}) \\
& \rightsquigarrow & \mathsf{if}\ \underline{F}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2} + R\,(\underline{2} - \underline{1}) \\
& \rightsquigarrow & \underline{2} + R\,(\underline{2} - \underline{1}) \\
& \rightsquigarrow & \underline{2} + (\lambda x.M[R/f])\,(\underline{2} - \underline{1}) \\
& \rightsquigarrow_{(*)} & \underline{2} + (\lambda x.M[R/f])\,\underline{1} \\
& \rightsquigarrow & \underline{2} + M[R/f][\underline{1}/x] \ \equiv\ \underline{2} + (\mathsf{if}\ \underline{1} = \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{1} + R\,(\underline{1} - \underline{1})) \\
& \rightsquigarrow & \underline{2} + (\mathsf{if}\ \underline{T}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{1} + R\,(\underline{1} - \underline{1})) \\
& \rightsquigarrow & \underline{2} + \underline{1} \\
& \rightsquigarrow & \underline{3}
\end{array}$$

Note that in step $(*)$ the argument $\underline{2} - \underline{1}$ is evaluated eagerly, that is, evaluated before being passed to $\lambda x.M[R/f]$.

(3) We write a program P in $\mathbb{UL}$ which supplies an infinite list of the positive odd numbers. Clearly $\mathsf{hd}(P)$ should compute to $\underline{1}$, $\mathsf{hd}(\mathsf{tl}(P))$ to $\underline{3}$ and so on. Let

$$R \stackrel{\text{def}}{=} \mathsf{rec}\, f.\lambda n.n : f\,(n + \underline{2}) \text{ and } P \stackrel{\text{def}}{=} R\,\underline{1}.$$

Then

$$\begin{array}{rcl}
\mathsf{hd}(P) & \rightsquigarrow & \mathsf{hd}((\lambda n.n : R\,(n + \underline{2}))\,\underline{1}) \\
& \rightsquigarrow & \mathsf{hd}(\underline{1} : R\,(\underline{1} + \underline{2})) \\
& \rightsquigarrow & \underline{1}
\end{array}$$

and

$$\begin{array}{rcl}
\mathsf{hd}(\mathsf{tl}(P)) & \rightsquigarrow^* & \mathsf{hd}(\mathsf{tl}(\underline{1} : R\,(\underline{1} + \underline{2}))) \\
& \rightsquigarrow & \mathsf{hd}(R\,(\underline{1} + \underline{2})) \\
& \rightsquigarrow^* & \mathsf{hd}((\underline{1} + \underline{2}) : R\,((\underline{1} + \underline{2}) + \underline{2})) \\
& \rightsquigarrow & \underline{1} + \underline{2} \\
& \rightsquigarrow & \underline{3}
\end{array}$$

However in $\mathbb{UE}$ we would have

$$\begin{array}{rcl}
\mathsf{hd}(P) & \rightsquigarrow & \mathsf{hd}((\lambda n.n : R\,(n + \underline{2}))\,\underline{1}) \\
& \rightsquigarrow & \mathsf{hd}(\underline{1} : R\,(\underline{1} + \underline{2})) \\
& \rightsquigarrow & \mathsf{hd}(\underline{1} : R\,\underline{3}) \\
& \rightsquigarrow^* & \mathsf{hd}(\underline{1} : \underline{3} : R\,\underline{5}) \\
& \rightsquigarrow^* & \mathsf{hd}(\underline{1} : \underline{3} : \underline{5} : \underline{7} : R\,\underline{9}) \\
& \rightsquigarrow & \ldots
\end{array}$$

and $\mathsf{hd}(P)$ diverges (or loops)—the list has to be evaluated to a list of values before the head is taken.

(4) This example uses the programs of example (3). We shall prove that

$$\boxed{\forall k \geq 1 \in \mathbb{N}. \quad \mathsf{hd}(\mathsf{tl}^k(P)) \Downarrow \underline{2k+1}}$$

where $\mathsf{tl}^k(\xi)$ means k applications of the tail constructor to ξ.

To do this, write $S_m \stackrel{\text{def}}{=} \underline{1} + \underbrace{\underline{2} + \ldots \underline{2}}_{m \text{ times}}$. We first prove that

$$\forall k \in \mathbb{N}. \quad \underbrace{\mathsf{tl}^k(P) \Downarrow S_k : R\,(S_k + \underline{2})}_{Prop(k)} \qquad (*)$$

by Mathematical Induction. (Note that the idea for S_m comes from example (3)).

(*Case* $Prop(1)$): This is $\mathsf{tl}(P) \Downarrow (\underline{1} + \underline{2}) : R\,((\underline{1} + \underline{2}) + \underline{2})$ which follows from example (3).

(*Case* $\forall k, Prop(k)$ *implies* $Prop(k+1)$): We assume the result holds for an arbitrary k and show that it must hold at $k+1$. It is quite easy to derive the following tree:

$$\dfrac{\dfrac{}{R \Downarrow \lambda n.n : R\,(n + \underline{2})}\,\text{REC} \quad \downarrow \quad \dfrac{\dfrac{}{(n : R\,(n + \underline{2}))[(S_k + \underline{2})/n] \;\Downarrow\; (S_k + \underline{2}) : R\,((S_k + \underline{2}) + \underline{2})}\,\text{VAL}}{R\,(S_k + \underline{2}) \;\Downarrow\; (S_k + \underline{2}) : R\,((S_k + \underline{2}) + \underline{2})}\,\text{AP}}{\mathsf{tl}(\mathsf{tl}^k(P)) \;\Downarrow\; (S_k + \underline{2}) : R\,((S_k + \underline{2}) + \underline{2})}\,\text{TL}$$

where $IH \stackrel{\text{def}}{=} \mathsf{tl}^k(P) \;\Downarrow\; S_k : R\,(S_k + \underline{2})$ is the inductive hypothesis $Prop(k)$, and so we have shown that

$$\mathsf{tl}^{k+1}(P) \;\Downarrow\; S_{k+1} : R\,(S_{k+1} + \underline{2})$$

as required. Hence by induction, $(*)$ holds. Therefore, for *any* $k \in \mathbb{N}$ we have

$$\dfrac{\mathsf{tl}^k(P) \;\Downarrow\; S_k : R\,(S_k + \underline{2}) \qquad S_k \;\Downarrow\; \underline{2k+1}}{\mathsf{hd}(\mathsf{tl}^k(P)) \;\Downarrow\; \underline{2k+1}}$$

where it is easy to see that $S_k \;\Downarrow\; \underline{2k+1}$, and so we are done.

3
The SECD Machine

3.1 Why Introduce the SECD Machine?

Motivation 3.1.1 We have seen how to define a transition relation $\rightsquigarrow$ for the language $\mathbb{UE}$. Given a program P, it is fairly easy for a human to give the full 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 the deduction tree

$$\dfrac{\dfrac{}{\underline{3}+\underline{2} \rightsquigarrow \underline{5}}\,\mathrm{OP}_3}{(\underline{3}+\underline{2}) \leq \underline{6} \rightsquigarrow \underline{5} \leq \underline{6}}\,\mathrm{OP}_1 \qquad (*)$$

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

$$\overset{\text{deduction tree}}{P \quad \rightsquigarrow \quad P'}$$

An evaluation semantics, $\Downarrow$, is very much an opposite to the notion of a rewrite relation $\mapsto$. To show that $P \Downarrow V$ requires a "large" proof search for a deduction tree, and completely suppresses any notion of "mechanistic evaluation" of P to V. However, $\Downarrow$ is more useful for proving general properties of programs—cf (4) of Examples 2.13.1. We illustrate these ideas in Figure 3.1.

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

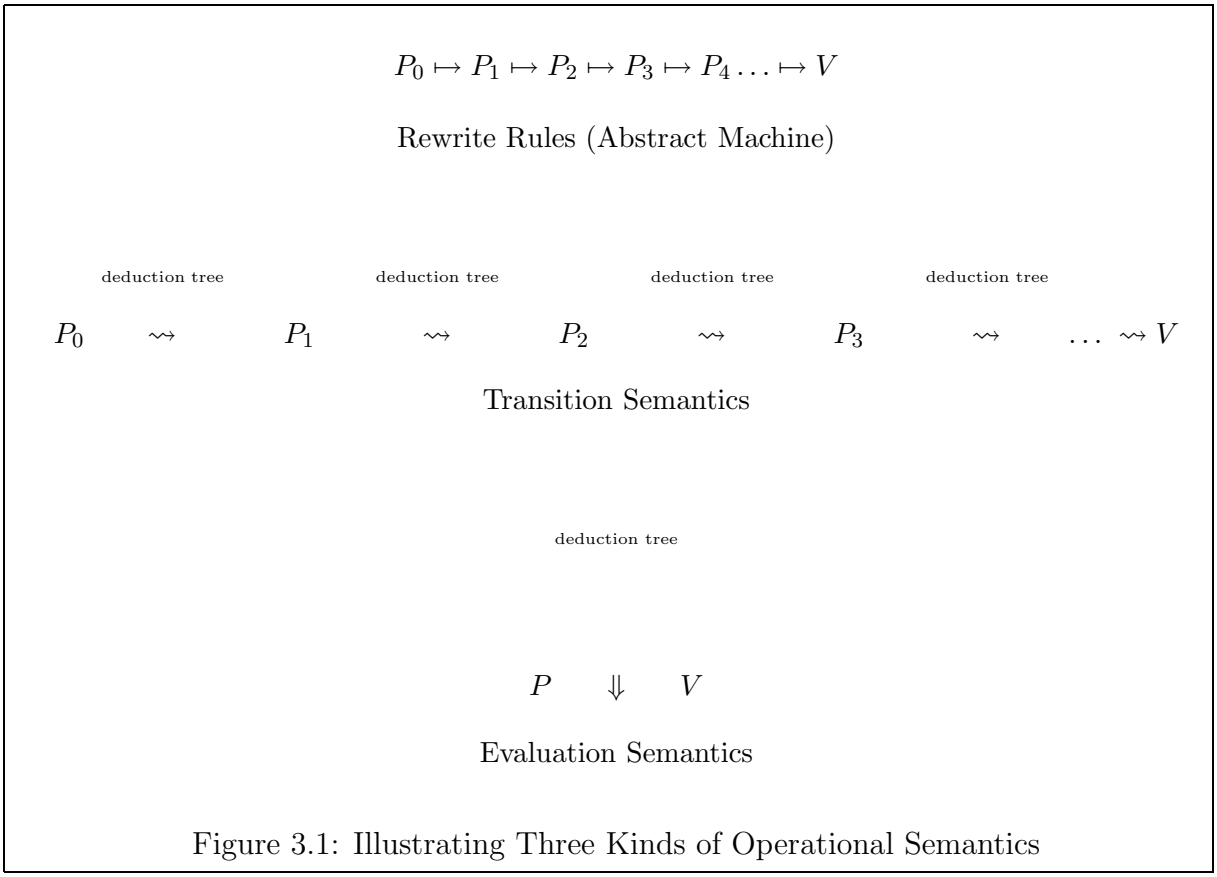


Figure 3.1: Illustrating Three Kinds of Operational Semantics

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 $\mathbb{UE}$. The terms of this language fragment are given by the grammar

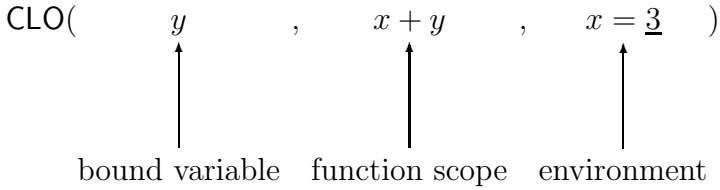
$$M ::= x \mid \underline{c} \mid M\ M \mid \lambda x.M \mid M\ op\ M$$

and the definition of $Prog^{\mathbb{UE}}$ is just as in Chapter 2 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 $\mathbb{UE}$.

3.2 The Definition of the SECD Machine

Motivation 3.2.1 Before we outline the structure of the SECD machine, we introduce the notion of a closure. Consider $(\lambda x.\lambda y.(x+y))\,\underline{3}\,\underline{5} \rightsquigarrow (\lambda y.(\underline{3}+y))\,\underline{5}$. The transition involves the substitution of $\underline{3}$ for the free variable x in $\lambda y.(x+y)$. The SECD machine implements substitution via an environment which records the values of variables. The SECD machine represents $(\lambda x.\lambda y.(x+y))\,\underline{3}$, that is $(\lambda y.(x+y))[\underline{3}/x]$, as a *closure*, which

is a triple consisting of the bound variable, the scope, and the current 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 state (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 state (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 3.2.2 Let us write SECD machine states as arrays:

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

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

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

If the control is non-empty, then its first command triggers a state rewrite, whereby the SECD machine changes to a new state. 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

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

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

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

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	$M\ op\ N\ ;\ C$
D	D

$\overset{\text{optm}}{\longmapsto}$

S	S
E	E
C	$N\ ;\ M\ ;\ op\ ;\ C$
D	D

An operator *op* is computed:

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

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

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	$\lambda x.M\ ;\ C$
D	D

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

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	$M\ N\ ;\ C$
D	D

$\overset{\text{fapp}}{\longmapsto}$

S	S
E	E
C	$N\ ;\ M\ ;\ \mathsf{APP}\ ;\ C$
D	D

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

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

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

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

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

S	a	res	**S**	$a\ ;\ S$
E	E'	$\longmapsto$	**E**	E
C	$-$		**C**	C
D	(S, E, C, D)		**D**	D

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

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

3.3 Example Evaluations

Examples 3.3.1

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

S	$-$	fapp	**S**	$-$	cst	**S**	$\underline{4}$	abs			
E	$-$	$\longmapsto$	**E**	$-$	$\longmapsto$	**E**	$-$	$\longmapsto$			
C	$(\lambda y.y + \underline{2})\,\underline{4}$		**C**	$\underline{4}\ ;\ \lambda y.y + \underline{2}\ ;\ \mathsf{APP}$		**C**	$\lambda y.y + \underline{2}\ ;\ \mathsf{APP}$				
D	$-$		**D**	$-$		**D**	$-$				

S	$\mathsf{CLO}(y, y + \underline{2}, -)\ ;\ \underline{4}$	clo	**S**	$-$	optm	**S**	$-$	cst			
E	$-$	$\longmapsto$	**E**	$y = \underline{4}$	$\longmapsto$	**E**	$y = \underline{4}$	$\longmapsto$			
C	APP		**C**	$y + \underline{2}$		**C**	$\underline{2}\ ;\ y\ ;\ +$				
D	$-$		**D**	$(-, -, -, -)$		**D**	$(-, -, -, -)$				

S	$\underline{2}$	var	**S**	$\underline{4}\ ;\ \underline{2}$	op	**S**	$\underline{6}$	res	**S**	$\underline{6}$	
E	$y = \underline{4}$	$\longmapsto$	**E**	$y = \underline{4}$	$\longmapsto$	**E**	$y = \underline{4}$	$\longmapsto$	**E**	$-$	
C	$y\ ;\ +$		**C**	$+$		**C**	$-$		**C**	$-$	
D	$(-, -, -, -)$		**D**	$(-, -, -, -)$		**D**	$(-, -, -, -)$		**D**	$-$	

Hence $\lambda y.(y + \underline{2})\,\underline{4} \Downarrow \underline{6}$.

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

S	$-$	optm	**S**	$-$	cst	**S**	$\underline{5}$	fapp			
E	$-$	$\longmapsto$	**E**	$-$	$\longmapsto$	**E**	$-$	$\longmapsto$			
C	$(\lambda x.x)\,\underline{3} \leq \underline{5}$		**C**	$\underline{5}\ ;\ (\lambda x.x)\,\underline{3}\ ;\leq$		**C**	$(\lambda x.x)\,\underline{3}\ ;\leq$				
D	$-$		**D**	$-$		**D**	$-$				

S	$\underline{5}$
E	$-$
C	$\underline{3}$; $\lambda x.x$; APP ;$\leq$
D	$-$

$\xrightarrow{\text{cst}}$

S	$\underline{3}$; $\underline{5}$
E	$-$
C	$\lambda x.x$; APP ;$\leq$
D	$-$

$\xrightarrow{\text{abs}}$

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

$\xrightarrow{\text{clo}}$

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

$\xrightarrow{\text{var}}$

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

$\xrightarrow{\text{res}}$

S	$\underline{3}$; $\underline{5}$
E	$-$
C	$\leq$
D	$-$

$\xrightarrow{\text{op}}$

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

Hence $(\lambda x.x)\,\underline{3} \leq \underline{5} \Downarrow \underline{F}$.

4

A Typed Functional Language

4.1 Introduction

Motivation 4.1.1 Types are the names of collections of expressions (programs) of the same "kind" (type!). Such expressions will have the same use/behaviour. Types are used to eliminate certain kinds of *error* in code—recall your experiences with Haskell!! Examples of types are int, $(\mathsf{int} \to \mathsf{int}) \to \mathsf{bool}$, $(\mathsf{bool}, \mathsf{bool}) \to \mathsf{int}$ and so on.

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*. Haskell is an example of such a language. 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*.

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, recall that the function `until` of Haskell is polymorphic:

```
until ::  (a-> bool)->(a->a)->a->a
```

In this chapter we shall study type inference in a language called $\mathbb{ML}$. We shall be able to study some of the formal properties of types and type inference, without being cluttered by the full complexities of Haskell typing.

$\mathbb{ML}$ has so-called *implicit* polymorphism, and we discuss this later on. Two other 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.

4.2 The Types and Terms of $\mathbb{ML}$

Motivation 4.2.1 The types we see in $\mathbb{ML}$ are a subset of those in Haskell. This will make it easier to illustrate the ideas of type inference. We shall use capital letters to denote type variables. In this section we give the types of $\mathbb{ML}$ and define rules for assigning types to expressions.

Definitions 4.2.2 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 $\mathbb{ML}$ is inductively specified by the grammar which follows:

σ	::=	X	type variable
	|	int	type of Integers
	|	bool	type of Booleans
	|	$\sigma \to \sigma$	function type
	|	(σ, σ)	product type
	|	$[\sigma]$	type of lists of elements of type σ

and in general we denote types by the Greek letters σ, τ and ρ and their primed variants, σ', τ' and ρ'. We adopt the convention that $\sigma \to \sigma' \to \sigma''$ means $\sigma \to (\sigma' \to \sigma'')$, with the obvious extension to four or more types just as in Haskell.

Each type is a *finite syntax tree* (recall that the terms of Chapter 2 are also finite syntax trees). Of course two finite trees are syntactically equal iff their corresponding subtrees are syntactically equal. Thus it follows that (for example) $\sigma \to \tau \equiv \sigma' \to \tau'$ iff $\sigma \equiv \sigma'$ and $\tau \equiv \tau'$. Also, note that (for example) $\mathsf{int} \not\equiv [\sigma]$ for any type σ. Why?

Definitions 4.2.3 The set $Term^{\mathbb{ML}}$ of **terms** of $\mathbb{ML}$ is specified inductively by the following grammar:

M	::=	x	variable
	|	$\underline{c}$	constant
	|	$M\ op\ M$	arithmetic operator
	|	$\mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M$	conditional
	|	$\mathsf{let}\ x = M\ \mathsf{in}\ M$	local declaration
	|	$\lambda x.M$	function term
	|	$M\,M$	function application
	|	(M, M)	pairing
	|	$\mathsf{fst}(M)$	first projection
	|	$\mathsf{snd}(M)$	second projection
	|	nil	empty list
	|	$M : M$	cons
	|	$\mathsf{hd}(M)$	head of list
	|	$\mathsf{tl}(M)$	tail of list
	|	$\mathsf{elist}(M)$	test for empty list

We do not bother to give the full set of rules for inductively defining $Term^{\mathbb{ML}}$; by now, it should be clear to you what the rules would say. The definition of the **subterm**

binary relation $\lhd$ on $Term^{\mathbb{ML}}$ is defined as you expect, but we do give the clause for local declarations:

$$S \lhd \mathsf{let}\ x = M\ \mathsf{in}\ N \quad \Longleftrightarrow \quad S \equiv x \text{ or } S \lhd M \text{ or } S \lhd N.$$

Note that λx binds variables just as in the previous chapters. There is also variable binding in local declarations of the form $\mathsf{let}\ x = M\ \mathsf{in}\ N$. In fact, the occurrence of x just after the let, and occurrences of the variable x in N are bound. More precisely, an occurrence of a variable x in a term M of $\mathbb{ML}$ is **bound** if either

• the occurrence is in a subterm of M of the form $\lambda x.L$; or

• the occurrence of x is immediately to the right of a let, or in L' in a subterm of M of the form $\mathsf{let}\ x = L\ \mathsf{in}\ L'$.

The **scope** of x in $\mathsf{let}\ x = M\ \mathsf{in}\ N$ is N. An occurrence of x in M is **free** if it is not bound. We can give a recursive definition of the set of free variables of a term. The definition is the same as that on page 24, extended by the following clauses:

• $fvar(\mathsf{let}\ x = M\ \mathsf{in}\ N) \stackrel{\text{def}}{=} fvar(M) \cup (fvar(N) \setminus \{\, x \,\})$ $\quad \star\star\star$ NB!! $\star\star\star$

• $fvar((M, N)) \stackrel{\text{def}}{=} fvar(M) \cup fvar(N)$;

• $fvar(\mathsf{fst}(M)) \stackrel{\text{def}}{=} fvar(M)$;

• $fvar(\mathsf{snd}(M)) \stackrel{\text{def}}{=} fvar(M)$;

The reader can provide (exercise!!) definitions of $var(M)$ and $bvar(M)$.

Example 4.2.4 In $\mathsf{let}\ x = \underline{x} + y\ \mathsf{in}\ xyz$, the first and third occurrences of x are bound, but the second (underlined) is free.

Definitions 4.2.5 The formal definition of α-**equivalence** of terms is omitted; by now, the basic idea should be clear—see the examples below. The set $Exp^{\mathbb{ML}}$ of **expressions** of $\mathbb{ML}$ is defined to be the set of terms identified up to α-equivalence. As usual, we notationally confuse a term with the α-equivalence class it denotes. The definition of **substitution** of expressions for free occurrences of variables is the obvious one, changing variables to avoid capture.

Examples 4.2.6

(1) $x * \underline{3} \lhd (\lambda x.x * \underline{3})\, y$.

(2) $\lambda x.x + y \sim_\alpha \lambda u.u + y$.

(3) $\mathsf{nil}[\underline{2}/x] = \mathsf{nil}$.

(4) $(\lambda x.x \leq y)[x\, z/y] = \lambda u.u \leq x\, z$.

(5) $\mathsf{fst}(\underline{4}) \ \sim_\alpha \ \mathsf{fst}(\underline{4})$.

(6) $\mathsf{let}\ z = z\ \mathsf{in}\ z \ \sim_\alpha \ \mathsf{let}\ y = z\ \mathsf{in}\ y$.

(7) $\mathsf{let}\ x = \lambda y.x\ \mathsf{in}\ x + y \ \sim_\alpha \ \mathsf{let}\ u = \lambda y.x\ \mathsf{in}\ u + y$.

(8) $(x, y)[z/x] = (z, y)$.

(9) $(\mathsf{let}\ x = x + z\ \mathsf{in}\ xyz)[(x + \underline{2})/z] = \mathsf{let}\ u = x + (x + \underline{2})\ \mathsf{in}\ uy(x + \underline{2})$. Here, the occurrences of x which are bound in the local declaration are renamed to u.

4.3 Type Assignment in $\mathbb{ML}$

Motivation 4.3.1 In Chapter 2 we defined a relation $\Gamma \vdash M$ in which the free variables of M all appeared in the context Γ. Thus Γ is an *environment* of all declared variables. In the untyped setting, $\Gamma \vdash M$ did not play a very significant role—we used it to define the set of programs (expressions which are closed)—but we could simply have said that programs were expressions with no free variables without defining $\Gamma \vdash M$. However, in the typed setting, terms with contexts take on an important role. They allow us to write down rules from which only "sensible" or "well typed" terms can arise. In $\mathbb{UL}$, $\underline{2} + \underline{T}$ is a program. In $\mathbb{ML}$, programs are those (closed) expressions which have been assigned a type, and (as we shall see) $\underline{2} + \underline{T}$ is untypable in $\mathbb{ML}$.

In $\mathbb{ML}$, contexts Γ are sets of variables which "carry a type". The (binary) relation $x_1, \ldots, x_n \vdash M$ is replaced by a (ternary) relation

$$x_1 :: \sigma_1, \ldots, x_n :: \sigma_n \vdash M :: \sigma$$

which one reads as "in an environment in which the variables x_i are assigned the types σ_i, it follows that the expression M is assigned the type σ".

Definitions 4.3.2 Let Γ be a finite set of (variable, type) pairs $\{ (x_1, \sigma_1), \ldots, (x_n, \sigma_n) \}$ in which all the variables are *distinct*. We call such a finite set a (typed) **context**, and a typical such context will be written

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

where we write $x_i :: \sigma_i$ for the pair (x_i, σ_i). Note that Γ is a *set*, so the order of the pairs does not matter. We write *Cxt* for the set of all contexts, and $var(\Gamma)$ for the set of variables appearing in Γ. If Γ and Γ' are contexts, and the variables in Γ are distinct from those in Γ' then $\Gamma \cup \Gamma'$ is a context, and we denote it by Γ, Γ'. We also write $\Gamma \cup \{ x :: \sigma \}$ as $\Gamma, x :: \sigma$. We shall define a (ternary) relation $\vdash$ between (typed) contexts and terms and types. Formally, then, $\vdash$ is a subset

$$\vdash \subseteq (Cxt \times Term^{\mathbb{ML}} \times Type^{\mathbb{TL}})$$

and we shall write $\Gamma \vdash M{:}\sigma$ if $(\Gamma, M, \sigma) \in \vdash$. We define $\vdash$ inductively by the rules in Figure 4.1. Given a term $M \in Term^{\mathbb{ML}}$, if there exists a context Γ and some type σ for which $\Gamma \vdash M :: \sigma$, then we say that M is **typable**.

$$\frac{}{\Gamma \vdash x :: \sigma}\,[x :: \sigma \in \Gamma]\ \text{VAR} \qquad \frac{}{\Gamma \vdash \underline{n} :: \mathsf{int}}\ \text{INT}$$

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

$$\frac{\Gamma \vdash M :: \mathsf{int} \quad \Gamma \vdash N :: \mathsf{int}}{\Gamma \vdash M\ op\ N :: \mathsf{int}}\ \text{OP}_1 \qquad op \in \{+, -, *\}$$

$$\frac{\Gamma \vdash M :: \mathsf{int} \quad \Gamma \vdash N :: \mathsf{int}}{\Gamma \vdash M\ op\ N :: \mathsf{bool}}\ \text{OP}_2 \qquad op \in \{\leq, \geq, =\}$$

$$\frac{\Gamma \vdash B :: \mathsf{bool} \quad \Gamma \vdash N :: \sigma \quad \Gamma \vdash L :: \sigma}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ N\ \mathsf{else}\ L :: \sigma}\ \text{COND}$$

$$\frac{\Gamma \vdash M :: \sigma \quad \Gamma \vdash N[M/x] :: \sigma'}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N :: \sigma'}\ \text{LET}$$

$$\frac{\Gamma, x :: \sigma \vdash M :: \tau}{\Gamma \vdash \lambda x.M :: \sigma \to \tau}\ \text{ABS} \qquad \frac{\Gamma \vdash M :: \sigma \to \tau \quad \Gamma \vdash N :: \sigma}{\Gamma \vdash M\,N :: \tau}\ \text{AP}$$

$$\frac{\Gamma \vdash M :: \sigma \quad \Gamma \vdash N :: \sigma'}{\Gamma \vdash (M, N) :: (\sigma, \sigma')}\ \text{PAIR} \qquad \frac{\Gamma \vdash M :: (\sigma, \sigma')}{\Gamma \vdash \mathsf{fst}(M) :: \sigma}\ \text{FST} \qquad \frac{\Gamma \vdash M :: (\sigma, \sigma')}{\Gamma \vdash \mathsf{snd}(M) :: \sigma'}\ \text{SND}$$

$$\frac{}{\Gamma \vdash \mathsf{nil} :: [\sigma]}\ \text{NIL} \qquad \frac{\Gamma \vdash M :: \sigma \quad \Gamma \vdash N :: [\sigma]}{\Gamma \vdash M : N :: [\sigma]}\ \text{CONS}$$

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

Figure 4.1: Rules Generating the Type Assignments $\Gamma \vdash M :: \sigma$ in $\mathbb{ML}$

4.4 Type Assignment Examples

Motivation 4.4.1 Suppose that you are asked, given some Γ, M and σ, to prove that $\Gamma \vdash M :: \sigma$. So, you have to give a deduction of $\Gamma \vdash M :: \sigma$ using the rules in Figure 4.1. To figure out the deduction tree, suppose that the tree has the general form

$$\begin{array}{c} \text{deduction tree} \\ \hline \Gamma \vdash M :: \sigma \end{array} R$$

where R was the final rule used in the deduction. Which rule is R? To see this, look at the expression M. You will then see that *only one of the rules in Figure 4.1 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.

Examples 4.4.2

(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 $\Gamma \vdash \lambda x.\underline{0} : x :: [\mathsf{int}] \to [\mathsf{int}]$ 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 $M \equiv \underline{0} : x$, and $\sigma \equiv \tau \equiv [\mathsf{int}]$.

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

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

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 $M \equiv \underline{T}$ and $N \equiv \underline{2} : \mathsf{nil}$ 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 $\sigma \equiv \mathsf{int}$ and $\sigma \equiv \mathsf{bool}$ 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 $\mathbb{ML}$.

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 $\mathsf{int} \equiv [\sigma]$, a contradiction. So the expression cannot be typable.

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

To do this, we note that a unique rule from Figure 4.1 must be used to derive the typing assertion, and it has to be ABS. A careful inspection shows us that we have $\Gamma = \varnothing$, $M \equiv x : \mathsf{nil}$, $\sigma \equiv X$ and $\tau \equiv [X]$. 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:

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

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

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

(7) Show that $\vdash \lambda f.\lambda x.f\,(f\,x) :: (X \to X) \to X \to X$. Writing Γ for $f :: X \to X, x :: X$

we have

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

(8) Show that $(\lambda x.xy)\,y$ 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:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y :: \tau, x :: \sigma_2 \vdash x :: \sigma_3 \to \sigma_1}\,\text{VAR} \quad \dfrac{}{y :: \tau, x :: \sigma_2 \vdash y :: \sigma_3}\,\text{VAR}}{y :: \tau, x :: \sigma_2 \vdash xy :: \sigma_1}\,\text{AP}}{y :: \tau \vdash \lambda x.xy :: \sigma_2 \to \sigma_1}\,\text{ABS} \quad \dfrac{}{y :: \tau \vdash y :: \sigma_2}\,\text{VAR}}{y :: \tau \vdash (\lambda x.xy)\,y :: \sigma_1}\,\text{AP}$$

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

4.5 Formal Properties of Type Assignment

Motivation 4.5.1 It is easy to see that there are expressions of $\mathbb{ML}$ 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 $\sigma \equiv \mathsf{int}$. However, if a closed expression is typable, it usually has many types. For example, $\vdash \lambda x.x :: \sigma \to \sigma$ holds for any type σ. However, in $\mathbb{ML}$, 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 now make all this precise, and give a number of general results about $\mathbb{ML}$ type inference:

Definitions 4.5.2 Suppose that S: *TyVar* $\rightharpoonup$ *Type* is a partial function and that σ is an $\mathbb{ML}$ type. We define $S^*\sigma$ to be the type resulting from replacing each type variable X of σ with the type $S(X)$ whenever $S(X)$ is defined. We call S^* a **type substitution**. We write $S^*\Gamma$ for $\ldots, x_i :: S^*\sigma_i, \ldots$. If S is defined at just one type variable, say $S(Y) \stackrel{\text{def}}{=} \tau$, then we write $\sigma[\tau/Y]$ instead of $S^*\sigma$—an obvious notational convention. We say that σ **generalises** σ' if there exists a type substitution S^* for which $\sigma' = S^*\sigma$. We say that σ' is an **instance** of σ. Note that if $S(X)$ is defined, then it follows that $S^*X = S(X)$.

Examples 4.5.3

(1) Define S by setting $S(X) = U$, $S(Y) = \mathsf{bool}$ and S is otherwise undefined. Let $\sigma \stackrel{\text{def}}{=} (X, Y \to Z)$ and $\Gamma \stackrel{\text{def}}{=} x :: X, y :: Y \to Z$. Then

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

(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 is defined by $S(X) = [\mathsf{bool}]$ and $S(Z) = \mathsf{int}$.

Motivation 4.5.4 In $\mathbb{ML}$, if $\vdash M :: \sigma$, the type σ assigned to the (closed) expression M is **principal** if σ generalises any other type which can be assigned to M. The principal type of $\lambda x.x$ is $X \to X$. If an expression has a principal type, there is an algorithm (due to Hindley; Damas-Milner) which will compute it—but this is a story for another time. In Standard ML, program values are returned to the user with their principal types. Note that "principal type" is simply another phrase for "most general type".

It can be shown that if $\vdash M :: \sigma$ and X is a type variable, then for any type σ' we have

$$\Gamma \vdash M :: \sigma[\sigma'/X]. \qquad (*)$$

Note that (by definition!) the type σ generalises $\sigma[\sigma'/X]$. Thus if $\vdash \lambda x.x : \mathsf{nil} :: X \to [X]$, and σ is any type, then $(X \to [X])[\sigma/X] \equiv \sigma \to [\sigma]$, and so $\vdash \lambda x.x : \mathsf{nil} :: \sigma \to [\sigma]$ using $(*)$.

4.6 Local Polymorphism in $\mathbb{ML}$

Motivation 4.6.1 The LET rule permits different occurrences of x in N to have different **implicit** types in a local declaration $\mathsf{let}\ x = M\ \mathsf{in}\ N$. Thus, M can be used polymorphically in the body N. This idea is best explained by example.

Example 4.6.2 We first note that $\vdash \lambda x.x :: X \to X$:

$$\dfrac{\mathrm{DT}(X)\left\{\dfrac{}{x :: X \vdash x :: X}\,\text{VAR}\right.}{\vdash \lambda x.x :: X \to X}\,\text{ABS}$$

where we label the upper tree by $\mathrm{DT}(X)$. Note that replacing X by any type σ in $\mathrm{DT}(X)$ yields a deduction $\mathrm{DT}(\sigma)$ for $\vdash \lambda x.x :: \sigma \to \sigma$. Hence

$$\dfrac{\mathrm{DT}_1\left\{\dfrac{\dfrac{\mathrm{DT}(\mathsf{bool})}{\vdash \lambda x.x :: \mathsf{bool} \to \mathsf{bool}}\ (2) \qquad \dfrac{}{\vdash \underline{T} :: \mathsf{bool}}\,\text{TRUE}\right.}{\vdash (\lambda x.x)\,\underline{T} :: \mathsf{bool}}\,\text{AP}$$

and

$$\mathrm{DT}_2\left\{\begin{array}{c}\dfrac{\mathrm{DT}([X])}{\vdash \lambda x.x :: [X] \to [X]} \quad (3) \qquad \dfrac{}{\vdash \mathsf{nil} :: [X]}\ \text{NIL}\end{array}\right.$$
$$\frac{}{\vdash (\lambda x.x)\,\mathsf{nil} :: [X]}\ \text{AP}$$

Putting things together we get

$$\mathrm{DT}_3\left\{\begin{array}{c}\dfrac{\mathrm{DT}_1}{\vdash (\lambda x.x)\,\underline{T} :: \mathsf{bool}}\ \text{AP} \qquad \dfrac{\mathrm{DT}_2}{\vdash (\lambda x.x)\,\mathsf{nil} :: [X]}\ \text{AP}\end{array}\right.$$
$$\frac{}{\vdash ((\lambda x.x)\,\underline{T}, (\lambda x.x)\,\mathsf{nil}) :: (\mathsf{bool}, [X])}\ \text{PAIR}$$

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

$$\frac{\dfrac{\mathrm{DT}(Y)}{\vdash \lambda x.x :: Y \to Y}\ \text{ABS} \qquad \dfrac{\mathrm{DT}_3}{\vdash (f\,\underline{T}, f\,\mathsf{nil})[(\lambda x.x)/f] :: (\mathsf{bool}, [X])}}{\vdash \mathsf{let}\ f = (\lambda x.x)\ \mathsf{in}\ (f\,\underline{T}, f\,\mathsf{nil}) :: (\mathsf{bool}, [X])}\ \text{LET}$$

If we look at the above deduction of

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

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 $\lambda x.x$ is $Y \to Y$ and both of the implicit types of f are substitution instances of this most general type, with $S(Y) = \mathsf{bool}$ and $S(Y) = [X]$, respectively.

Motivation 4.6.3 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, $\mathbb{ML}$ has one other variable binding operation, that in function terms $\lambda x.M$. Can such bound variables have polymorphic instances within the scope of λx abstractions? The answer is in fact no. An example in the final section illustrates this.

4.7 Further Examples

Examples 4.7.1

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

The deduction tree must look like

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

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

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

To see this, we derive a possible deduction tree.

$$\dfrac{\dfrac{\dfrac{}{f :: \sigma_2 \vdash f :: \sigma_6 \to \sigma_4}\ \text{VAR} \qquad \dfrac{}{f :: \sigma_2 \vdash \underline{T} :: \sigma_6 \equiv \mathsf{bool}}\ \text{TRUE}}{f :: \sigma_2 \vdash f\,\underline{T} :: \sigma_4}\ \text{AP} \qquad \dfrac{\dfrac{}{f :: \sigma_2 \vdash f :: \sigma_7 \to \sigma_5}\ \text{VAR} \qquad \dfrac{}{f :: \sigma_2 \vdash \mathsf{nil} :: \sigma_7 \equiv [\sigma_8]}\ \text{NIL}}{f :: \sigma_2 \vdash f\,\mathsf{nil} :: \sigma_5}\ \text{AP}}{f :: \sigma_2 \vdash (f\,\underline{T}, f\,\mathsf{nil}) :: \sigma_3 \equiv (\sigma_4, \sigma_5)}$$

$$\dfrac{f :: \sigma_2 \vdash (f\,\underline{T}, f\,\mathsf{nil}) :: \sigma_3 \equiv (\sigma_4, \sigma_5)}{\vdash \lambda f.(f\,\underline{T}, f\,\mathsf{nil}) :: \sigma_1 \equiv \sigma_2 \to \sigma_3}\ \text{ABS}$$

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

Index