# A Structural Approach to Operational Semantics

G. D. Plotkin

University of Aarhus, Denmark

# Contents


**1 Transition System and Interpreting Automata** **3**

1.1 Introduction . . . 3
1.2 Transition Systems . . . 4
1.3 Examples of Transition Systems . . . 5
1.3.1 Finite Automata . . . 5
1.3.2 Three Counter Machines . . . 6
1.3.3 Context-Free Grammars . . . 8
1.4 Labelled Transition Systems . . . 9
1.5 Interpreting Automata . . . 13
1.5.1 The Language L . . . 13
1.5.2 The SMC-Machine . . . 15
1.6 Exercises . . . 18
1.7 Bibliography . . . 23

**2 Simple Expressions and Commands** **25**

2.1 Simple Expressions . . . 25
2.1.1 L-Expressions . . . 29
2.1.2 Boolean Expressions . . . 30
2.2 Simple Commands . . . 32
2.3 L-commands . . . 35
2.4 Structural Induction . . . 37
2.5 Dynamic Errors . . . 41
2.6 Simple Type-Checking . . . 42
2.7 Static Errors . . . 45
2.8 Exercises . . . 47
2.9 BIBLIOGRAPHICAL REMARKS . . . 51
2.10 Bibliography . . . 52

**3 Definitions and Declarations** **54**

3.1 Introduction . . . 54
3.2 Simple Definitions in Appllcative Languages . . . 54
3.3 Compound Definitions . . . 58
3.4 Type-Checking and Definitions . . . 64

3.5 Exercises . . . 79
3.6 Remarks . . . 84
3.7 Bibliography . . . 85

**4 Functions, Procedures and Classes** **87**
4.1 Functions in Applicative Languages . . . 88
4.2 Procedures and Functions . . . 100
4.3 Other Parameter Mechanisms . . . 106
4.4 Higher Types . . . 112
4.5 Modules and Classes . . . 115
4.6 Exercises . . . 122

**A A GUIDE TO THE NOTATION** **128**

**B NOTES ON SETS** **130**

# Chapter 1

# Transition System and Interpreting Automata

## 1.1 Introduction

It is the purpose of these notes to develop a simple and direct method for specifying the semantics of programming languages. Very little is required in the way of mathematical background; all that will be involved is "symbol-pushing" of one kind or another of the sort which will already be familiar to readers with experience of either the non-numerical aspects of programming languages or else formal deductive systems of the kind employed in mathematical logic.

Apart form a simple kind of mathematics the method is intended to produce concise comprehensible semantic definitions. Indeed the method is even intended as a direct formalisation of (many aspects of) the usual informal natural language descriptions. I should really confess here that while I have some experience what has been expressed above is rather a pious hope than a statement of fact. I would therefore be most grateful to readers for their comments and particularly their criticisms.

I will follow the approach to programming languages taken by such authors a Gordon [Gor] and Tennent [Ten] considering the main syntactic classes – expressions, commands and declarations – and the various features found in each. The linguistic approach is that developed by the Scott-Strachey school (together with Landin and McCarthy and others) but within an operational rather than a denotational framework. These notes should be considered as an attempt at showing the feasibility of such an approach. Apart from various inadequacies of the treatment as presented many topics of importance are omitted. These include data structures and data types; various forms of control structure from jumps to exceptions and coroutines; concurrency including semaphores, monitors and communicating process.

Many thanks are due to the Department of Computer Science at Aarhus University at whose invitation I was enabled to spend a very pleasant six months developing this material. These notes partially cover a series of lectures given at the department. I would like also to thank the staff and students whose advice and criticism had a strong influence and also Jette Milwertz whose typing skills made the work look better than it should.

## 1.2 Transition Systems

The announced "symbol-pushing" nature of our method suggests what is the truth; it is an *operational* method of specifying semantics based on *syntactic* transformations of programs and *simple* operations on discrete data. The idea is that in general one should be interested in computer *systems* whether hardware or software and for semantics one thinks of systems whose *configurations* are a mixture of syntactical objects – the programs and data – such as stores or environment. Thus in these notes we have

$$\text{SYSTEM} = \text{PROGRAM} + \text{DATA}$$

One wonders if this study could be generalized to other kinds of systems, especially hardware ones.

Clearly systems have some behavior and it is that which we wish to describe. In an operational semantics one focuses on the operations the system can perform – whether internally or interactively with some suppersystem or the outside world. For in our discrete (digital) computer systems behavior consists of elementary steps which are occurrences of operations. Such elementary steps are called here, (and also in many other situations in Computer Science) *transitions* (= *moves*). Thus a transitions steps from one configuration to another and as a first idea we take it to be a binary relation configurations.

Definition 1.2.1: A *Transition System* (ts) is (just!) a structure $\langle \Gamma, \longrightarrow \rangle$ where $\Gamma$ is a set (of elements, $\gamma$, called *configurations*) and $\longrightarrow \subseteq \Gamma \times \Gamma$ is a binary relation (called the *transition relation*). Read $\gamma \longrightarrow \gamma'$ as saying that there is a transition from the configuration $\gamma$ to the configuration $\gamma'$. (Other notations sometimes seen are $\vdash$, $\Rightarrow$ and $\rhd$).

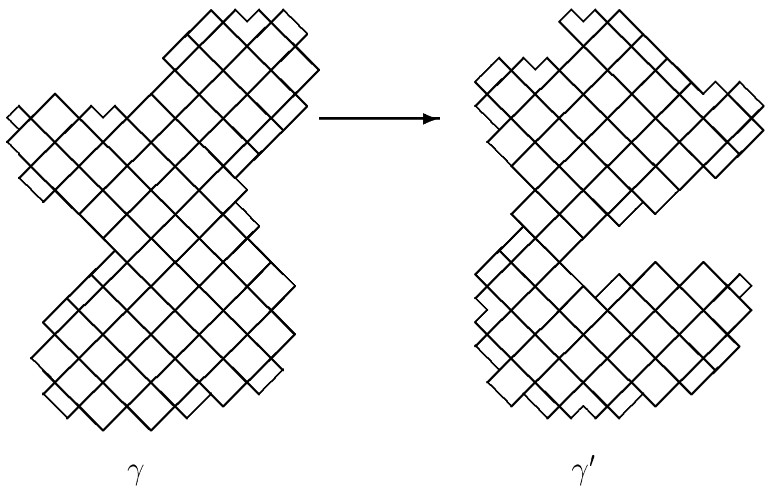



Fig. 1. A Transition

Of course this idea is hardly new and examples can be found in any book on automata or formal languages. Its application to the definition of programming languages can be found in the work of Landin and the Vienna Group ([Lan],[Oll],[Weg])

Structures of the form, $\langle \Gamma, \longrightarrow \rangle$ are rather simple and later we will consider several more elaborate variants, tailored to individual circumstances. For example it is often helpful to have an idea of *terminal* (= *final* = *halting*) configurations.

Definition 1.2.2: A *Terminal Transition System* (tts) is a structure $\langle \Gamma, \longrightarrow, T \rangle$ where $\langle \Gamma, \longrightarrow \rangle$ is a ts, and $T \subseteq \Gamma$ (the set of *final* configurations) satisfies $\forall \gamma \in T \, \forall \gamma' \in \Gamma \, . \, \gamma \not\longrightarrow \gamma'$.

A point to watch is to make a distinction between *internal* and *external* behavior. Internally a system's behavior is nothing but the sum of its transitions. (We ignore here the fact that often these transitions make sense only at a certain level; what counts as one transitions for one purpose may in fact consist of many steps when viewed in more detail. Part of the spirit of our method is to choose steps of the appropriate "size".) However externally many of the transitions produce no detectable effect. It is a matter of experience to choose the right definition of external behavior. Often two or more definitions of behavior (or of having the same behavior) are possible for a given transition system. Indeed on occasion one must turn the problem around and look for a transition system which makes it possible to obtain an expected notion of behavior.

## 1.3 Examples of Transition Systems

We recall a few familiar and not so familiar examples from computability and formal languages.

### 1.3.1 Finite Automata

A *finite automaton* is a quintuplet ($M = \langle Q, \Sigma, \delta, q_0, F \rangle$) where

- $Q$ is a finite set (of *states*)
- $\Sigma$ is a finite set (the *input* alphabet)
- $\delta : Q \times \Sigma \longrightarrow \mathcal{P}(Q)$ (is the *state transition relation*)
- $q_0 \in Q$ (is the *initial state*)
- $F \subseteq Q$ (is the set of *final states*)

To obtain a transition system we set

$$\Gamma = Q \times \Sigma^*$$

So any configuration, $\gamma = \langle q, w \rangle$ has a *state* component, $q$, and a *control* component, $w$, for data.

For the transitions we put whenever $q' \in \delta(q, a)$ :

$$\langle q, aw \rangle \vdash \langle q', w \rangle$$

(More formally, $\vdash = \{\langle\langle q, aw\rangle, \langle q', w\rangle\rangle \mid q, q' \in Q, a \in \Sigma, w \in \Sigma^*, q' \in \delta(q, a)\}$).

The behavior of a finite automaton is just the set $L(M)$ of strings it accepts:

$$L(M) = \{w \in \Sigma^* \mid \exists q \in F\ \langle q_0, w\rangle \vdash^* \langle q, \varepsilon\rangle\}$$

Of course we could also define the terminal configurations by:

$$T = \{\langle q, \varepsilon\rangle \mid q \in F\}$$

and then

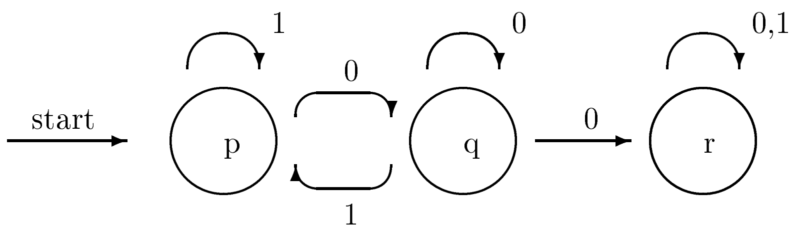
$$L(M) = \{w \in \Sigma^* \mid \exists \gamma \in T\ \langle q_0, w\rangle \vdash^* \gamma\}$$

In fact we can even get a little more abstract. Let $\langle \Gamma, \longrightarrow, T\rangle$ be a tts. An input function for it is any mapping in: $I \longrightarrow \Gamma$ and the *language* it accepts is then $L(\Gamma) \subseteq I$ where:

$$L(\Gamma) = \{i \in I \mid \exists \gamma \in T \,.\, \mathrm{in}(i) \longrightarrow^* \gamma\}$$

(For finite automata as above we take $I = \Sigma^*$, and $\mathrm{in}(w) = \langle q_0, w\rangle$). Thus we can easily formalize at least one general notion of behavior.

**Example**: The machine:




A transition sequence:

$$\begin{aligned}\langle p, 01001\rangle &\vdash \langle q, 1001\rangle \vdash \langle p, 001\rangle\\ &\vdash \langle q, 01\rangle \quad \vdash \langle r, 1\rangle\\ &\vdash \langle r, \varepsilon\rangle\end{aligned}$$

### 1.3.2 Three Counter Machines

We have three counters, $C$, namely I, J and K. There are instructions, $O$, of the following four types:

- **Increment**: **inc** $C : m$
- **Decrement**: **dec** $C : m$
- **Zero Test**: **zero** $C : m/n$
- **Stop**: **stop**

Then *program* are just sequences $P = O_1, \ldots, O_l$ of instructions. Now, fixing $P$, the set of *configurations* is:

$$\Gamma = \{\langle m, i, j, k\rangle \mid 1 \leq m \leq l;\ i, j, k \in \mathrm{N}\}$$

Then the transition relation is defined in terms of the various possibilities by:

- Case II: $O_m = \textbf{inc}\ \mathrm{I} : m'$

  $$\langle m, i, j, k\rangle \vdash \langle m', i+1, j, k\rangle$$

- Case ID: $O_m = \textbf{dec}\ \mathrm{I} : m'$

  $$\langle m, i+1, j, k\rangle \vdash \langle m', i, j, k\rangle$$

- Case IZ: $O_m = \textbf{zero}\ \mathrm{I} : m'/m''$

  $$\begin{array}{l} \langle m, 0, j, k\rangle \vdash \langle m', 0, j, k\rangle \\ \langle m, i+1, j, k\rangle \vdash \langle m'', i+1, j, k\rangle \end{array}$$

and similarly for J and K.

**Note 1** There is no case for the stop instruction.

**Note 2** In case $m'$ or $m''$ are 0 or $> k$ the above definitions do not (of course!) apply.

**Note 3** The transition relation is *deterministic*, that is:

$$\forall \gamma, \gamma', \gamma'' \cdot \gamma \longrightarrow \gamma' \wedge \gamma \longrightarrow \gamma'' \Rightarrow \gamma' = \gamma''$$

or, diagrammatically:

$$\begin{tikzcd} & \gamma \arrow[dl] \arrow[dr] & \\ \gamma' \arrow[rr, equal] & & \gamma'' \end{tikzcd}$$

(*Exercise* – prove this).

Now the set of *terminal* configurations is defined by:

$$T = \{\langle m, 0, j, 0\rangle \ \mid\ O_m = \textbf{stop}\}$$

and the behavior is a partial function $f : N \underset{P}{\longrightarrow} N$ where:

$$f(i) = j \quad \equiv_{\text{def}} \quad \langle 1, i, 0, 0\rangle \longrightarrow^* \langle m, 0, j, 0\rangle \in T$$

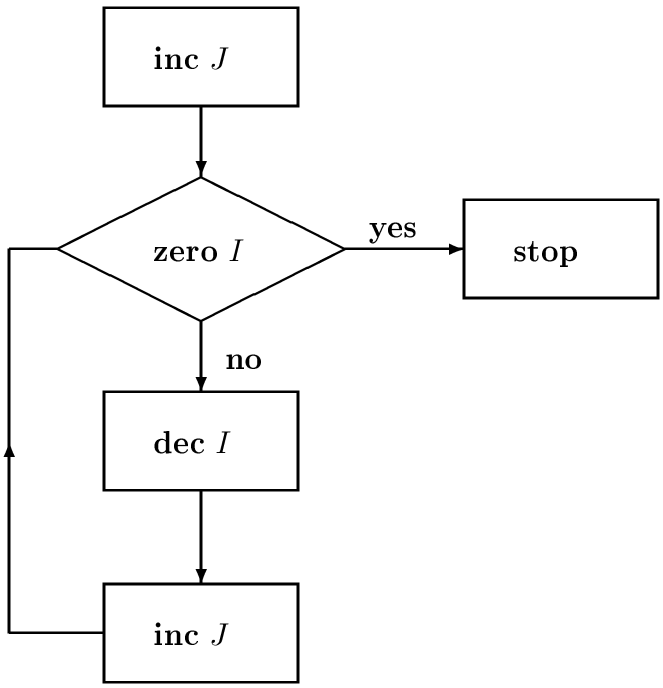
This can be put a little more abstractly, if we take for any tts $\langle \Gamma, \longrightarrow, T\rangle$ an *input* function, $\text{in} : I \longrightarrow \Gamma$ as before and also an *output* function, $\text{out} : T \longrightarrow O$ and define a partial function $f_\Gamma : I \underset{P}{\longrightarrow} O$ by

$$f_\Gamma(i) = o \quad \equiv \quad \exists \gamma \ \text{in}(i) \longrightarrow^* \gamma \in T \wedge o = \text{out}(\gamma)$$

Of course for this to make sense the tts must be *deterministic* (why?). in the case of a three-counter machine we have

$$\begin{cases} I = O = \mathrm{N} \\ \text{in}(i) = \langle 1, i, 0, 0\rangle \\ \text{out}(\langle m, i, j, k\rangle) = j \end{cases}$$

**Example**: A program for the successor function, $n \mapsto n + 1$




### 1.3.3 Context-Free Grammars

A *context-free grammar* is a quadruple, $G = \langle N, \Sigma, P, S\rangle$ where

- $N$ is a finite set (of *non-terminals*)
- $\Sigma$ is a finite set (the *input alphabet*)

- $P \subseteq N \times (N \cup \Sigma)^*$ (is the set of *productions*)
- $S \in N$ (is the *start symbol*)

Then the configurations are given by:

$$\Gamma = (N \cup \Sigma)^*$$

and the *transition relation* $\Rightarrow$ is given by:

$$wXv \Rightarrow wxv \qquad \text{(when } X \to x \text{ is in } P\text{)}$$

Now the *behaviour* is just

$$L(G) = \{w \in T \mid S \Rightarrow^* w\} \qquad (T = \Sigma^*)$$

Amusingly, this already does not fit into our abstract idea for behaviours as sets (the one which worked for finite automata). The problem is that was intended for *acceptance* where here we have to do with *generation* (by leftmost derivations).

**Exercise**: Write down an abstract model of generation.

**Example**: The grammar is:

$$\begin{aligned}&S \to\\ &S \to (S)\\ &S \to SS\end{aligned}$$

and a transition sequence could be

$$\begin{aligned}\text{S} &\Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S)\\ &\Rightarrow ()(SS) \Rightarrow^2 ()(()S)\\ &\Rightarrow^2 ()(()())\end{aligned}$$

## 1.4 Labelled Transition Systems

Transition systems in general do not give the opportunity of saying very much about any individual transition. By adding the possibility of such information we arrive at a definition.

Definition 1.4: A *Labeled Transition System* (lts) is a structure $\langle \Gamma, A, \longrightarrow \rangle$ where $\Gamma$ is a set (of *configurations*) and $A$ is a set (of *actions* (= *labels*= *operations*)) and

$$\longrightarrow \subseteq \Gamma \times A \times \Gamma$$

is the *transition relation*!

We write a transitions as: $\gamma \xrightarrow{a} \gamma'$ where $\gamma$, $\gamma'$ are configurations and $a$ is an action. The idea is that an action can give information about what went on in the configuration during the transition (*internal* actions) or about the interaction between the system and its environment (external actions) (or both). The labels are particularly useful for specifying distributed systems where the actions may relate to the communications between sub-systems. The idea seems to originate with Keller (see [Kel]).

The idea of *Labeled Terminal Transition Systems* $\langle \Gamma, A, \longrightarrow, T \rangle$ should be clear to the reader who will also expect the following generalization of reflective (resp. transitive) closure. For any lts let $\gamma$ and $\gamma'$ be configurations and take $X = a_1, \ldots, a_k$ in $A^+$ (resp. $A^*$) then:

$$\gamma \longrightarrow^{+} \overset{X}{\text{(resp. } *)} \gamma' \quad \equiv_{\text{def}} \quad \exists \gamma_1, \ldots, \gamma_k.\ \gamma \xrightarrow{a_1} \gamma_1, \ldots, \xrightarrow{a_k} \gamma_k = \gamma'$$

where $k > 0$ (resp. $k \geq 0$).

**Example 1.4.1 Finite Automata (continued)**: This time define an lts by taking

- $\Gamma = Q$
- $A = \Sigma$
- $q \xrightarrow{a} q' \equiv q' \in \delta(q, a)$
- $T = F$

Then we have $L(M) = \{w \in A^* \mid \exists q \in T q_0 \xrightarrow{w}{}^{*} q\}$. The example transition sequence given above now becomes simply:

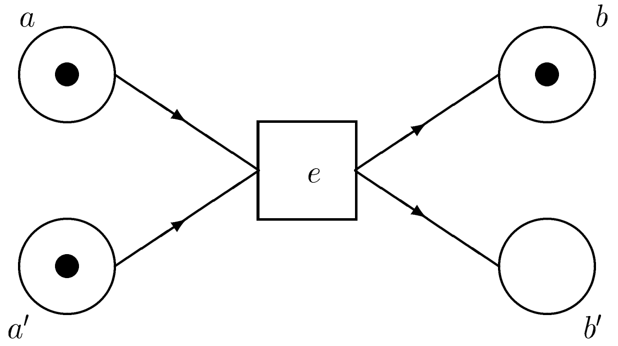
$$p \xrightarrow{0} q \xrightarrow{1} p \xrightarrow{0} q \xrightarrow{0} r \xrightarrow{1} r \in F$$

**Example 1.4.2 Petri Nets**: One idea of a Petri Net is just a quadruple $N = \langle B, E, F, m \rangle$ where

- $B$ is a finite set (of *conditions*)
- $E$ is a finite set (of *events*)
- $F \subseteq (B \times E) \cup (E \times B)$ (is the *flow* relation)
- $m \subseteq B$ (is the *initial* case)

A configuration, $m$, is *contact-free* if

$$\neg \exists e \in E.\ (F^{-1}(e) \subseteq m \wedge F(e) \cap m \neq \phi)$$




A contact situation for $m = a, a^1, b$

The point of this definition is that the occurence of an event, $e$, is nothing more than the ceasing-to-hold of its *precondition* ( $= F^{-1}(e)$) and the starting-to-hold of its *postconditions* ( $= F(e)$) in any given case. Here a case is a set of conditions (those that hold in the case). A contact-situation is one where this idea does not make sense. Often one excludes this possibility axiomatically (and imposes also other intuitively acceptable axioms). We will just (somewhat arbitrarily) regard them as "runtime errors" and take

$$\Gamma = \{m \subseteq B \mid m \text{ is } \textit{contact-free}\}$$

If two different events share a precondition in a case, then according to the above intentions they cannot both occur at once. Accordingly we define a *conflict relation* between events by:

$$e \sharp e' \quad \equiv \quad (F^{-1}(e) \cap F^{-1}(e') \neq \phi \wedge e \neq e')$$

An event can occur from a given case if all its preconditions hold in the case. What is (much) more, Petri Nets model *concurrency* in that several events (not in conflict) can occur together in a given case. So we put

$$A = \{X \subseteq E \mid \neg \exists e, e' \in X . e \sharp e'\}$$

and define

$$m \xrightarrow{X} m' \quad \equiv \quad F^{-1}(X) \subseteq m \wedge m' = [m \backslash F^{-1}(X)] \cup F(X)$$

Here is a pictorial example of such a transition

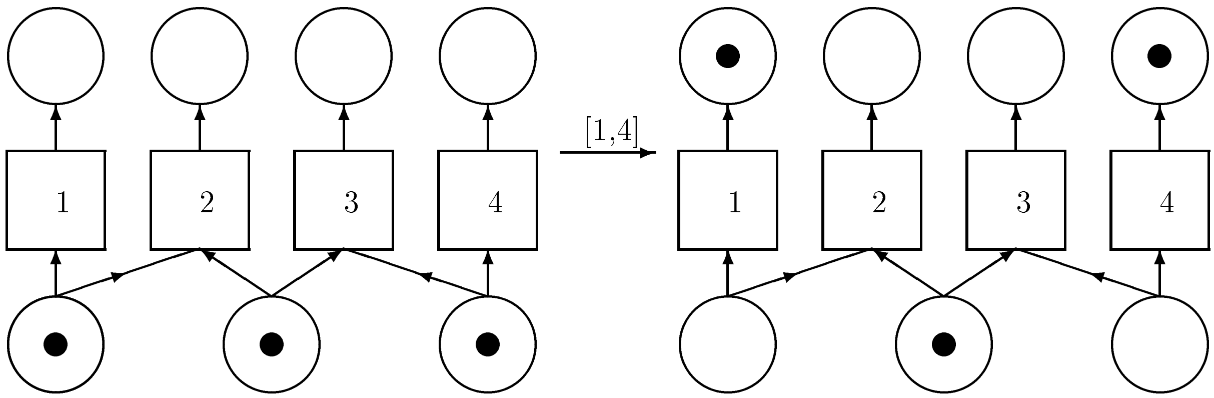
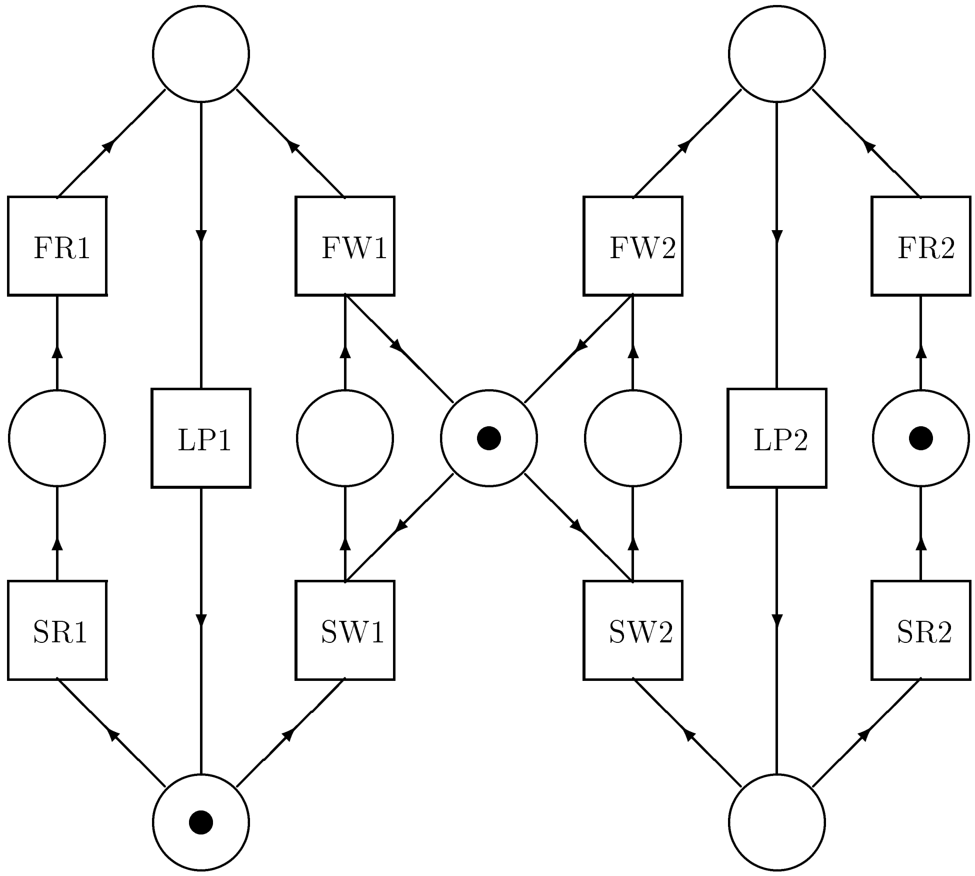



Fig. 2. A Transition

We give no definition of behavior as there does not seem to be any generally accepted one in the literature. For further information on Petri Nets see ([Bra], [Pet]).

Of course our transitions with their actions must also be thought of as kinds of events; even more so when we are discussing the semantics of languages for concurrency. We believe there are very strong links between our ideas and those in Net Theory, but, alas, do not have time here to pursue them.

**Example Readers and Writers**: This is a (partial) specification of a Readers and Writers problem with two agents each of whom can read and write (and do some local processing) but where the writes should not overlap.




SW$i$ is Start Writing $i$
FW$i$ is Finish Writing $i$
SR$i$ is Start Reading $i$
FR$i$ is Finish Reading $i$
LR$i$ is Local Processing $i$ where $1 \leq i \leq 2$

# 1.5 Interpreting Automata

To finish Chapter 1 we give an example of how to define the operational semantics of a language by an interpreting automaton. The reader should obtain some feeling for what ti possible along these lines (see the references given above for more information), as well as a feeling that the method is somehow a little too indirect thus paving the way for the approach taken in the next chapter.

## 1.5.1 The Language L

We begin with the *Abstract Syntax* of a very simple programming language called L. What is abstract about it will be discussed a little here and later at greater length. For us syntax is a collection of syntactic sets of phrases; each set corresponds to a different type of phrase. Some of these sets are very simple and can be taken as given:

- Basic Syntactic Sets

  **Truthvalues** This is the set $\mathrm{T} = \{\mathrm{tt}, \mathrm{ff}\}$ and is ranged over by (the metavariable) $t$ (and we also happily employ for this (and any other) metavariable sub- and super-scripts to generate other metavariables: $t', t_0, t'''_{1k}$).

  **Numbers** $m$, $n$ are the metavariables over $\mathrm{N} = 0, 1, 2, \ldots$.

  **Variables** $v \in \mathrm{Var} = \{a, b, c, \ldots, z\}$

  Note how we have progressed to a fairly spare style of specification in the above.

- Derived Syntactic Sets

  **Expressions** $e \in \mathrm{Exp}$ given by

  $$e ::= m \mid v \mid e + e' \mid e - e' \mid e \times e'$$

  **Boolean Expressions** $b \in \mathrm{BExp}$ given by

  $$b ::= t \mid e = e' \mid b \textbf{ or } b' \mid {\sim}b$$

  **Commands** $c \in \mathrm{Com}$ given by

  $$c ::= \textbf{nil} \mid v := e \mid c; c' \mid \textbf{ if } b \textbf{ then } c \textbf{ else } c' \mid \textbf{while } b \textbf{ do } c$$

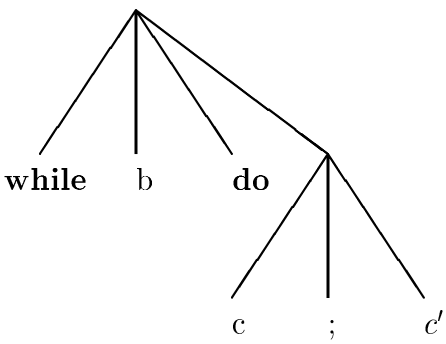
This specification can be taken, roughly speaking, as a context-free grammar if the reader just ignores the use of the infinite set N and the use of primes. It can also (despite appearances!) be taken as *unambiguous* if the reader just regards the author as having lazily omitted brackets as in:

$$b ::= t \mid e = e' \mid b \textbf{ or } b' \mid {\sim}b$$

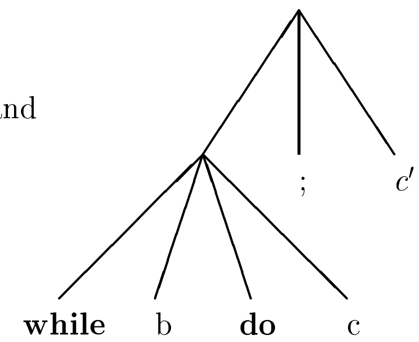
specifying parse trees so that rather than saying ambiguously that (for example):

$$\textbf{while } b \textbf{ do } c;\ c'$$

is a program what is being said is that both




and




are trees.

So we are abstract in not worrying about some lexical matters and just using for example integers rather than numerals and in not worrying bout the exact specification of phrases. What we are really trying to do is abstract away from the problems of parsing the token strings that really came into the computer and considering instead the "deep structure" of programs. Thus the syntactic categories we choose are supposed to be those with *independent semantic significance*; the various program constructs – such as semicolon or **while** ... **do** ... – are the constructive operations on phrases that possess semantic significance.

For example contrast the following concrete syntax for (some of) our expressions (taken from [Ten]):

$$
\begin{array}{ll}
\langle\text{expression}\rangle & ::= \langle\text{term}\rangle \;\mid\; \langle\text{expression}\rangle\ \langle\text{addop}\rangle\ \langle\text{term}\rangle \\
\langle\text{term}\rangle & ::= \langle\text{factor}\rangle \;\mid\; \langle\text{term}\rangle\ \langle\text{multop}\rangle\ \langle\text{factor}\rangle \\
\langle\text{factor}\rangle & ::= \langle\text{variable}\rangle \;\mid\; \langle\text{literal}\rangle \;\mid\; (\langle\text{expression}\rangle) \\
\langle\text{addop}\rangle & ::= + \mid - \\
\langle\text{multop}\rangle & ::= \times \\
\langle\text{variable}\rangle & ::= \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \ldots \mid \mathrm{z} \\
\langle\text{literal}\rangle & ::= 0 \mid 1 \mid \ldots \mid 9
\end{array}
$$

Now, however convenient it is for a parser to distinguish between $\langle$expression$\rangle$, $\langle$term$\rangle$ and $\langle$factor$\rangle$ it does not make much semantic sense!

Thus we will never give semantics directly to token strings but rather to their real structure. However, we can always obtain the semantics of token strings via *parsers* which we regard as essentially just maps:

$$\text{Parser: Concrete Syntax} \longrightarrow \text{Abstract Syntax}$$

Of course it is not really so well-defined what the abstract syntax for a given language is, and we shall clearly make good use of the freedom of choice available.

Returning to our language L we observe the following "dependency diagram":

$$
\begin{tikzcd}
\mathrm{C} \arrow[d] \arrow[dr] & \\
\mathrm{B} \arrow[r] & \mathrm{E}
\end{tikzcd}
$$

### 1.5.2 The SMC-Machine

Now we define a suitable transition system whose configurations are those of the SMC-machine.

- Value Stacks is ranged over by $S$ and is the set $(\mathrm{T} \cup \mathrm{N} \cup \mathrm{Var} \cup \mathrm{BExp} \cup \mathrm{Com})^*$

- Memories is ranged over by $M$ and is $\mathrm{Var} \longrightarrow \mathrm{N}$

- Control Stacks is ranged over by $C$ and is

$$(\mathrm{Com} \cup \mathrm{BExp} \cup \mathrm{Exp} \cup \{+, -, \times, =, \mathbf{or}, \sim, :=, \mathbf{if}, \mathbf{while}\})^*$$

The set of configurations is

$$\Gamma = \text{Value Stacks} \times \text{Memories} \times \text{Control Stacks}$$

and so a typical configuration is $\gamma = \langle S, M, C \rangle$. The idea is that we *interpret* commands and produce as our interpretation proceeds, stacks $C$, of control information (initially a command but later bits of commands). Along the way we accumulate partial results (when evaluating expressions), and bits of command text which will be needed later; this is all put (for some reason) on the value stack, $S$. Finally we have a model of the *store* (= *memory*) as a function $M : \mathrm{Var} \longrightarrow \mathrm{N}$ which given a variable, $v$, says what its value $M(v)$ is in the store.

**Notation**: In order to discuss updating variables, we introduce for a memory, $M$, integer, $m$, and variable $v$ the memory $M' = M[m/v]$ where

$$M'(v') = \begin{cases} m & (\text{if } v' = v) \\ M(v') & (\text{otherwise}) \end{cases}$$

So $M[m/v]$ is the memory resulting from updating $M$ by changing the value of $v$ from $M(v)$ to m.

*The transition relation*, $\Rightarrow$, is defined by cases according to what is on the top of the control stack.

- Expressions

$$\begin{array}{lll}
En & \langle S, M, n\ C \rangle & \Rightarrow \langle n\ S, M, C \rangle \\
Ev & \langle S, M, v\ C \rangle & \Rightarrow \langle M(v)\ S, M, C \rangle \\
E\overset{+}{\underset{\times}{-}}I & \langle S, M, e \overset{+}{\underset{\times}{-}} e'\ C \rangle & \Rightarrow \langle S, M, e\ e'\ \overset{+}{\underset{\times}{-}}\ C \rangle \\
E\overset{+}{\underset{\times}{-}}E & \langle m'\ m\ S, M, \overset{+}{\underset{\times}{-}} C \rangle & \Rightarrow \langle n\ S, M, C \rangle \\
 & & \quad (\text{where } n = m \overset{+}{\underset{\times}{-}} m')
\end{array}$$

**Note**: The symbols $+$, $-$, $\times$, are being used both as symbols of L and to stand for the functions addition, subtraction and multiplication.

- Boolean Expressions

$$
\begin{array}{lll}
Bt & \langle S, M, t\ C\rangle & \Rightarrow \langle t\ S, M, C\rangle \\
B{=}I & \langle S, M, e = e'\ C\rangle & \Rightarrow \langle S, M, e\ e' = C\rangle \\
B{=}E & \langle m'\ m\ S, M, =C\rangle & \Rightarrow \langle t\ S, M, C\rangle \\
 & & \quad \text{(where } t = (m = m')\text{)} \\
B\mathbf{or}I & \langle S, M, b\ \mathbf{or}\ b'\ C\rangle & \Rightarrow \langle S, M, b\ b'\ \mathbf{or}\ C\rangle \\
B\mathbf{or}E & \langle t'\ t\ S, M, \mathbf{or}\ C\rangle & \Rightarrow \langle t''\ S, M, C\rangle \\
 & & \quad \text{(where } t'' = (t \vee t')\text{)} \\
B{\sim}I & \langle S, M, {\sim}b\ C\rangle & \Rightarrow \langle S, M, b \sim C\rangle \\
B{\sim}E & \langle t\ S, M, \sim C\rangle & \Rightarrow \langle t'\ S, M, C\rangle \\
 & & \quad \text{(where } t' = {\sim}t\text{)}
\end{array}
$$

- Commands

$$
\begin{array}{lll}
C\mathbf{nil} & \langle S, M, \mathbf{nil}\ C\rangle & \Rightarrow \langle S, M, C\rangle \\
C{:=}I & \langle S, M, v := e\ C\rangle & \Rightarrow \langle v\ S, M, e := C\rangle \\
C{:=}E & \langle m\ v\ S, M, := C\rangle & \Rightarrow \langle S, M[m/v], C\rangle \\
C; & \langle S, M, c; c'\ C\rangle & \Rightarrow \langle S, M, c\ c'\ C\rangle \\
C\mathbf{if}I & \langle S, M, \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'\ C\rangle & \Rightarrow \langle c\ c'\ S, M, b\ \mathbf{if}\ C\rangle \\
C\mathbf{if}E & \langle t\ c\ c'\ S, M, \mathbf{if}\ C\rangle & \Rightarrow \langle S, M, c''\ C\rangle \\
 & \multicolumn{2}{r}{\text{(where if } t = \mathrm{tt} \text{ then } c'' = c \text{ else } c'' = c'\text{)}} \\
C\mathbf{while}I & \langle S, M, \mathbf{while}\ b\ \mathbf{do}\ c\ C\rangle & \Rightarrow \langle b\ c\ S, M, b\ \mathbf{while}\ C\rangle \\
C\mathbf{while}E1 & \langle \mathrm{tt}\ b\ c\ S, M, \mathbf{while}\ C\rangle & \Rightarrow \langle S, M, c\ \mathbf{while}\ b\ \mathbf{do}\ c\ C\rangle \\
C\mathbf{while}E2 & \langle \mathrm{ff}\ b\ c\ S, M, \mathbf{while}\ C\rangle & \Rightarrow \langle S, M, C\rangle
\end{array}
$$

Now that we have at some length defined the transition relation, the terminal configurations are defined by:

$$T = \{\langle \varepsilon, M, \varepsilon\rangle\}$$

and an input function in : Commands $\times$ Memories $\longrightarrow \Gamma$ is defined by:

$$\mathrm{in}(C, M) = \langle \varepsilon, M, C\rangle$$

and out : $T \longrightarrow$ Memories by:

$$\mathrm{out}(\langle \varepsilon, M, \varepsilon\rangle) = M$$

The behavior of the SMC-machine is then a partial function, Eval : Commands $\times$ Memories $\underset{P}{\longrightarrow}$ Memories and clearly:

$$\mathrm{Eval}(C, M) = M' \qquad \equiv \qquad \langle \varepsilon, M, C\rangle \Rightarrow^* \langle \varepsilon, M', \varepsilon\rangle$$

**Example Factorial**:

$$\mathrm{y} := 1;\ \underbrace{\textbf{while} \sim(\mathrm{x} = 0)\ \textbf{do}\ \overbrace{\mathrm{y} := \mathrm{y} \times \mathrm{x}; \mathrm{x} := \mathrm{x} - 1}^{C'}}_{C}$$

$$\begin{array}{ll}
\langle \varepsilon, \langle 3,5\rangle, \mathrm{y} := 1; C\rangle & \\
\Rightarrow \langle \varepsilon, \langle 3,5\rangle, \mathrm{y} := 1\ C\rangle & \text{by } C; \\
\Rightarrow \langle \mathrm{y}, \langle 3,5\rangle, 1\ :=\ C\rangle & \text{by } C := I \\
\quad \Rightarrow \langle 1\ \mathrm{y}, \langle 3,5\rangle, :=\ C\rangle & \text{by } Em \\
\Rightarrow \langle \varepsilon, \langle 3,1\rangle, C\rangle & \text{by } C{:=}E \\
\Rightarrow \langle \sim(\mathrm{x}=0)\ C', \langle 3,1\rangle, \sim(\mathrm{x}=0)\ \textbf{while}\rangle & \text{by } C\textbf{while}I \\
\quad \Rightarrow \langle \sim(\mathrm{x}=0)\ C', \langle 3,1\rangle, (\mathrm{x}=0) \sim \textbf{while}\rangle & \text{by } E{\sim}I \\
\qquad \Rightarrow \langle \sim(\mathrm{x}=0)\ C', \langle 3,1\rangle, \mathrm{x}\ 0\ =\ \sim \textbf{while}\rangle & \text{by } E{=}I \\
\qquad\quad \Rightarrow \langle 3 \sim(\mathrm{x}=0)\ C', \langle 3,1\rangle, 0\ =\ \sim \textbf{while}\rangle & \text{by } Ev \\
\qquad\quad \Rightarrow \langle 0\ 3 \sim(\mathrm{x}=0)\ C', \langle 3,1\rangle, =\ \sim \textbf{while}\rangle & \text{by } Em \\
\qquad \Rightarrow \langle \mathrm{ff} \sim(\mathrm{x}=0)\ C', \langle 3,1\rangle, \sim \textbf{while}\rangle & \text{by } E{=}E \\
\quad \Rightarrow \langle \mathrm{tt} \sim(\mathrm{x}=0)\ C', \langle 3,1\rangle, \textbf{while}\rangle & \text{by } E{\sim}E \\
\Rightarrow \langle \varepsilon, \langle 3,1\rangle, C'\ C\rangle & \text{by } C\textbf{while}E1 \\
\Rightarrow \langle \varepsilon, \langle 3,1\rangle, \mathrm{y} := \mathrm{y} \times \mathrm{x}\ \mathrm{x} := \mathrm{x} - 1\ C\rangle & \text{by } C; \\
\Rightarrow^* \langle \varepsilon, \langle 3,3\rangle, \mathrm{x} := \mathrm{x} - 1\ C\rangle & \\
\Rightarrow^* \langle \varepsilon, \langle 2,3\rangle, C\rangle & \\
\Rightarrow^* \langle \varepsilon, \langle 1,6\rangle, C\rangle & \\
\Rightarrow^* \langle \varepsilon, \langle 0,6\rangle, C\rangle & \\
\Rightarrow \langle \sim(\mathrm{x}=0)\ C', \langle 0,6\rangle, \sim(x=0)\ \textbf{while}\rangle & \text{by } C\textbf{while}I \\
\Rightarrow^* \langle \mathrm{ff} \sim(x=0)\ C', \langle 0,6\rangle, \textbf{while}\rangle & \\
\Rightarrow \langle \varepsilon, \langle 0,6\rangle, \varepsilon!\rangle & \text{by } C\textbf{while}E2
\end{array}$$

Many other machines have been proposed along these lines. It is, perhaps, fair to say that none of them can be considered as *directly formalizing* the *intuitive* operational semantics to be found in most language definitions. Rather they are more or less clearly correct on the *basis* of this intuitive understanding. Further, although this is of less importance, they all have a tendency to pull the syntax to pieces or at any rate to wander around the syntax creating various complex symbolic structures which do not seem particularly forced by the demands of the language itself. Finally, they do not in general have any great claim to begin *syntax-directed* in the sense of defining the semantics of compound phrases in terms of the semantics of their components, although the definition of the transition relation does fall into natural cases following the various syntactical possibilities.

## 1.6 Exercises

### Finite Automata

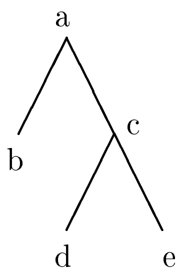
Let $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ be a finite automaton.

**1.** Redefine the behavior of $M$ so that it accepts *infinite* strings $a_1 a_2 \ldots a_n \ldots$, that is so that $L(M) \subseteq \Sigma^\omega$ [Hint: There are actually two answers, which can with difficulty be proved equivalent].

**2.** Suppose that $\delta$ were changed so that the labeled transition relation had instead the form:

$$q \xrightarrow{a} q_1, q_2$$

and $F$ so that $F \subseteq Q \times \Sigma$. What is the new type of $\delta$? How can *binary trees* like




now be accepted by $M$?

**3.** Suppose instead transitions occurred with *probability* so that we had

$$q \xrightarrow[p]{a} q'$$

with $0 \leq p \leq 1$ and for any $q$ and $a$:

$$\Sigma\{p \mid q \xrightarrow[p]{a} q' \text{ for some } q'\} \leq 1$$

What is a good definition of behavior now?

**4.** Finite automata can be turned into *transducer* by taking $\delta$ to be a finite set of *transitions* of the form:

$$q \xrightarrow[w]{v} q'$$

with $v$, $w \in \Sigma^*$. Define the relation $q \xrightarrow[w]{v} q'$ and the appropriate notion of behavior. Show any finite-state transducer can be turned into an equivalent one, where we have in any transition that $0 \leq |v| \leq 1$.

## Various Machines

**5.** Define $k$ counter machines. Show that any function computable by a $k$ counter machine is computable by a 3-counter machine. [Hint: First program elementary functions on the

3-counter machine including pairing, pair : $\mathrm{N}^2 \longrightarrow \mathrm{N}$, and selection functions, fst, snd : $\mathrm{N} \longrightarrow \mathrm{N}$ such that:

$$\begin{aligned}&\mathrm{fst}(\mathrm{pair}(m,n)) = m\\&\mathrm{snd}(\mathrm{pair}(m,n)) = n\end{aligned}$$

Then simulate by coding all the registers of the $k$ counter machine by a big tuple held in one of the registers of the 3-counter machine]. Show that any partial-recursive function (= one computable by a Turing Machine) can be computed by some 3-counter machine (and vice-versa).

**6.** Consider stack machines where the registers hold stacks and operations on a stack (= element of $\Sigma^*$) are $\mathrm{push}_a$, pop, $\mathrm{ishd}_a$ (for each $a \in \Sigma$) given by:

$$\begin{aligned}&\mathrm{push}_a(w) = aw\\&\mathrm{pop}(aw) = w\\&\mathrm{ishd}_a(w) = \begin{cases}\text{true} & (\text{if } w = aw' \text{ for some } w')\\ \text{false} & (\text{otherwise})\end{cases}\end{aligned}$$

Show stack machines compute the same functions as Turing Machines. How many stacks are needed at most?

**7.** Define and investigate queue machines.

**8.** See how your favorite machines (Turing Machines, Push-Down Automata) fit into our framework. For a general view of machines, consult the eminently readable: [Bir] or [Sco]. Look too at [Gre].

## Grammars

**9.** For CF grammars our notion of behavior is adapted to *generation*. Define a notion that is good for *acceptance*. What about mixed generation/acceptance? Change the definitions so that you get parse trees as behaviors. What is the nicest way you can find to handle syntax-directed translation schemes?

**10.** Show that for LL(1) grammars you can obtain *deterministic* labeled (with $\Sigma$) transitions of the form

$$w \xrightarrow{a} w'$$

with $w$ strings of terminals and non-terminals. What can you say about LL($k$), LR($k$)?

**11.** Have another look at other kinds of grammar too, e.g. Context-Sensitive, Type 0 (= arbitrary) grammars. Discover other ideas for Transition Systems in the literature. Examples include: Tag, Semi-Tue Systems, Markov Algorithms, $\lambda$-Calculus, Post Systems, Tag, L-Systems, Conway's Game of Life and other forms of Cell Automata, Kleene's Nerve Nets ...

## Petri Nets

**12.** Show that if we have

$$\begin{tikzcd} & \mathrm{m} \arrow[dl, "X"'] \arrow[dr, "X'"] & \\ m' & & m'' \end{tikzcd}$$

where $F^{-1}(X) \cap F^{-1}(X') = \phi$ (i.e. no *conflict* between $X$ and $X'$) then for some $m'''$ we have:

$$\begin{tikzcd} & m \arrow[dl, "X"'] \arrow[dr, "X'"] \arrow[dd, "Y"] & \\ m' \arrow[dr, "X'"'] & & m'' \arrow[dl, "X"] \\ & m''' & \end{tikzcd} \qquad \text{where } Y = X \cup X'$$

This is a so-called *Chyrch-Rosser Property*.

**13.** Show that if we have $m \xrightarrow{X} m'$ where $X = \{e_1, \ldots, e_k\}$ then for some $m_1, \ldots, m_k$ we have:

$$m \xrightarrow{e_1} m_1 \xrightarrow{e_2} \cdots \xrightarrow{e_k} m_k = m$$

What happens if we remove the restrictions on finiteness?

**14.** Write some Petri Nets for a parallel situation you know well (e.g. from something you knew at home or some computational situation).

**15.** How can nets accept languages (= subsets of $\Sigma^*$)? Are they always regular?

**16.** Find, for the Readers and Writers net given above, all the cases you can reach by transition sequences starting at the initial case. Draw (nicely!) the graph of cases and transitions (this is a so-called case graph).

## Interpreting Automata

**17.** Let $G = \langle N, \Sigma, P, S \rangle$ be a context-free grammar. It is *strongly unambiguous* if there are no two left most derivations of the same word in $\Sigma^*$, even possibly starting from different non-terminals. Find suitable conditions on the productions of $P$ which ensure that $G' = \langle N, \Sigma', P', S' \rangle$ is strongly unambiguous where $\Sigma' = \Sigma \cup \{(\ ,\ )\}$ where the parentheses are assumed not to be in $N$ or $\Sigma$ and where

$$T \longrightarrow (w) \text{ is in } P' \text{ if } T \longrightarrow w \text{ is in } P.$$

**18.** See what changes you should make in the definition of the interpreting automaton when some of the following features are added:

$$e ::= \textbf{if } b \textbf{ then } e \textbf{else } e \ \mid\ \textbf{begin } c \textbf{ result } e$$

$$\begin{array}{lll} c ::= & \textbf{if } b \textbf{ then } c \ \mid & \\ & \textbf{case } e \textbf{ of } & e_1 : c \\ & & \vdots \\ & & e_k : c \\ & \textbf{end} \ \mid & \\ & \textbf{for } v := e, e \textbf{ do } c \ \mid & \\ & \textbf{repeat } c \textbf{ until } b & \end{array}$$

**19.** Can you handle constructions that drastically change the flow of control such as:

$$c ::= \textbf{stop} \ \mid m : c \mid \ \textbf{goto } m$$

(Here **stop** just stops everything!)

**20.** Can you handle elementary read/write instructions such as:

$$c ::= \textbf{read}(v) \mid \textbf{write}(e)$$

[Hint: Consider an analogy with finite automata – especially transducers.]

**21.** Can you add facilities to the automaton to handle run-time errors?

**22.** Can you produce measures of time/space complexity by adding extra components to the automaton?

**23.** Can you treat diagnostic (debugging, tracing) facilities?

**24.** What about real-time? That is suppose we had the awful expression:

$$e ::= \textbf{time}$$

which delivers the correct time.

**25.** Treat the following PASCAL subset. The basic sets are T, N and $x \in \mathrm{I} \times \{\mathrm{i}, \mathrm{r}, \mathrm{b}\}$ – the set of typical *identifiers* (which is *infinite*) and $o \in \mathrm{O}$ – the set { =, <>, <, <=, >, >=, +, −, ×, /, **div**, **mod**, **and** } of operations. The idea for typical identifiers is that i, r, b are *type symbols* for integer, real and boolean respectively and so $\langle \mathrm{FRED}, r\rangle$ is the real identifier FRED.

The derived sets are expressions and commands where:

$$e ::= m \mid t \mid v \mid -e \mid \textbf{not}\ e \mid e\ o\ e$$
$$c ::= \textbf{nil} \mid v := e \mid c;\ c' \mid \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' \mid \textbf{while}\ e\ \textbf{do}\ c$$

The point of the question is that you must think about compile-time type checking and the memories used in the $\langle S, M, C\rangle$ machine should be finite (even although there are potentially infinitely many identifiers).

**26.** Can you treat the binding mechanism

$$s ::= i \mid r \mid b$$
$$c ::= \textbf{var}\ v : s\ \textbf{begin}\ c\ \textbf{end}$$

so that you must now incorporate symbol tables?

## 1.7 Bibliography


[Bir] Bird, R. (1976) *Programs and Machines* Wiley and Sons

[Bra] Brauer, W. (1980) *Net Theory and Applications* Proceedings 1979. LNCS Vol. 84

[Gor] Gordon, M.J.C. (1979) *The Denotational Description of Programming Languages* Berlin: Springer-Verlag

[Gre] Greibach, S.A. (1975) *Theory of Program Structures Schemes, Semantics, Verification.* LNCS, Vol. 36

[Kel] Keller, R.M. (1976) *Formal Verification of Parallel Programs* CACM 19, pages 371-384

[Lan] Landin, P.J. (1966) *A Lambda-calculus Approach, Ch. 5* Advances in Programming and Non-numerical Computation Pergamon

[Oll] Ollengren, A. (1976) *Definition of Programming Languages by Interpreting Automata* Academic Press

[Pet] Peterson, J.L. (1977) *Petri Nets* Computing Surveys, Vol. 9, pages 223-252

[Sco] Scott, D. (1967) *Some Definitional Suggestions for Automata Theory* JCSS 1 (2), pages 187-212

[Ten] Tennent, R.D. (1981) *Principles of Programming Languages* Prentice-Hall

[Weg] Wegner, P. (1972) *The Vienna Definition Language* A.C.M Computing Surveys, Vol. 1, pages 5-63

# Chapter 2

# Simple Expressions and Commands

The $\langle S, M, C \rangle$ machine emphasizes the idea of computation as a sequence of transitions involving simple data manipulations; further the definition of the transitions falls into simple cases according to the syntactic structure of the expression or command on top of the control stack. However, many of the transitions are of little intuitive importance, contradiciting our idea of the right choice of the "size" of the transitions. Further the definition of the transitions is not syntax-directed so that, for example, the transitions of $c; c'$ are not directly defined in terms of those for $c$ and those for $c'$. Finally but really the most important, the $\langle S, M, C \rangle$ machine is not a formalization of intuitive operational ideas but is rather, fairly clearly, correct *given* these intuitive ideas.

in this chapter we develop a method designed to answer these objections, treating simple expressions and commands as illustrated by th language L. We consider run-time errors and say a little on how to establish properties of transition relations. Finally we take a first look at simple type-checking.

## 2.1 Simple Expressions

Let us consider first the very simple subset of expressions given by:

$$e ::= m \mid e_0 + e_1$$

and how the $\langle S, M, C\rangle$ machine deals with them. For example we have the transition sequence for the expression $(1 + (2 + 3)) + (4 + 5)$:

$$\begin{array}{rll}
\langle \varepsilon, M, (1 + (2 + 3)) + (4 + 5)\rangle & \longrightarrow \langle \varepsilon, M, (1 + (2 + 3))\ (4 + 5)\ +\rangle & \\
& \longrightarrow \langle \varepsilon, M, 1\ (2 + 3)\ +\ (4 + 5)\ +\rangle & \\
& \longrightarrow \langle 1, M, (2 + 3)\ +\ (4 + 5)\ +\rangle & \\
& \longrightarrow \langle 1, M, 2\ 3\ +\ +\ (4 + 5)\ +\rangle & \\
& \longrightarrow \langle 2\ 1, M, 3\ +\ +\ (4 + 5)\ +\rangle & \\
& \longrightarrow \langle 3\ 2\ 1, M, +\ +\ (4 + 5)\ +\rangle & (*) \\
& \longrightarrow \langle 5\ 1, M, +\ (4 + 5)\ +\rangle & (*) \\
& \longrightarrow \langle 6, M, (4 + 5)\ +\rangle & \\
& \longrightarrow^3 \langle 5\ 4\ 6, M, +\ +\rangle & (*) \\
& \longrightarrow \langle 9\ 6, M, +\rangle & (*) \\
& \longrightarrow \langle 15, M, \varepsilon\rangle &
\end{array}$$

In these 13 transitions only the 4 additions marked $(*)$ are of any real interest as system events. Further the intermediate structures generated on the stacks are also of little interest. Preferable would be a sequence of 4 transitions on the expression itself thus:

$$\begin{array}{rl}
(1 + (2 \overset{\bigtriangledown}{+} 3)) + (4 + 5)) & \longrightarrow (1 \overset{\bigtriangledown}{+} 5) + (4 + 5) \\
& \longrightarrow 6 + (4 \overset{\bigtriangledown}{+} 5) \\
& \longrightarrow 6 \overset{\bigtriangledown}{+} 9 \\
& \longrightarrow 15
\end{array}$$

where we are ignoring the memory and we have marked the occurrences of the additions in each transition. (These transition sequences of expressions are often called *reduction* sequences (= *derivations*) and the occurrences are called *redexes*; this notation originates in the $\lambda$ - calculus (see e.g. [Hin])).

Now consider an informal specification of this kind of expression *evaluation*. Briefly one might just say one evaluates from left-to-right. More pedantically one could say:

**Constants** Any constant, $m$, is already evaluated with itself as value.

**Sums** To evaluate $e_0 + e_1$

1. Evaluate $e_0$ obtaining $m_0$, say, as result.
2. Evaluate $e_1$ obtaining $m_1$, say, as result.
3. Add $m_0$ to $m_1$ obtaining $m_2$, say, as result.

This finishes the evaluation and $m_2$ is the result of the evaluation.

Note that this specification is syntax-directed, and we use it to obtain *rules* for describing steps (= transitions) of evaluation which we think of as nothing else than a derivation of the form:

$$e = e_1 \longrightarrow e_2 \longrightarrow \ldots \longrightarrow e_{n-1} \longrightarrow e_n = m$$

(where $m$ is the result). Indeed if we just look at the first step we see from the above specification that

1. If $e_0$ is not a constant the first step of the evaluation of $e_0 + e_1$ is the first step of the evaluation of $e_0$.
2. If $e_0$ is a constant, but $e_1$ is not, the first step of the evaluation of $e_0 + e_1$ is the first step of the evaluation of $e_1$.
3. If $e_0$ and $e_1$ are constants the first (and last!) step of the evaluation of $e_0 + e_1$ is the addition of $e_0$ and $e_1$.

Clearly too the first step of evaluating an expression, $e$, can be taken as resulting in an expression $e_1$ with the property that the evaluation of $e$ is the first step followed by the evaluation of $e_1$. We now put all this together to obtain rules for the first step. These are rules for establishing binary relationships of the form:

$$e \longrightarrow e' \quad \equiv \quad e' \text{ is the result of the first step of the evaluation of } e.$$

**Rules Sum**

1. $\dfrac{e_0 \longrightarrow e_0'}{e_0 + e_1 \longrightarrow e_0' + e_1}$
2. $\dfrac{e_1 \longrightarrow e_1'}{m_0 + e_1 \longrightarrow m_0 + e_1'}$
3. $m_0 + m_1 \longrightarrow m_2$ (if $m_2$ is the sum of $m_0$ and $m_1$)

Thus, for example, rule 1 states what is obvious from the above discussion:

> If $e_0'$ is the result of the first step of the evaluation of $e_0$ then $e_0' + e_1$ is the result of the first step of the evaluation of $e_0 + e_1$.

We now take these rules as a *definition* of what relationships hold – namely exactly these we can establish from the rules. We take the above discussion as showing why this *mathematical* definition makes sense form an *intuitive view*; it is the *direct* formalization referred to above.

As an example consider the step:

$$(1 + (2 + 3)) + (4 + 5) \longrightarrow (1 + 5) + (4 + 5)$$

To establish this step we have

| | | | | |
|---|---|---|---|---|
| 1. | $2 + 3$ | $\longrightarrow$ | $5$ | (By rule 3) |
| 2. | $1 + (2 + 3)$ | $\longrightarrow$ | $1 + 5$ | (By rule 2) |
| 3. | $(1 + (2 + 3)) + (4 + 5)$ | $\longrightarrow$ | $(1 + 5) + (4 + 5)$ | (By rule 1) |

Rather than this unnatural "bottom-up" method we usually display these little proofs in the "top-down" way they are actually "discovered". The arrow is supposed to show the "direction" of discovery:

$$(1+(2+3))+(4+5) \xrightarrow{Sum1} (1+5)+(4+5))$$

$$1 + (2+3) \xrightarrow{Sum2} 1 + 5$$

$$2 + 3 \xrightarrow{Sum3} 5$$

Thus, while the evaluation takes four steps, the justification (proof) of each step has a certain size if its own (which need not be displayed). In this light the $\langle S, M, C\rangle$ machine can be viewed as mixing-up the additions with the reasons why they should be performed into one long linear sequence.

It could well be argued that our formalization is not really that direct. A more direct approach would be to give rules for the transition sequences themselves (the evaluations). For the intuitive specification refers to these evaluations rather than any hypothetical atomic actions from which they are composed. However, axiomatizing a step is intuitively simpler, and we prefer to follow a simple approach until it leads us into such difficulties that it is better to consider whole derivations.

Another point concerns the lack of formalization of our ideas. The above relies are easily turned into a formal system of formulae, axioms and rules. What we would want is a sufficiently elastic conception of a *range* of such formal systems which on the one hand allows the natural expression of all the systems of rules we wish, and on the other hand returns some profit in the form of interesting theorems about such systems or interesting computer systems based on such systems. However, The present work is too explorating for us to fix our ideas, although we may later try out one or two possibilities. We also fear that introducing such formalities could easily lead us into abscurities in the presentation of otherwise natural ideas.

Now we try out more expressions. To evaluate variables we need the memory component of the $\langle S, M, C\rangle$ machines – indeed that is the only "natural" component they have! It is convenient here to change our notation to a more generally accepted one:

| OLD | NEW |
|---|---|
| Memory | Store |
| Memories = Var $\longrightarrow$ N = S | |
| M | $\sigma$ |
| M$[m/v]$ | $\sigma[m/v]$ |

### 2.1.1 L-Expressions

Now for the expression language of L:

$$e ::= m \mid v \mid (e + e') \mid (e - e') \mid (e \times e')$$

we introduce the configurations

$$\Gamma = \{\langle e, \sigma \rangle\}$$

and the relation

$$\langle e, \sigma \rangle \longrightarrow \langle e', \sigma \rangle$$

meaning one step of the evaluation of $e$ (with store $\sigma$)results in the expression $e'$ (with store $\sigma$). The rules are just those we already have, adapted to take account of stores plus an obvious rule for printing the value of a variable in a store.

**Rules Sum**

1. $\dfrac{\langle e_0, \sigma \rangle \longrightarrow \langle e_0', \sigma \rangle}{\langle e_0 + e_1, \sigma \rangle \longrightarrow \langle e_0' + e_1, \sigma \rangle}$
2. $\dfrac{\langle e_1, \sigma \rangle \longrightarrow \langle e_1', \sigma \rangle}{\langle m + e_1, \sigma \rangle \longrightarrow \langle m + e_1', \sigma \rangle}$
3. $\langle m + m', \sigma \rangle \longrightarrow \langle n, \sigma \rangle \qquad$ (where $n = m + m'$)

**Minus**

1. Exercise for the reader.
2. Exercise for the reader.
3. $\langle m - m', \sigma \rangle \longrightarrow \langle n, \sigma \rangle \qquad$ (if $m \geq m'$ and $n = m - m'$)

**Times**

1. Exercise for the reader
2. Exercise for the reader
3. Exercise for the reader

**Variable**

1. $\langle v, \sigma \rangle \longrightarrow \langle \sigma(v), \sigma \rangle$

Note the two users of the symbol, +, in rules Sum 3: One as a syntactic construct and one for the addition function. We will often overload symbols in this way relying on the context for disambiguation. So here, for example, to make sense of $n = m + m'$ we must be meaning addition as the left-hand-side of the equation denotes an integer.

Of course the terminal configurations are those of the form $\langle m, \sigma\rangle$, and $m$ is the result of the evaluation. Note that there are configurations such as:

$$\gamma = \langle 5 + (7 - 11), \sigma\rangle$$

which are not terminal but for which there is no $\gamma'$ with $\gamma \longrightarrow \gamma'$.

Definition: Let $\langle \Gamma, T, \longrightarrow\rangle$ be a tts. A configuration $\gamma$ is *stuck* if $\gamma \notin T$ and $\neg\exists\gamma' \cdot \gamma \longrightarrow \gamma'$.

In most programming languages these stuck configurations result in run-time errors. These will be considered below.

The behavior of expressions is the result of their evaluation and is defined by:

$$\text{eval}(e, \sigma) = m \quad \Leftrightarrow \quad \langle e, \sigma\rangle \longrightarrow^* \langle m, \sigma\rangle$$

The reader will see (from 2.3 below, if needed) that eval is a well-defined partial function.

One can also define the *equivalence* of expressions by:

$$e \equiv e' \quad \Leftrightarrow \quad \forall\sigma.\text{eval}(e, \sigma) = \text{eval}(e', \sigma)$$

### 2.1.2 Boolean Expressions

Now we turn to the Boolean expressions of the language L given by:

$$b := t \mid b \textbf{ or } b' \mid e = e' \mid \sim b$$

Here we take $\Gamma\{b, \sigma\}$ and consider the rules for the transition relation. There are clearly none for truthvalues, $t$, but there are several possibilities for disjunctions, $b$ **or** $b'$. These possibilities differ not only in the order of the transitions, but even on which transitions occur. The configurations are pairs $\langle b, \sigma\rangle$.

A. **Complete Evaluation**: This is just the Boolean analogue of our rules for expressions and corresponds to the method used by our SMC-machine.

1. $$\frac{\langle b_0, \sigma\rangle \longrightarrow \langle b_0', \sigma\rangle}{\langle b_0 \text{ or } b_1, \sigma\rangle \longrightarrow \langle b_0' \textbf{ or } b_1, \sigma\rangle}$$
2. $$\frac{\langle b_1, \sigma\rangle \longrightarrow \langle b_1, \sigma\rangle}{\langle t \textbf{ or } b_1, \sigma\rangle \longrightarrow \langle t \textbf{ or } b_1' , \sigma\rangle}$$
3. $t \textbf{ or } t' \longrightarrow t''$ (where $t'' = t \vee t'$)

B. **Left-Sequential Evaluation**: This takes advantage of the fact that it is not needed to evaluate $b$, in tt **or** $b$, as the result will be tt independently of the result of evaluating $b$,

1. $$\frac{\langle b_0, \sigma\rangle \longrightarrow \langle b_0', \sigma\rangle}{\langle b_0 \textbf{ or } , b_1, \sigma\rangle \longrightarrow \langle b_0' \text{ or } b_1, \sigma\rangle}$$
2. $\langle \text{tt} \textbf{ or } b_1, \sigma\rangle \longrightarrow \langle \text{tt}, \sigma\rangle$

3. $\langle \mathrm{ff}\ \mathbf{or}\ b_1, \sigma\rangle \longrightarrow \langle b_1, \sigma\rangle$

C. **Right-Sequential Evaluation**: Like B but "backwards".

D. **Parallel Evaluation** This tries to combine the advantages of B and C by evaluating $b_0$ and $b_1$ in parallel. in practice that would mean having two processors, one for $b_1$, or using one but interleaving, somehow the evaluations of $b_0$ and $b_1$. This idea is therefore not found in the usual sequential programming languages (as opposed to these making explicit provisions for concurrency). However, it may be useful for hardware specification.

1. $$\frac{\langle b_0, \sigma\rangle \longrightarrow \langle b_0', \sigma\rangle}{\langle b_0\ \mathbf{or}\ b_1, \sigma\rangle \longrightarrow \langle b_0\ \mathbf{or}\ b_1', \sigma\rangle}$$
2. $$\frac{\langle b_1, \sigma\rangle \longrightarrow \langle b_1', \sigma\rangle}{\langle b_0\ \mathbf{or}\ b_1, \sigma\rangle \longrightarrow \langle b_0\ \mathbf{or}\ b_1', \sigma\rangle}$$
3. $\langle \mathrm{tt}\ \mathbf{or}\ b_1, \sigma\rangle \longrightarrow \langle \mathrm{tt}, \sigma\rangle$
4. $\langle b_0\ \mathbf{or}\ \mathrm{tt}, \sigma\rangle \longrightarrow \langle \mathrm{tt}, \sigma\rangle$
5. $\langle \mathrm{ff}\ \mathbf{or}\ b_1, \sigma\rangle \longrightarrow \langle b_1, \sigma\rangle$
6. $\langle b_0\ \mathbf{or}\ \mathrm{ff}, \sigma\rangle \longrightarrow \langle b_0, \sigma\rangle$

The above evaluation mechanisms are very different when subexpressions can have non-terminating evaluations, when we have the following relationships:

$$\begin{tikzcd} & \mathrm{B} \arrow[dl] & \\ \mathrm{C} & & \mathrm{A} \arrow[ul] \arrow[dl] \\ & \mathrm{C} \arrow[ul] & \end{tikzcd}$$

where $X \Rightarrow Y$ means that if method $X$ terminates with result t, so does method $Y$. We take method $A$ for the semantics of our example language L.

For Boolean expressions of the form $e = e'$ our rules depend on those for expressions, but otherwise are normal (and for brevity we omit the $\sigma'$s).

- **EQUALITY**

1. $$\frac{e_0 \longrightarrow e_0'}{e_0 = e_1 \longrightarrow e_0' = e_1}$$
2. $$\frac{e_1 \longrightarrow e_1'}{m = e_1 \longrightarrow m = e_1'}$$
3. $m = n \longrightarrow t$ (where $t$ is tt if $m = n$ and ff otherwise)

For negations $\sim b$ we have, again omitting the $\sigma'$s:

- **NEGATION**

1. $\dfrac{b \longrightarrow b'}{\sim b \longrightarrow \sim b'}$

2. $\sim t \longrightarrow t' \qquad\qquad (\textit{where } t' = \neg t)$

The *behavior* of Boolean expressions is defined by:

$$\mathrm{eval}(b, \sigma) = t \Leftrightarrow \langle b, \sigma\rangle \longrightarrow^{*} \langle t, \sigma\rangle$$

One can also define *equivalence* of Boolean expressions by:

$$b \equiv b' \quad \Leftrightarrow \quad \forall\sigma.\mathrm{eval}(b, \sigma) = \mathrm{eval}(b', \sigma)$$

## 2.2 Simple Commands

Again we begin with a trivial language of commands,

$$c ::= \ \mathbf{nil} \ \mid \ v := e \ \mid \ c; \ c'$$

and see how the SMC-machine behaves on an example:

$$\begin{aligned}
\langle \varepsilon, abc, \mathrm{z} := \mathrm{x}; (\mathrm{x} := \mathrm{y}; \mathrm{y} := \mathrm{z})\rangle &\longrightarrow \langle \varepsilon, abc, \mathrm{z} := \mathrm{x}\ \mathrm{x} := \mathrm{y}; \mathrm{y} := \mathrm{z}\rangle \\
&\longrightarrow \langle \mathrm{z}, abc, \mathrm{x} := \ \ \mathrm{x} := \mathrm{y}; \mathrm{y} := \mathrm{z}\rangle \\
&\longrightarrow \langle a\ \mathrm{z}, abc, := \ \ \mathrm{x} := \mathrm{y}; \mathrm{y} := \mathrm{z}\rangle \quad (*) \\
&\longrightarrow \langle \varepsilon, aba, \mathrm{x} := \mathrm{y}; \mathrm{y} := \mathrm{z}\rangle \\
&\longrightarrow \langle \varepsilon, aba, \mathrm{x} := \mathrm{y}\ \mathrm{y} := \mathrm{z}\rangle \\
&\longrightarrow^{2} \langle b\ \mathrm{x}, aba, := \ \ \mathrm{y} := \mathrm{z}\rangle \qquad\qquad (*) \\
&\longrightarrow \langle \varepsilon, bba, \mathrm{y} := \mathrm{z}\rangle \\
&\longrightarrow^{2} \langle a\ \mathrm{y}, bba, :=\rangle \qquad\qquad\qquad (*) \\
&\longrightarrow \langle \varepsilon, baa, \varepsilon\rangle
\end{aligned}$$

And we see that of the eleven transitions only three – the assignments – are of interest as system events.

Preferable here would be a sequence of three transitions on configurations of the form $\langle c, \sigma\rangle$, thus:

$$\begin{aligned}
\langle \mathrm{z} \stackrel{\triangledown}{:=} \mathrm{x}; \ (c := \mathrm{y}; \ \mathrm{y} := \mathrm{z}), abd\rangle &\longrightarrow \langle (\mathrm{x} \stackrel{\triangledown}{:=} \mathrm{y}; \ \mathrm{y} := \mathrm{z}), aba\rangle \\
&\longrightarrow \langle \mathrm{y} \stackrel{\triangledown}{:=} \mathrm{z}\ , bba\rangle \\
&\longrightarrow baa
\end{aligned}$$

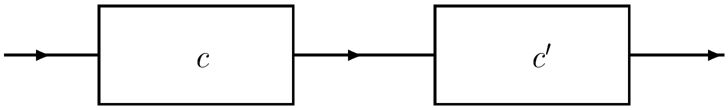
where we have marked the assignments occurring in transitions.

Now informally one can specify such command *executions* as follows:

- **Nil**: To execute **nil** from store $\sigma$ take no action and terminate with $\sigma$ as the final store of the execution.

- **Assignment**: To execute $v := e$ from store $\sigma$ evaluate $e$, and if the result is $m$, change $\sigma$ to $\sigma[m/v]$ (the final store of the execution).
- **Composition**: To execute $c; c'$ form store $\sigma$
  1. Execute $c$ from store $\sigma$ obtaining a final store, $\sigma'$, say, if this execution terminates.
  2. Execute $c'$ from the store $\sigma'$. The final store of this execution is also the final store of the execution of $c; c'$

Sometimes the execution of $c; c'$ is pictured in terms of a little flowchart:




As in the case of expressions one sees that this description is syntax-directed. We formalize it considering terminating executions of a command $c$ from a store $\sigma$ to be transition sequences of the form:

$$\langle c, \sigma \rangle = \langle c_0, \sigma_0 \rangle \longrightarrow \langle c_1, \sigma_1 \rangle \longrightarrow \ldots \longrightarrow \langle c_{n-1}, \sigma_{n-1} \rangle \longrightarrow \sigma_n$$

Here we take the configurations to be:

$$\Gamma = \{\langle c, \sigma \rangle\} \cup \{\sigma\}$$

and the terminal configurations to be

$$T = \{\sigma\}$$

where the transition relation $\langle c, \sigma \rangle \longrightarrow \langle c', \sigma' \rangle$ (resp. $\sigma'$) is read as:

> One step of execution of the command $c$ from the store $\sigma$ results in the store $\sigma'$ and the rest of the execution of $c$ is the execution of $c'$ form $\sigma'$ (resp. and the execution terminates).

Thus we choose $c'$ to represent, in as simple a way as is available, the remainder of the execution of $c$ after its first step. The rules are

- Nil: $\langle \mathbf{nil}, \sigma \rangle \longrightarrow \sigma$
- Assignment:
  1. $$\frac{\langle e, \sigma \rangle \longrightarrow^* \langle m, \sigma \rangle}{\langle v := e, \sigma \rangle \longrightarrow \sigma[m/v]}$$
- Composition]:
  1. $$\frac{\langle c_0, \sigma \rangle \longrightarrow \langle c_0', \sigma' \rangle}{\langle c_0;\ c_1, \sigma \rangle \longrightarrow \langle c_0';\ c_1, \sigma' \rangle}$$

2. $$\frac{\langle c_0, \sigma\rangle \longrightarrow \sigma'}{\langle c_0;\ c_1, \sigma\rangle \longrightarrow \langle c_1, \sigma'\rangle}$$

**Notes 1.**: In formulating the rule for assignment we have considered the *entire evaluation* of the right-hand-side as part of *one* execution step. This corresponds to a change in vies of the size of our step when considering commands, but we could just as well have chosen otherwise.

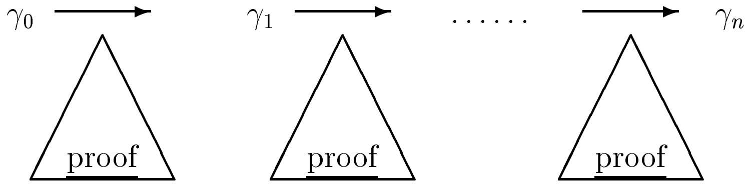
As an example consider the first transition desired above for the execution

$$\langle \mathrm{z} := \mathrm{x}; (\mathrm{x} := \mathrm{y}; \mathrm{y} := \mathrm{z}), abc\rangle$$

It is presented in the top-down way

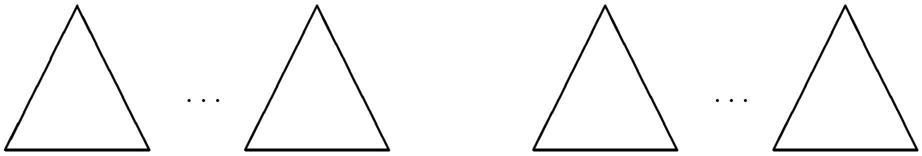
$$\langle \mathrm{z} := \mathrm{z};\ (\mathrm{z} := \mathrm{y};\ \mathrm{y} := \mathrm{z}), abc\rangle \overset{\text{Comp 2}}{\longrightarrow} \langle(\mathrm{z} := \mathrm{y}; \mathrm{y} := \mathrm{z}), aba\rangle\langle \mathrm{z} := \mathrm{z}, abc\rangle \overset{\text{Ass 1}}{\longrightarrow} aba\langle \mathrm{z}, abc\rangle \overset{\text{Var 1}}{\longrightarrow} \langle a, abc\rangle$$

Again we see, as in the case of expressions a "two-dimensional" structure consisting of a "horizontal" transition sequence of the events of system significance and for each transition a "vertical" explanation of why and how it occurs.




For terminating executions of $c_0; c_1$ this will have the form:

$$\frac{< c_0; c_1, \sigma > \overset{c1}{\rightarrow}}{< c_0, \sigma > \rightarrow} \cdots \quad \frac{\overset{c1}{\rightarrow} < c_0^{'\cdot\cdot'}; c_1, \sigma^{'\cdot\cdot'} >}{\rightarrow < c_0^{'\cdot\cdot'}, \sigma^{'\cdot\cdot'} >} \quad \frac{\overset{c2}{\rightarrow} < c_1, \sigma^{'\cdot\cdot''} >}{\rightarrow \sigma^{'\cdot\cdot''}} \cdots \quad \rightarrow \sigma^{'\cdot\cdot''\cdot\cdot'}$$




Again we see that the SMC-machine transition sequences are more-or-less linearisations of these structures. Note the appearance of rules for binary relations (with additional data components) such as:

$$\mathrm{R}(c, c', \sigma, \sigma') \quad \equiv_{def} \quad \langle c, \sigma\rangle \longrightarrow \langle c', \sigma'\rangle$$
$$\mathrm{S}(e, e', \sigma) \quad \equiv_{def} \quad \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle$$

Later we shall make extensive use of predicates to treat the context-sensitive aspects of syntax(= the static aspects of semantics). As far as we can see there is no particular need for *ternary* relations, although the above discussion on the indirectness of our formalization does suggest the possibility of needing relation of *variable* degree for dealing with execution sequences.

## 2.3 L-commands

Recalling the syntax of L-commands,

$$c ::= \textbf{nil} \mid v := e \mid c; c' \mid \textbf{if } b \textbf{ then } c \textbf{ else } c' \mid \textbf{while } b\ \mathbf{c}$$

we see that it remains only to treat conditionals and repetitions.

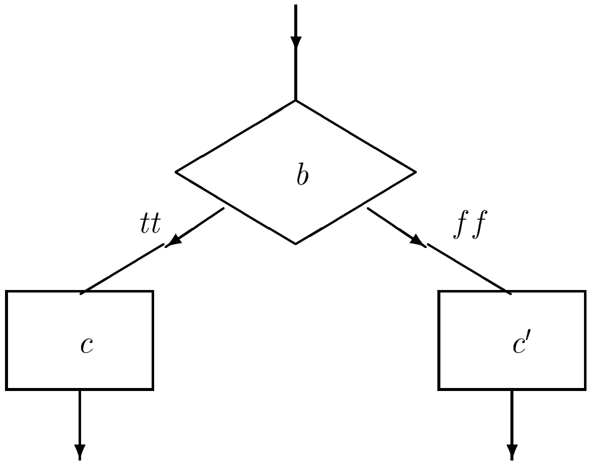
- **Conditionals:** To execute **if** $b$ **then** $c$ **else** $c'$ from $\sigma$

  1. Evaluate $b$ in $\sigma$

  2.1 If result was tt execute $c$ from $\sigma$

  2.2 If result was ff execute $c'$ from $\sigma$

  In pictures we have:




And the rules are:

1. $$\frac{\langle b, \sigma \rangle \longrightarrow^* \langle \mathrm{tt}, \sigma \rangle}{\langle \textbf{if } b \textbf{ then } c \textbf{ else } c', \sigma \rangle \longrightarrow \langle c, \sigma \rangle}$$

2. $$\frac{\langle b, \sigma \rangle \longrightarrow^* \langle \mathrm{ff}, \sigma \rangle}{\langle \textbf{if } b \textbf{ then } c \textbf{ else } c', \sigma \rangle \longrightarrow \langle c', \sigma \rangle}$$

**Notes 1**: Again we are depending on the transition relation of another syntactic class – here Boolean expressions – and a whole computation from that class becomes one step of the computation.

**Notes 2**: No rules for $T$ (**if** $b$ **then** $c$ **else** $c'$) are given as that predicate never applies For a conditional is never terminal as one always has at least one action – namely evaluation the condition.

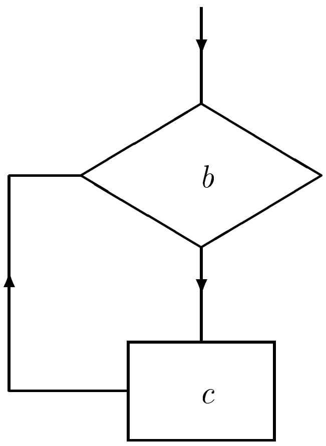
- **While**: To execute **while** $b$ **do** $c$ *from* $\sigma$

1. Evaluate $b$

2.1 If the result is tt, execute $c$ from $\sigma$. If that terminates with final state $\sigma'$, execute **while** $b$ **do** $c$ from $\sigma'$.

2.2 If the result is ff the execute is finished and the final state is $\sigma$.

In pictures we have the familiar flowchart:




The rules are:

$$1. \quad \frac{\langle b, \sigma\rangle \longrightarrow^* \langle \mathrm{tt}, \sigma\rangle}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle \longrightarrow \langle c;\ \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle}$$

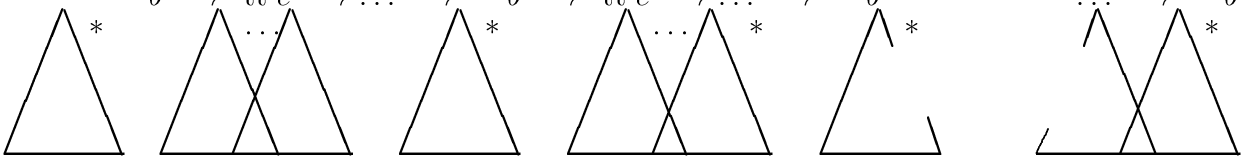
$$2. \quad \frac{\langle b, \sigma\rangle \longrightarrow^* \langle \mathrm{ff}, \sigma\rangle}{\langle \mathbf{while}\ b\ \mathbf{do}\ c\ , \sigma\rangle \longrightarrow \sigma}$$

**Example**: Consider the factorial example $\mathrm{y} := 1; w$ from chapter1, where $w =$ **while** $\sim(\mathrm{x} = 0)$ **do** $c$ where $c = (\mathrm{y} := \mathrm{y} \times \mathrm{x}; \mathrm{x} := \mathrm{x} - 1)$. We start from the state $\langle 3, 5\rangle$.

$$\begin{array}{rlll}
\langle \mathrm{y} := 1; w, \langle 3,5\rangle\rangle & \xrightarrow{\mathrm{ASS1}} & \langle w, \langle 3,1\rangle\rangle\rangle & \\
& \xrightarrow{\mathrm{COMP2}} & \langle c; w, \langle 3,1\rangle\rangle & \text{(via WHI)} \\
& \xrightarrow{\mathrm{COMP1}} & \langle \mathrm{x} := \mathrm{x} - 1; w, \langle 3,3\rangle\rangle & \text{(via COMP2 and ASS1)} \\
& \xrightarrow{\mathrm{COMP2}} & \langle w, \langle 2,3\rangle\rangle & \\
& \xrightarrow{\mathrm{COMP2}} & \langle c; w, \langle p2,3\rangle\rangle & \text{(WHI)} \\
& \xrightarrow{\mathrm{COMP1}} & \langle \mathrm{x} := \mathrm{x} - 1; w, \langle 2,6\rangle\rangle & \text{(via COMP2 and ASS1)} \\
& \xrightarrow{\mathrm{COMP1}} & \langle w, \langle 1,6\rangle\rangle & \\
& \longrightarrow & \langle c; w, \langle 1,6\rangle\rangle & \\
& \longrightarrow & \langle \mathrm{x} := \mathrm{x} - 1; w, \langle 1,6\rangle\rangle & \\
& \longrightarrow & \langle w, \langle 0,6\rangle\rangle & \\
& \xrightarrow{\mathrm{COMP2}} & \langle 0,6\rangle & \text{(via WHI2)}
\end{array}$$

A terminating execution sequence of a while-loop $w = \textbf{while}\ b\,\textbf{do}\ c$ looks like this (omitting $\sigma'$s):

$$\begin{array}{l} w \longrightarrow c; w \longrightarrow \ldots \longrightarrow w \longrightarrow c; w \longrightarrow \ldots \longrightarrow w - - - - - - - \ldots \longrightarrow w \longrightarrow \\ b \longrightarrow^* \mathrm{tt}\ c \longrightarrow \ldots \longrightarrow \cdots b \longrightarrow^* \mathrm{tt}\ c \longrightarrow \ldots \longrightarrow \cdots b - - - - - - - \ldots \longrightarrow \cdots b \longrightarrow^* \mathrm{ff} \end{array}$$




One can now define the behavior and equivalence of commands by:

$$\mathrm{exec}(c, \sigma) = \sigma' \quad \Leftrightarrow \quad \langle c, \sigma \rangle \longrightarrow^* \sigma'$$

and

$$c \equiv c' \quad \Leftrightarrow \quad \forall \sigma . \mathrm{exec}(c, \sigma) = \mathrm{exec}(c', \sigma)$$

## 2.4 Structural Induction

Although we have no particular intention of proving very much either about or with our operational semantics, we would like to introduce enough mathematical apparatus to enable us to establish the truth of such obvious statements as:

$$\text{if } \gamma \notin T \text{ then for some } \gamma' \text{ we have } \gamma \longrightarrow \gamma'$$

The standard tool is the principle of *Structural Induction*(SI). It enables us to prove properties $P(p)$ of syntactic phrases, and it takes on different forms according to the abstract syntax of the language. For $L$ we have three such principles, one for expressions, one for Boolean expressions and one for commands.

### Structural Induction for Expressions

Let $P(e)$ be a property of expressions. Suppose that:

1. For all $m$ in $N$ it is the case that $P(m)$ holds, and

2. For all $v$ in $V$ it is the case that $P(v)$ holds, and

3. For all $e$ and $e'$ in $E$ if $P(e)$ and $P(e')$ holds so does $P(e + e')$, and

4. As 3 but for $-$, and

5. As 3 but for $*$

Then for all expressions $e$, it is the case that $P(e)$ holds.

We take this principle as being intuitively obvious. It can be stated more compactly by using standard logical notation:

$$\begin{aligned}
&[(\forall m \in \mathrm{N}.\ P(m)) \wedge (\forall v \in V.\ P(v))\\
&\quad \wedge (\forall e, e' \in \mathrm{E}.\ P(e) \wedge P(e') \supset P(e + e'))\\
&\quad \wedge (\forall e, e' \in \mathrm{E}.\ P(e) \wedge P(e') \supset P(e - e'))\\
&\quad \wedge (\forall e, e' \in \mathrm{E}.\ P(e) \wedge P(e') \supset P(e * e'))]\\
&\quad \supset \forall e \in \mathrm{E}.\ P(e)
\end{aligned}$$

As an example we prove

**Fact**: The transition relation for expressions is deterministic

**Proof**: We proceed by SI on the property $P(e)$ where

$$P(e) \equiv \forall \sigma, \gamma', \gamma''(\langle e, \sigma\rangle \longrightarrow \gamma' \wedge \langle e, \sigma\rangle \longrightarrow \gamma'') \supset \gamma' = \gamma''$$

Now there are five cases according to the hypotheses necessary to establish the conclusion by SI.

**1.** $e = m \in N$ Suppose $\langle e, \sigma\rangle \longrightarrow \gamma', \gamma''$ . But this cannot be the case as $\langle m, \sigma\rangle$ is stuck. Thus $P(e)$ holds vacuously.

**2.** $e = v \in N$ Suppose $\langle v, \sigma\rangle \longrightarrow \gamma', \gamma''$. Then as there is only one rule for variables, we have $\gamma' = \langle \sigma(v), \sigma\rangle = \gamma''$.

**3.** $e = e_0 + e_1$ Suppose $\langle e_0 + e_1, \sigma\rangle \longrightarrow \gamma', \gamma''$. There are three subcases according to why $\langle e_0 + e_1, \sigma\rangle \longrightarrow \gamma'$

**3.1 Rule 1** For some $e_0'$ we have $\langle e_0, \sigma\rangle \longrightarrow \langle e_0', \sigma\rangle$ and $\gamma' = \langle e_0' + e_1, \sigma\rangle$. Then $e_0$ is not in $N$ (otherwise $\langle e_0, \sigma\rangle$ would be stuck) and so for some $e_0''$ we have $\langle e_0, \sigma\rangle \longrightarrow \langle e_0'', \sigma\rangle$ and so $\gamma'' = \langle e_0'' + e_1, \sigma\rangle$. But by the induction hypothesis applied to $e_0$ we therefore have $e_0' = e_0''$ and so $\gamma' = \gamma''$.

**3.2 Rule 2** We have $e_0 = m \in N$ and for some $e_1'$ we have $\langle e_1, \sigma\rangle \longrightarrow \langle e_1'', \sigma\rangle$ and $\gamma'' = \langle m + e_1'', \sigma\rangle$. But applying the induction hypothesis to $e_1$, we see that $e_1' = e_1''$ and so $\gamma' = \gamma''$.

**3.3 Rule 3** We have $e_0 = m_0, e_1 = m_1$. Then clearly $\gamma' = \gamma''$.

**4.** $e = e_0 - e_1$

**5.** $e = e_0 * e_1$ These cases are similar to the third case and are left to the reader.■

In the above we did not need such a strong induction hypothesis. Instead we could choose a fixed $\sigma$ and proceed by SI on $Q(e)$ where:

$$Q(e) \equiv \forall \gamma', \gamma''. (\langle e, \sigma \rangle \longrightarrow \gamma \wedge \langle e, \sigma \rangle \longrightarrow \gamma'') \supset \gamma' = \gamma''$$

However, this is just a matter of luck (here that the evaluation of expressions does not side effect the state). Generally it is wise to choose one's induction hypothesis as strong as possible. The point is that if one's hypothesis has the form (for example)

$$P(e) \equiv \forall \sigma \, Q(e, \sigma)$$

then when proving $P(e_0 + e_1)$ given $P(e_0$ and $P(e_1)$ one fixes $\sigma$ and tries to prove $Q(e, \sigma)$. But in this proof one is at liberty to use the facts $Q(e_0, \sigma)$, $Q(e_0, \sigma'), Q(e_1, \sigma), Q(e_1, \sigma'')$ for *any* $\sigma'$ and $\sigma''$.

## SI for Boolean Expressions

We just write down the symbolic version for a desired property $P(b)$ of Boolean expressions.

$$\begin{aligned}
&[(\forall t \in \mathrm{T}.\ P(t)) \wedge (\forall e, e' \in E.\ P(e = e')) \\
&\quad \wedge (\forall b, b' \in \mathrm{B}.(b) \wedge P(b') \supset P(b \ \mathbf{or}\ b')) \\
&\quad \wedge (\forall b \in \mathrm{B}.\ P(b) \supset P(\sim b))] \\
&\quad \supset \forall b \in B. P(b)
\end{aligned}$$

In general when applying this principle one may need further structural inductions on expressions. For example:

**Fact**: If $b$ is not in $T$ and contains no occurrence of an expression of the form $m - n)$ where $m < n$, then no $\langle b, \sigma \rangle$ is stuck.

**Proof**: We fix $\sigma$ and proceed by SI on Boolean expressions on the property:

$$\begin{aligned}
Q(b) \equiv\ &[b \notin \mathrm{T} \wedge (\forall m < n.(m - n') \text{ does not occur in } b)] \\
&\supset \langle e, \sigma \rangle \text{ is not stuck}
\end{aligned}$$

**Case 1** $B = tt$ This holds vacuously.

**Case 2** $B = (e = e')$ Here there are three subcases depending on the forms of $e$ and $e'$.

**Case 2.1** If $e$ is not in N, then for some $e''$ we have $\langle e, \sigma \rangle \longrightarrow \langle e'', \sigma \rangle$

**Lemma** For any expression $e$ not in $N$ if $e$ has no subexpressions of the form, $m - n$, where $m < n$, then no $\langle e, \sigma \rangle$ is stuck.

**Proof** By SI on expressions and left to the reader.⌧

Continuing with case 2.1 we see that $\langle e = e', \sigma \rangle \longrightarrow \langle e'' = e', \sigma \rangle$ so $\langle b, \sigma \rangle$ is not stuck.

**Case 2.2** Here $e$ is in $N$ but $e'$ is not; the proof is much like case 2.1 and also uses the lemma.

**Case 2.3** Here $e$, $e'$ are in $N$ and we an use rule **EQU. 3**

**Case 3** $B = (b_0 \text{ or } b_1)$ This is like case 3 of the proof of fact 1.

**Case 4** $B = \sim b_0$ If $b_0$ is not in $T$ we can easily apply the induction hypothesis. Otherwise use rule Neg.2

This concludes all the cases and hence the proof.■

## SI for Commands

We just write down the symbolic version for a (desired) property $P(c)$ of commands:

$$
\begin{aligned}
[P(\mathbf{nil}) \wedge \forall v \in \mathrm{V},\ e \in \mathrm{E}.\ P(v := e)& \\
\wedge\ (\forall c, c' \in \mathrm{C}.P(c) \wedge P(c') \supset P(c; c'))& \\
\wedge\ (\forall b \in \mathrm{B}.\forall c, c' \in C.P(c) \wedge P(c') \supset P(\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c'))& \\
\wedge\ (\forall b \in \mathrm{B}.\ \forall c \in C.P(c) \supset P(\mathbf{while}\ b\ \mathbf{do}\ c))& \\
\supset \forall c \in \mathrm{C}.\ P(c)&
\end{aligned}
$$

For an example we prove:

**Fact**: If $v$ does not occur on the left-hand-side of an assignment in in $c$, then the execution of $c$ cannot affect its value. That is if $\langle c, \sigma\rangle \longrightarrow^* \sigma'$ then $\sigma(v) = \sigma'(v)$.

**Proof**: By SI on commands. The statement of the hypothesis should be apparent from the proof, and is left to the reader.

**Case 1** $n = \mathbf{nil}$ Clear.

**Case 2** $c = (v := e)'$ Here $v' \neq v$ and we just use the definition of $\sigma[m/v']$.

**Case 3** $c = (c_0;\ c_1)$ Here if $\langle c_0; c_1, \sigma\rangle \longrightarrow^* \sigma'$ then for some $\sigma''$ we have $\langle c_0, \sigma\rangle \longrightarrow^* \sigma''$ and $\langle c_1, \sigma''\rangle \longrightarrow^* \sigma'$. (This requires a lemma for proof by the reader).
Then by the induction hypothesis applied first to $c_1$ and then to $c_0$ we have:

$$\sigma'(v) = \sigma''(v) = \sigma(v)$$

**Case 4** $c = \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$ Here we easily use the induction hypothesis on $c_0$ and $c_1$ (according to the outcome of the evaluation of $b$).

**Case 5** $c = \mathbf{while}\ b\ \mathbf{do}\ c'$ Here we argue on the *length* of the transition sequence $\langle c, \sigma\rangle \longrightarrow \ldots \longrightarrow \sigma'$. This is just an ordinary mathematical induction. In case the sequence has length 0, we have $\sigma' = \sigma$. Otherwise there are two cases according to the result of evaluating $b$. We just look at the harder one.

**Case 5.1** $\langle c, \sigma \rangle \longrightarrow \langle c'; c, \sigma \rangle \longrightarrow \ldots \longrightarrow \sigma_1$. Here we see that $\langle c', \sigma \rangle \longrightarrow^* \sigma_2$ (and apply the main SI hypothesis) and also that $\langle c, \sigma_2 \rangle \longrightarrow^* \sigma_1$ and a shorter transition sequence to which the integer induction hypothesis can there fore be applied. ■

This particular lemma shows that on occasion we will use other induction principles such as induction on the length of a derivation sequence.

Another possibility is to use induction on some measure of the *size* of the proof of an assertion $\gamma \longrightarrow \gamma'$ (which would, strictly speaking, require a careful definition of the size measure). Anyway we repeat that we will not develop too much "technology" for making these proofs, but would like the reader to be able, in principle, to check out simple facts.

## 2.5 Dynamic Errors

In the definition of the operational semantics of L-expressions we allowed configurations of the kind $\langle (5+7) * (10-16), \sigma \rangle$ to stick. Thus, although we did ensure:

$$\gamma \in T \supset \neg \exists \gamma'. \gamma \longrightarrow \gamma'$$

we did not ensure the converse. Implementations of real programming languages will ensure the converse generally by issuing a run-time ( = dynamic) error report and forcibly terminating the computation. It would therefore be pleasant if we could also specify dynamic errors.

As a first approximation we add an *error* configuration to the possible configurations of each of the syntactic classes of L. Then we add some *error* rules.

- Expressions
  - Sum

    4. $$\frac{\langle e_0, \sigma \rangle \longrightarrow \mathbf{error}}{\langle e_0 + e_1, \sigma \rangle \longrightarrow \mathbf{error}}$$

    5. $$\frac{\langle e_1, \sigma \rangle \longrightarrow \mathbf{error}}{\langle m + e_1, \sigma \rangle \longrightarrow \mathbf{error}}$$
  - Minus

    4,5 as for Sum

    6. $\langle m - m', \sigma \rangle \longrightarrow \mathbf{error}$ (if $m < m'$)
  - Times

    4,5 as for Sum
- Boolean Expressions
  - Disjunction

    4,5 as for Sum
  - Equality

4,5 as for Sum

– Negation

3. $$\frac{\langle e, \sigma\rangle \longrightarrow \textbf{error}}{\langle \sim e, \sigma\rangle \longrightarrow \textbf{error}}$$

- Commands

  – Assignment

  2. $$\frac{\langle e, \sigma\rangle \longrightarrow \textbf{error}}{\langle v := e, \sigma\rangle \longrightarrow \textbf{error}}$$

  – Composition

  3. $$\frac{\langle c_0, \sigma\rangle \longrightarrow \textbf{error}}{\langle c_0; c_1, \sigma\rangle \longrightarrow \textbf{error}}$$

  – Conditional

  3. $$\frac{\langle b, \sigma\rangle \longrightarrow^* \textbf{error}}{\langle \textbf{if } b\textbf{then } c \textbf{ else} c', \sigma\rangle \longrightarrow \textbf{error}}$$

  – Repetition

  3. $$\frac{\langle b, \sigma\rangle \longrightarrow^* \textbf{error}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \longrightarrow \textbf{error}}$$

So the *only* possibility of dynamic errors in L arises form the subtraction of greater form a smaller. Of course other languages can provide many other kinds of dynamic errors: Division by zero, overflow, taking the square root of a negative number, failing dynamic type-checking tests, overstepping array bounds, missing a dangling reference or reaching an uninitialized location etc. etc. But the above simple example does at least indicate a possibility.

**Fact**: No L-configuration sticks (with the above rules added).

**Proof**: Left to the reader as an exercise. ∎

## 2.6 Simple Type-Checking

We consider a variant, L', of L in which expressions and Boolean expressions are amalgamated into one syntactic class and have to be sorted out again by type-checking. Here is the language L'.

- Basic Syntactic Sets

  – **Truthvalues**: $t \in \mathrm{T}$

  – **Numbers**: $m, n \in \mathrm{T}$

  – **Variables**: $v \in \mathrm{Var} = \{a, b, x, x', x_1, x_2, \dots\}$

- **Binary Operations**: $bop \in \mathrm{BOp} = \{+, -, *, =, \mathbf{or}\}$

- Derived Syntactic Sets
  - **Expressions**: $e \in \mathrm{Exp}$ where:

    $$e ::= m \mid t \mid v \mid e_0 \; bop \; e_1 \mid {\sim}e$$

  - **Commands**: $c \in Com$ where:

    $$c ::= \mathbf{nil} \mid v := e \mid c_0;\ c_1 \mid \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ e\ \mathbf{do}\ c$$

**Note**: We have taken Var to be infinite in the above in order to raise a little problem (later) on how to avoid infinite memories.

Many expressions such as (tt + 5) or ∼6 now have no sense to them, and nor do such commands as **if** x **or** 5 **then** $c_0$ **else** $c_1$. To make sense an expression must have a type, and in L' there are exactly two possibilities:

- **Types**: $\tau \in \mathrm{Types} = \{\mathrm{int}, \mathrm{bool}\}$

To see which expression have types and what they are we will just give some rules for assertions:

$$e : \tau \quad \equiv \quad e \text{ has type } \tau$$

Note first that the basic syntactic sets have, in a natural way, associated type information. Clearly we will have truthvalues having type bool., numbers having type int, variables having type int and for each binary operation, bop, we have a *partial binary function* on Types:

| +, −, ∗ | bool | int |
|---|---|---|
| bool | ? | ? |
| int | ? | int |

| = | bool | int |
|---|---|---|
| bool | ? | ? |
| int | ? | bool |

| **or** | bool | int |
|---|---|---|
| bool | bool | ? |
| int | ? | ? |

- Rules

**Truthvalues:** $t : \mathrm{bool}$

**Numbers:** $m : \mathrm{int}$

**Variables:** $v : \mathrm{int}$

**BinaryOperations:** $$\frac{e_0 : \tau_0 \quad e_1 : \tau_1}{e_0 \text{ bop } e_1 : \tau_2} \qquad (\text{where } \tau_2 = \tau_{\mathrm{bop}}(\tau_0, \tau_1))$$

**Negation:** $$\frac{e : \mathrm{bool}}{{\sim}e : \mathrm{bool}}$$

Now for commands we need to sort out those commands which are *well-formed* in the sense that all subexpressions have a type and are Boolean when they ought to be. The rules for commands involve assertions:

$$\mathrm{Wfc}(c) \equiv c \text{ is a well-formed command.}$$

**Nil:** $\mathrm{Wfc}(\mathbf{nil})$

**Assignment:** $$\frac{e : \mathrm{int}}{\mathrm{Wfc}(v := e)}$$

**Sequencing:** $$\frac{\mathrm{Wfc}(c_0)\ \ \mathrm{Wfc}(c_1)}{\mathrm{Wfc}(c_0; c_1)}$$

**Conditional:** $$\frac{e : \mathrm{bool}\ \ \mathrm{Wfc}(c_0)\ \ \mathrm{Wfc}(c_1)}{\mathrm{Wfc}\ (\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1)}$$

**While:** $$\frac{e : \mathrm{bool}\ \ \mathrm{Wfc}(c)}{\mathrm{Wfc}\ (\mathbf{while}\ e\ \mathbf{do}\ c)}$$

Of course all of this is really quite trivial and one could have separated out the Boolean expressions very easily in the first place, as was done with L. However, we will see that the method generalizes to the *context-sensitive* aspects, also referred to in the literature as the *static semantics.*

Turning to the dynamic semantics we want now to avoid configurations $\langle c, \sigma\rangle$ with $\sigma : \mathrm{Var} \longrightarrow \mathrm{N}$, as such stores are infinite objects. For we have more or less explicitly indicated that we are doing (hopefully nice) finitary mathematics. The problem is easily overcome by noting that we only need $\sigma$ to give values for all Consequently for any *finite* subset $V$ of Var we set:

$$\mathrm{Stores}_V = V \longrightarrow \mathrm{N}$$

and take the configurations also to be indexed by $V$

$$\Gamma_{E,V} = \{\langle e, \sigma\rangle \mid \exists\tau, e : \tau, \mathrm{Var}(e) \subseteq V,\ \sigma \in \mathrm{Stores}_V\}$$
$$\Gamma_{C,V} = \{\langle c, \sigma\rangle \mid \mathrm{Wfc}(c), \mathrm{Var}(c) \subseteq V, \sigma \in \mathrm{Stores}_V\}$$

where $\mathrm{Var}(e)$ is the set of variables occurring in $e$. The rules are much the same as before, formally speaking. That is they are the same as before but with the variables and metavariables ranging over the appropriate sets and an added index. So for example in the rule

**Comp 2** $$\frac{N\langle c_0, \sigma\rangle \longrightarrow_V \sigma'}{\langle c_0;\ c_1, \sigma\rangle \longrightarrow_V \langle c_1, \sigma'\rangle}$$

It is meant that $c_0, c_1$ (and hence $c_0; c_1$) are well formed commands with their variables all in $V$ all of the configurations mentioned in the rule are in $\Gamma_{C,V}$.

Equally in the rule

**Sum 1** $$\frac{\langle e_0, \sigma\rangle \longrightarrow_V \langle e_0', \sigma\rangle}{\langle e_0 + e_1, \sigma\rangle \longrightarrow_V \langle e_0' + e_1, \sigma\rangle}$$

It is meant that all the expressions $e_0, e_0', e_0 + e_1, e_0' + e_1$ have a type (which must here be int) and all their variables are in $V$ and all the configurations mentioned in the rule are in

$\Gamma_{E,V}$. Thus the rules define families of transition relations, $\longrightarrow_V \subseteq \Gamma_{E,V} \times \Gamma_{E,V}$ for expressions, $\longrightarrow_V \subseteq \Gamma_{C,V} \times \Gamma_{C,V}$ for commands.

In the above we have taken the definition of $\mathrm{Var}(e)$, the variables occurring in $e$ and also of $\mathrm{Var}(c)$ for granted as it is rather obvious what is meant. However, it is easily given by a so-called *definition by structural induction.*

$$\begin{array}{ll}
\mathrm{Var}(t) & = \mathrm{Var}(m) = \emptyset \\
\mathrm{Var}(v) & = \{v\} \\
\mathrm{Var}(e_0 \; bop \; e_1 & = \mathrm{Var}(e_0) \cup \mathrm{Var}(e_1) \\
\mathrm{Var}(\sim e) & = \mathrm{Var}(e)
\end{array}$$

With this kind of syntax-directed definition what is meant is that it can easily be shown by SI that the above equations ensure that for any $e$ there is only one $V$ with $\mathrm{Var}(e) = V$. The definition for commands is similar and is left to the reader, the only point of (very slight) interest is the definition of $\mathrm{Var}(v := e)$.

The definition can also be cast in the form of rules for assertions of the form $\mathrm{Var}(t) = V$.

**Truthvalues:** $\mathrm{Var}(t) = \emptyset$

**Numbers:** $\mathrm{Var}(m) = \emptyset$

**Variables** $\mathrm{Var}(v) = \{v\}$

**Binary Operations:** $$\frac{\mathrm{Var}(e_0) = V_0 \quad \mathrm{Var}(e_1) = V_1}{\mathrm{Var}(e_0 \; bop \; e_1) = V_0 \cup V_1}$$

**Negation:** $$\frac{\mathrm{Var}(e) = V}{\mathrm{Var}(\sim e) = V}$$

**Exercise**: Give rules for the assertion $\mathrm{Var}(e) \subseteq V$.

Finally we have a parametrical form of behavior. For example for commands we have a partial function:

$$\mathrm{Exec} : C_V \times \mathrm{Stores}_V \longrightarrow \mathrm{Stores}_V$$

where $C_V = \{c \in C \mid \mathrm{Wfc}(c) \wedge \mathrm{Var}(c) \subseteq V\}$, given by:

$$\mathrm{Exec}(c, \sigma) = \sigma' \quad \equiv \quad \langle c, \sigma \rangle \longrightarrow^* \sigma'$$

## 2.7 Static Errors

The point here is to specify failures in the type-checking mechanism. Here are some rules for a very crude specification where one just adds a new predicate Error.

- Binary Operations

  1. $$\frac{\mathrm{Error}(e_0)}{\mathrm{Error}(e_0 \; bop \; e_1)}$$

2. $\dfrac{\text{Error}(e_1)}{\text{Error}(e_0\ bop\ e_1)}$

3. $\dfrac{e_0 : \tau_0 \quad e_1 : \tau_1}{\text{Error}(e_0\ bop\ e_1)}$ (If $\tau_{bop}(\tau_0, \tau_1)$ is undefined)

- Negation

$$\frac{\text{Error}(e)}{\text{Error}(\sim e)}$$

- Assignment

1. $\dfrac{\text{Error}(e)}{\text{Error}(v := e)}$

2. $\dfrac{e : bool}{\text{Error}(v := e)}$

- Sequencing

1. $\dfrac{\text{Error}(c_0)}{\text{Error}(c_0;\ c_1)}$

2. $\dfrac{\text{Error}(c_1)}{\text{Error}(c_0;\ c_1)}$

- Conditional

1. $\dfrac{\text{Error}(e)}{\text{Error}(\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1)}$

2. $\dfrac{\text{Error}(c_0)}{\text{Error}(\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1)}$

3. $\dfrac{\text{Error}(c_1)}{\text{Error}(\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1)}$

4. $\dfrac{e : int}{\text{Error}(\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1)}$

- While

1. $\dfrac{\text{Error}(e)}{\text{Error}(\mathbf{while}\ e\ \mathbf{do}\ c)}$

2. $\dfrac{\text{Error}(c)}{\text{Error}(\mathbf{while}\ e\ \mathbf{do}\ c)}$

3. $\dfrac{e : int}{\text{Error}(\mathbf{while}\ e\ \mathbf{do}\ c)}$

## 2.8 Exercises

### Expressions

**1.** Try out a few example evaluations.

**2.** Write down rules for the right-to-left evaluation of expressions, as opposed to the left-to-right evaluation described above.

**3.** Write down rules for the *parallel* evaluation of expressions, so that the following kind of transition sequence is possible:

$$(1 + (2 + 3)) + ((4 + 5) + 6) \longrightarrow (1 + (2 + 3)) + (9 + 6) \longrightarrow (1 + 5) + (9 + 6)$$
$$\longrightarrow 6 + (9 + 6) \longrightarrow 6 + 15 \longrightarrow 21$$

Here one transition is one action of imaginary processors situated just above the leaves of the expressions (considered as a tree).

**4.** Note that in the rules if $\langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle$ then $\sigma' = \sigma$. This is the mathematical counterpart of the fact that evaluation of L-expressions produces no side-effects. Rephrase the rules for L-expressions in terms of relations $\sigma \vdash e \longrightarrow e'$ where $\sigma \vdash e \longrightarrow e' \;\; \equiv_{\text{def}} \langle e, \sigma\rangle \longrightarrow \langle e', \sigma\rangle$ and can be read as "given $\sigma$, $e$ reduces to $e'$".

**5.** Give rules for "genuine" parallel evaluation where one or more processors as imagined in 3 can perform an action during the same transition [**Hint** – Use the idea of exercise 4].

**6.****Try to develop a method of axiomatizing entire derivation sequences. Can you find any advantages for this idea?

### Boolean Expressions

**7.** Can you find various kinds of rules analogous to these for *or* for conjunctions $b$ **and** $b'$? (By the way, the left-sequential construct is often advantageous to avoid array subscripts going out of range as in:

$$\begin{array}{ll} \textbf{while} & (i <= n) \textbf{ and } a[i] <> x \\ \textbf{do} & i := i + 3;\; c \quad ). \end{array}$$

**8.** Treat the following additions to the syntax

$$e ::= \textbf{if } b \textbf{ then } e_0 \textbf{ else } e_1$$
$$b ::= \textbf{if } b_0 \textbf{ then } b_1 \textbf{ else } b_2$$

Presumably you will have given rules for the usual sequential conditional. Can you find and give rules for a parallel conditional analogous to parallel disjunction?

**9.** Treat the following additions to the syntax which introduce the possibilities of side-effects in the evaluation of expressions:

$$e ::= \textbf{begin } c \textbf{ result } e$$

(meaning: execute $c$ then evaluate $e$) and the assignment expression:

$$e ::= (v := e)$$

where the intention is that the value of $(v := e)$ is the value of $e$ but the assignment also occurs, producing a side-effect in general.

**10.** Show that the equivalence relations on expressions and boolean expressions are *respected* by the program constructs discussed above so that for example:

$$\begin{array}{lll}
\text{a) } e_0 \equiv e_0; \wedge e_1 \equiv e_1' & \supset & (e_0 + e_1) \equiv (e_0' + e_1') \\
\text{b) } e_0 \equiv e_0; \wedge e_1 \equiv e_1' & \supset & (e_0 - e_1) \equiv (e_0' - e_1') \\
\text{c) } e_0 \equiv e_0; \wedge e_1 \equiv e_1' & \supset & (e_0 = e_1) \equiv (e_0' = e_1') \\
\text{d) } b \equiv b' & \supset & \sim b \equiv \sim b'
\end{array}$$

## Commands

**11.** Give a semantics for the "desk calculator" command

$$v+ := e$$

so that the equivalence

$$(v+ := e) \quad \equiv \quad (v := v + e)$$

holds (and you can prove it!)

**12.** Give a semantics for the ALGOL-60 assignment command

$$v_1 := (v_2 := \ldots (v_n := e) \ldots)$$

so that (see exercise 9) the equivalence

$$(v_1 := (v_2 := \ldots (v_n := e) \ldots)) \quad \equiv \quad (v_1 := e)$$

where $e = (v_2 := \ldots (v_n := e) \ldots)$ holds, and you can prove it.

**13.** Treat the simultaneous assignment

$$v_1 := e_1 \textbf{ and } \ldots \textbf{ and } v_n := e_n$$

where the $v_1$ must all be different. Execution of this command consists of first evaluating all the expressions and then performing the assignments.

**14.** Treat the following variations on the conditional command:

$$\begin{array}{l} \textbf{if } b \textbf{ then } c \ \mid \ \textbf{unless} b \textbf{ then } c \ \mid \\ \qquad\qquad \textbf{if } b_1 \textbf{ then} c_1 \\ \qquad\qquad \textbf{elseif } b_2 \textbf{ then } c_2 \\ \qquad\qquad \vdots \\ \qquad\qquad \textbf{elseif } b_n \textbf{ then } c_n \\ \qquad\qquad \textbf{else } c_{n+1} \end{array}$$

and show they can all be eliminated (to within equivalence) in favor of the ordinary conditional.

**15.** Treat the simple iteration command

$$\textbf{do } e \textbf{ times } c$$

and the following variations on repetitive commands like **while** $b$ **do** $c$:

$$\begin{array}{l} \textbf{repeat } c \textbf{ until } b \ \mid \ \textbf{until } b \textbf{ repeat } c \ \mid \ \textbf{repeat } c \textbf{ unless } b \ \mid \\ \qquad \textbf{loop} \\ \qquad c_1 \\ \qquad \textbf{when } b_1 \textbf{ do } c'_1 \textbf{ exit} \\ \qquad c_2 \\ \qquad \vdots \\ \qquad \textbf{when } b_n \textbf{ do } c'_n \textbf{ exit} \\ \qquad c_{n+1} \\ \qquad \textbf{repeat} \end{array}$$

where the last construct has $n$ possible exits from the loop.

**16.** Show that the equivalence is respected by the above consturucts on commands so that, for example

a) $e \equiv e' \quad \supset \quad (v := e) \equiv (v := e')$
b) $c_0 \equiv c'_0 \wedge c_1 \equiv c'_1 \quad \supset \quad c_0;\ c_1 \equiv c'_0;\ c'_1$
c) $b \equiv b' \wedge c_0 \equiv c'_0 \wedge c_1 \equiv c'_1 \quad \supset \quad \textbf{if } b \textbf{ then } c_0 \textbf{ else} c_1 \equiv \textbf{if } b' \textbf{ then } c'_0 \textbf{ else } c'_1$
d) $b \equiv b' \wedge c \equiv c' \quad \supset \quad \textbf{while } b \textbf{ do } c \equiv \textbf{while } b' \textbf{ do } c'$

**17.** Redefine behavior and equivalence to take account of run-time errors. Do the statements of exercise 16 remain valid?

**18.****Try time and space complexity in the present setting. [**Hint** – Consider configurations of the form, say, $\langle c, \sigma, t, s \rangle$ where

$t =$ "the total time used so far"

$s =$ "the maximum space used so far"]

There is lots to do. Try finding fairly general definitions, define behavior and equivalence (approximate equivalence?) and see which program equivalences preserve equivalence. Try looking at measures for the parallel evaluation of expressions. Try to see what is reasonable to incorporate from complexity literature. Can you use the benefits of our structured languages to make standard simulation results easier/nicer for students?

**19.****Try exercise 23 and 24 from chapter 1 again.

**20.** Give an operational semantics for L, but where only 1 step of the evaluation of an expression or Boolean expression is needed for 1 step of execution of a command. Which of the two possibilities – the "big steps" of the main text or the "little steps" of the exercise – do you prefer and why?

## Proof

**21.** Let $c$ be any command not involving subexpressions of the form $(e - e')$ or *while* loops but allowing the simple iteration command of exercise 15. Show that any execution sequence $\langle c, \sigma \rangle \longrightarrow^* \ldots$ terminates.

**22.** Establish (for L) the following "arithmetic" equivalences:

$$\begin{array}{lcl} e_0 + 0 & \equiv & e_0 \\ e_0 + e_1 & \equiv & e_1 + e_0 \\ e_0 + (e_1 + e_2) & \equiv & (e_0 + e_1) + e_2 \\ \text{etc} & & \end{array}$$

Which ones fail if side-effects are allowed in expressions?

Establish the equivalences:

a) $\textbf{if } b \textbf{ then } c \textbf{ else } c \quad \equiv \quad c$

b) $\textbf{if } b \textbf{ then if } b \textbf{ then } c_0 \textbf{ else } c' \quad \equiv \quad \textbf{if } b \textbf{ then } c_0 \textbf{ else } c'_1$
$\qquad \textbf{else if } b \textbf{ then } c_1 \textbf{else} c'_1$

c) $\textbf{if } b \textbf{ then if } b' \textbf{ then } c_0 \textbf{ else } c_1 \quad \equiv \quad \textbf{if } b' \textbf{ then if } b \textbf{ then } c_0 \textbf{ else } c_2$
$\qquad \textbf{else if } b' \textbf{ then } c_2 \textbf{ else } c_3 \qquad\qquad \textbf{else if } b \textbf{ then } c_1 \textbf{ else } c_3$

Which ones remain true if Boolean expressions have side-effects/need not terminate?

**23.** Establish or refute each of the following suggested equivalences for the language L (and slight extensions, as indicated):

a) $\textbf{nil}; c \quad \equiv \quad c \quad \equiv \quad c; \textbf{ nil}$
b) $c; \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \quad \equiv \quad \textbf{if begin } c \textbf{ result } b \textbf{ then } c_0 \textbf{ else } c_1$
c) $(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1); c \quad \equiv \quad \textbf{if } b \textbf{ then } c_0; c \textbf{ else } c_1; c$
d) $\textbf{while } b \textbf{ do } c \quad \equiv \quad \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else nil}$
e) $\textbf{repeat } c \textbf{ until } b \quad \equiv \quad c; \textbf{while } {\sim} b \textbf{ do } c$

### Type Checking

**24.** Make L' a little more realistic by adding a type real, decimals, variables of all three types, and a variety of operators. Allow int to real conversion, but not vice-versa.

**25.** Show that if $\langle c, \sigma \rangle \longrightarrow \langle c', \sigma' \rangle$ and $x \in Dom(\sigma) \backslash Var(c)$ then $\sigma(x) = \sigma'(x)$.

**26.** Show that if $\langle c, \sigma \rangle \longrightarrow \langle c', \sigma' \rangle$ is a transition within $\Gamma_{C,V}$ and $\langle c, \overline{\sigma} \rangle \longrightarrow \langle c', \overline{\sigma'} \rangle$ is a transition within $\Gamma_{C,V'}$ where $V \subseteq V'$ then, if $\sigma = \overline{\sigma} \lceil V$, it follows that $\sigma' = \overline{\sigma'} \lceil V$.

**27.** The static error specification is far too cured. Instead one should have a set $M$ of *messages* and a relation:

$$\text{Error}(e, m) \equiv m \text{ is a report on an error in } e$$

and similarly for commands. Design a suitable $M$ and a specification of Error for $L'$. Try to develop a philosophy of what a nice error message should be. See [Hor] for some ideas.

**28.** How would you treat dynamic type-checking in L'? What would be the new ideas for error messages (presumably one adds an $M$ (see ex. 27) to the configurations).

## 2.9 BIBLIOGRAPHICAL REMARKS

The idea of reduction sequences originates in the $\lambda$-calculus [Hin] as does the present method of specifying steps axiomatically where I was motivated by Barendregt's thesis [Bar1]. I applied the idea to $\lambda$-calculus-like programming languages in [Plo1], [Plo2] and Milner saw how to extend it to simple imperative languages in [Mil1]. More recently the idea has been applied to languages for concurrency and distributed systems [Hen1], [Mil2], [Hen2]. The present course is a systematic attempt to apply the idea as generally as possible. A good deal of progress has been made on other aspects of reduction and the *lambda*-calculus, a partial survey and further references can be found in [Ber] and see [Bar2].

The Butch school has originated related ideas [deB], [deR]. A direct precursor of our method can be found in the work by Lauer and Hoaare, [Hoa] configurations which have the rough form $\langle s_1, \ldots, s_n, \sigma \rangle$ where the $s_1$ are statements (includes commands). They define a next-configuration function and the definition is to some extent syntax-directed. The idea of a syntax-directed approach was independently conceived and mentioned all too briefly in the work of Salwicki [Sal].

Somewhat more distantly various grammatical (= symbol-pushing too) approaches have been tried. For example W-grammars [Cle] and attribute grammars [Mad]; although these definitions are not syntax-directed definitions of single transitions it should be perfectly possible to use the formalisms to write definitions which are. The question is rather how appropriate the formalisms would be with regard to such issues as completeness, clarity (= readability), naturalness, realism, modularity (= modifiability + extensionality). One good discussion of some of these issues can be found in [Mar]. For concern with modularity consult the course notes of Petr Mosses. Our method is clearly intended to be complete and natural and realistic, and we try to be clear; the only point is that it is quite informal, begin normal finite mathematics. There must be many questions on good choices of formalism. As regards modularity we just hope that if we get the other things in a reasonable state, then current ideas for imposing modularity on specifications will prove useful.

For examples of good syntax-directed English specifications consult the excellent article by Ledgard on ten mini-languages [Led]. These languages will provide you with mini-projects which you should find very user full in understanding the course, and which could very well be the basic for more extended projects. For a much more extended example see the Algol 68 Report [Wij]. Structural induction seems to have been introduced to Computer Science by Burstall in [Bur]; for a system which performs automatic proofs by Structural induction on lists see [Boy]. For discussions of what error messages should be see [Hor] and for remarks on how and whether to specify them see [Mar].

## 2.10 Bibliography

[Bar1] Barendregt, H. (1971) *Some Extensional Term Models For Combinatory Logic and Lambda-Calculi*

[Bar2] Barendregt, H. (1981) *The Lambda Calculus.* Studies in Logic Vol. 103, North-Holland

[Ber] Berry, G. and Lévy, j-j. *A Survey of Some Syntactic Results in the $\lambda$-calculus.* MFCS'79 ed. J. Becvar LNCS Vol. 74, pp. 552-566

[Boy] Boyer, R.S. and Moore, J.S. (1979) *A Computational Logic* Academic Press

[Bur] Burstall, R.M.B. (1969) *Proving Properties of Programs by Structural Induction* Comput. J. 12(1) pp. 41-48

[Cle] Cleaveldan, J.C. and Uggalis, R.C. (1977) *Grammars for Programming Languages* Elsevier

[deB] de Bakker

[deR] de Roever

[Hen1] Hennessy , M.C.B. and Plotkin, G.D.(1979) *Full Abstraction for a Simple Parallel Programming Language* MFCS'79 Olomom. LNCS Vol. 74, pp. 108-120

[Hen2] Hennessy , M.C.B., Li, Weiand Plotkin, G.D. (1981) *A First Attempt at Translating CSP into CCS.* Proceedings of the Second international Conference on Distributed Computing Systems To appear

[Hin] Hindley, J.R., Lercher, B. and Seldin, J.P. (1972) *Introduction to Combinatory Logic* Cambiridge University Press

[Hoa] Hoare, C.A.R. and Luer, P.G. (1973) *Consistent and Complete Formal Theories of the Semantics of Programming Languages* Acta Infomatica

[Hor] Horning, J.J. (1974) *What the Compiler Should Tell The User. Compiler Construction, an Advanced Course* eds FL. Bauer and J. Eickel LNCS 21 pp. 525-548

[Lau] Luer, P.G. (1971) *Consistent Formal Theories of The Semantics of Programming Languages* IBM Laboratories Vienna TR 25.121

[Led] Ledgard, H.F. (1971) *Ten Mini-Languages. A Study of Topical issues in Programming Languages* Comp. Surveys 3, pp. 115-146

[Mad] Madsen, O.L. (1981) *On Defining Semantics By Means of Extended Attribute Grammars in Semantics-Directed Compiler Generation* ed. N.D. Jones LNCS 94, pp. 259-299

[Mar] Marcotty, M. Ledgard, H.F. and Bochmann, C.V. (1976) *A Sampler of Formal Definitions* Comp. Surveys 8, pp. 191-276

[Mil1] Milner, A.J.R.G. (1976) *Program Semantics and Mechanized Proof in Foundtions of COmputer Science II* eds K.R. Apt and J.W. de Bakker. Mathematical Centre Tracts 82 Amsterdam

[Mil2] Milner, A.J.R.G. (1980) *A Calculus of Communicating Systems* LNCS 92

[Plo1] Plotkin, G.D. (1975) *Call-by-name, Call-by-value and The Lambda-Calculus* TCS 1, pp. 125-159

[Plo2] Plotkin, G.D. (1977) *LCF Considered As A Programming Language* TCS 5, pp. 223-225

[Sal] Salwicki, A.

[Wij] van Wijngarden, A., Maiiloux, B.J., Peck, J.E., Koster, C.H.A, Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T. and Fishker, R.G. (1975) *Revised Report ont the Algorithmic Language* Algol 68, Acta Informatica 5, pp. 1-236

# Chapter 3

# Definitions and Declarations

## 3.1 Introduction

In this chapter we begin the journey towards realistic programming languages by considering binding mechanisms which enable the introduction of new names in local contexts. This leads to definitions of local variables in applicative languages and declarations of constant and variable identifiers in imperative languages. We will distinguish the semantic concepts of environments and stores. The former concerns those aspects of identifiers which do not change throughout the evaluation of expressions or the execution of commands and so on; the latter concerns those aspects which do as in sideeffects in the evaluation of expressions or the effects of the execution of commands. In the static semantics context-free methods no longer suffice, and we show how our rules enable the context-sensitive aspects to be handled in a natural and syntax-directed way.

## 3.2 Simple Definitions in Appllcative Languages

We consider a little applicative (= functional) language with simple local definitions of variables. It can be considered as a first step towards full-scale languages like ML [Gor].

- **Syntax Basic Sets**

  **Numbers:** $m, n \in \mathrm{N}$

  **Binary Op.:** $bop \in \mathrm{Bop} = \{+, -, *\}$

  **Variables:** $x, y, z \in \mathrm{Var} = \{x_1, x_2, \ldots\}$

- **Derived Sets**

  **Expressions:** $e \in \mathrm{Exp}$ where

$$e ::= m \mid x \mid e_0 \; bop \; e_1 \mid \; \textbf{let } x = e_0 \textbf{ in } e_1$$

**Note**: Sometimes **let** $x = e_0$ **in** $e_1$ is written instead as $e_1$ **where** $x = e_0$. From the point of view of readability the first form is preferable when a bottom-up style is appropriate, and the second where a top-down style is appropriate. For in the first case one first defines $x$ and then uses it, and in the second it is used before being defined.

Clearly any expression contains various occurrences of variables, and in our language there are two kinds of these occurrences. First we have *defining* occurrences where variables are used. For example considering figure 1 the defining occurrences are 2, 6, 9 and the others are *applied.* In some languages - but not ours! - one finds other occurrences which can fairly be termed *useless.*

$$
\begin{array}{l}
x^1 * (\,\textbf{let}\ x^2 = 5 * y^3 * x^4 \\
\qquad \textbf{in}\ x^5 + (\,\textbf{let}\ y^6 = 14 - x^7 \\
\qquad\qquad \textbf{in}\ y^8 + (\,\textbf{let}\ x^9 = 3 + x^{10} + x^{11} \\
\qquad\qquad\qquad \textbf{in}\ x^{12} * y^{13})))
\end{array}
$$

Figure 1. Some Variable Occurrences

Now the region of program text over which defining occurrences have an influence is known as their it scope. One often says, a little loosely, that for example, the scope of the first occurrence of $x$ in $e =$ **let** $x = e_0$ **in** $e_1$ is the expression $e_1$. But then one considers examples like figure 1 where occurrence 12 is not in the scope of 2 (as it is instead in the scope of 9), this is called a **hole** in the scope of 2. It is more accurate to say that the scope of a defining occurrence is a set of applied occurrences. In the case of **let** $x = e_0$ **in** $e_1$ the scope of $x$ is all those applied occurrence of $x$ in $e_1$. Thus in the case of figure 1 we have the following table showing which applied occurrences are in the scope of which defining occurrences (equivalently which defining occurrences bind which applied occurrences).

| Defining Occurrence | Applied Occurrences |
|---|---|
| 2 | {5, 7, 10, 11} |
| 6 | {8, 13} |
| 9 | {12} |

Note that each applied occurrence is in the scope of at most one defining occurrence. Those not in any scope are termed *free*(versus *bound*); for example occurrences 1, 3, 4 above are free. One can picture the bindings and the free variables by means of a drawing with arrows such as:

$$\begin{array}{l} \textbf{let } x = 5 + y \\ \textbf{in let } y = 4 + x + y \\ \textbf{in } x + y + z \end{array}$$

From the point of view of semantics it is irrelevant which identifiers are chosen just so long as the same set of bindings is generated. (Of course a sensible choice of identifiers greatly affects *readability*, but that is not a semantic matter.) All we really need are the arrows, but it is hard to accommodate then into our one-dimensional languages. In the literature on $\lambda$-calculus one does find direct attempts to formalise the arrows and also attempts to eliminate variables altogether; as in *Combinatory Logic* [Hin]; In *Dataflow* one sees graphical languages where the graphs display the arrows [Ack].

### Static Semantics

- **Free Variables**
  The following definition by structural induction is of FV($e$), the set of free variables (= variables with free occurrences) of $e$:

| | $m$ | $x$ | $e_0$ bop $e_1$ | **let** $x = e_0$ **in** $e_1$ |
|---|---|---|---|---|
| FV | $\emptyset$ | $\{x\}$ | $V_0 \cap V_1$ | $V_0 \cap (V_1 \backslash [x])$ |

Where $V_0 = \mathrm{FV}(e_0)$ and $V_1 = \mathrm{FV}(e_1)$

- **Example**

$$\begin{aligned} &\mathrm{FV}(\textbf{let } x = 5 + y \textbf{ in } (\textbf{ let } y = 4 + y + z \textbf{ in } x + y + z)) \\ &= \mathrm{FV}(5 + y) \cup (\mathrm{FV}(\textbf{ let } y = 4 + x + y \textbf{ in } x + y + z) \backslash \{x\}) \\ &= \{y\} \cup ((\{x, y\} \cup (\{x, y, z\} \backslash \{y\})) \backslash \{x\}) \\ &= \{y\} \cup (\{x, y, z\} \backslash \{x\}) \\ &= \{y, z\} \end{aligned}$$

### Dynamic Semantics

For the most part applicative languages have no concept of state; there is only the evaluation of expressions in different environments (= semantic contexts). We take:

$$\mathrm{Env}_V = (V \longrightarrow \mathrm{N})$$

for any finite subset of $V$ of the set Var of variables and let $\rho$ range over $\text{Env} = \sum_V \text{Env}_V$ and write $\rho : V$ to mean that $\rho$ is in $\text{Env}_V$. Of course $\text{Env}_V = Store_V$, but we introduce a new notation in order to emphasize the new idea.

The set of configuration is also parameterized on $V$ and

$$\Gamma_V = \{e \in \text{Exp} \mid \text{FV}(e) \subseteq V\}$$
$$\text{T}_V = N$$

The transition relation is now relative to an environment and for any $\rho$: $V$ and $e$, $e^1$ in $\Gamma_V$ we write

$$\rho \vdash_V e \longrightarrow e'$$

and read that in (= given) environment $\rho$ one step of the evaluation of the expression e results in the expression $e'$. The use of the turnstile is borrowed from formal logic as we wish to think of the above as an assertion of $e \longrightarrow e'$ conditional on $\rho$ which in turn is thought of as an assertion supplied by the environment on the values of the free variables of $e$ and $e'$. As this environment will not change from step to step of the evaluation of an expression, we will often use, fixing $\rho$ in the transition relation, the transitive reflexive closure $\rho \vdash_V e \longrightarrow *e'$. It is left to the reader to define *relative* transition systems (i.e. to give a general definition of "relative transition system").

**Rules**:

**Variables:** $\rho \vdash_V x \longrightarrow \rho(x)$

**Binary Operations:**

1. $\rho \vdash_V e_0 \longrightarrow e_0' \Rightarrow \rho \vdash_V e_0 \; bop \; e_1 \longrightarrow e_0' \; bop \; e_1$
2. $\rho \vdash_V e_0 \longrightarrow e_0' \Rightarrow \rho \vdash_V m \; bop \; e_1 \longrightarrow m \; bop \; e_1'$
3. $\rho \vdash_V m \; bop \; m' \longrightarrow n \qquad (\text{where } n = m \; bop \; m')$

**Note**: To save space we are using an evident horizontal lay-out for our rules. That is the rule:

$$\frac{A_1 \;\ldots\ldots\; A_k}{A}$$

can alternatively be written in the form

$$A_1, \ldots\ldots, A_k \Rightarrow A.$$

**Definitions**: Informally, to evaluate $e =$ **let** $x = e_0$ **in** $e_1$ given $\rho$

1. Evaluate $e_0$ given $\rho$ to get the value $m_0$.
2. Change $\rho$ to $rho[m_0/x]$.
3. Evaluate $e_1$ given $\rho$ to get the value $m$.

Then $m$ is the value of $e$ given $\rho$.

The rules for one step of the evaluation are:

1. $$\frac{\rho \vdash_V e_0 \longrightarrow e_1'}{\rho \vdash_V \textbf{let } x = e_0 \textbf{ in } e_1 \longrightarrow \textbf{let } x = e_0' \textbf{ in } e_1}$$

2. $$\frac{\rho[m/x] \vdash_{V \cup \{x\}} e_1 \longrightarrow e_1'}{\rho \vdash_V \textbf{let } x = m \textbf{ in } e_1 \longrightarrow \textbf{let } x = m \textbf{ in } e_1'}$$

3. $\rho \vdash_V \textbf{let } x = m \textbf{ in } n \longrightarrow n$

Of course these rules are just a clearer version of those given in Chapter 2 for expressions (as suggested in exercise 4). Continuing the logical analogy our rules look like a Gentzen system of natural deduction [Pra] written in a linear way. Possible definitions of behaviour are left to the reader.

## 3.3 Compound Definitions

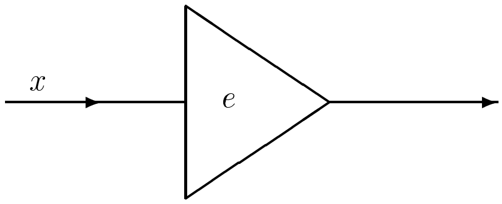
In general it is not convenient just to repeat simple definitions, and so we consider several ways of putting definitions together. The category of expressions is now:

$e ::= m \mid x \mid e_0 \; bop \; e_1 \mid \textbf{let } d \textbf{ in } e$

where $d$ ranges over the category Def of *definitions* where:

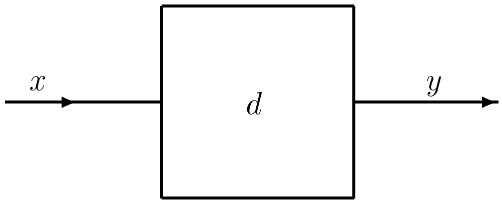
$d ::= \textbf{nil} \mid x = e \mid d_0 ; d_1 \mid d_0 \textbf{ and } d_1 \mid d_0 \textbf{ in } d_1$

To understand this it is convenient to think in terms of *import* and *export*. An *expression*, $e$, *imports* values for its free variables from its environment (and produces a value). This can be pictured as:



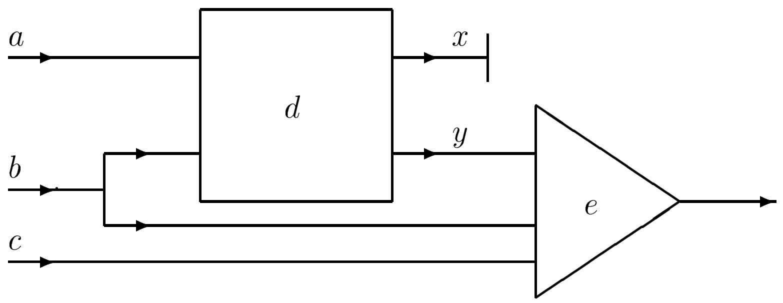

**An expression**

where $x$ is a typical free variable of $e$. A *definition*, $d$, *imports* values for its free variables and exports values for its defining variables (those with defining occurrences). This can be pictured as:




**A definition**

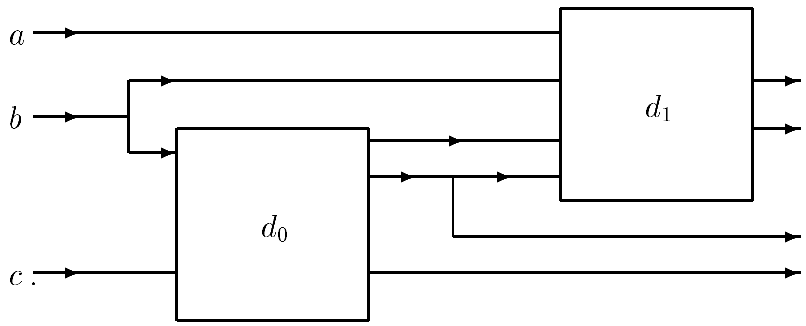
These are *dataflow diagrams* and they also help explain compound expressions and definition. For example a *definition block* **let** $d$ **in** $e$ imports from its environment into $d$ and then $d$ exports into $e$ with any other needed imports of $e$ coming from the block environment. Pictorially




**A definition block**

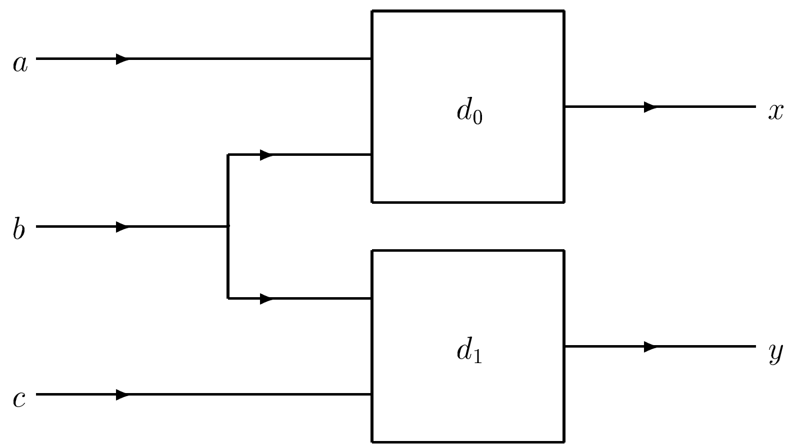
Here a is a typical variable imported by $d$ but not $e$, and b is one imported by $d$ and e, and c is one imported by $e$ and not $d$; again $x$ is a variable exported by $d$ and not imported by $e$ (useless but logically possible), and $y$ is a variable exported by $d$ and imported by $e$. Of course we later give a precise explanation of all this by formal rules of an operational semantics.

Turning to compound definitions we have *sequential* definition, $d_0; d_1$, and *simultaneous* definitions, $d_0$ and $d_1$, and *private* definitions, $d_0$ in $d_1$. What $d_0; d_1$ dose is import from the environment into $d_0$ and export from $d_0$ into $d_1$ (with any additional exports needed for $d_1$ being taken from the environment); then $d_0; d_1$ exports from both $d_0$ and $d_1$ with the latter taking precedence for common exports. Pictorially (and we need a picture!):



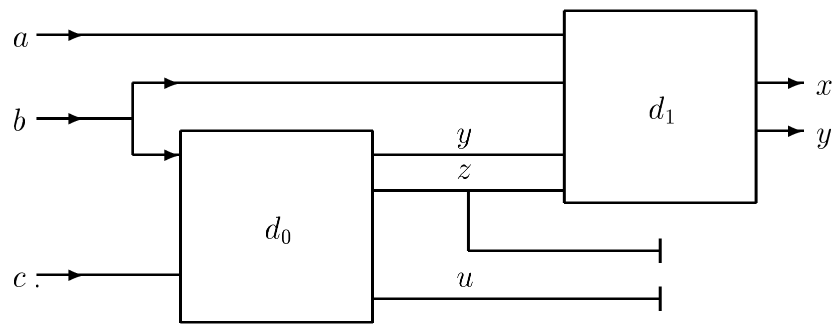

**Sequential Definition**

Simultaneous definition is much simpler; $d_0$ **and** $d_1$ imports into both $d_0$ and $d_1$ from the environment and then exports form both (and there must be no common defining variable). pictorially




**Simultaneous Definition**

Finally, a private definition $d_0$ **in** $d_1$ is just like a sequential one, except that the *only* exports are from $d_1$. It can be pictured as:




**Private Definition**

We may write also $d_0$ **in** $d_1$ as **let** $d_0$ **in** $d_1$ or as *private* do *within* $d_1$. Private definitions provide examples of blocks where the body is a definition.We have already seen blocks with expression bodies and will see ones with command bodies. Tennent's *Principle of Qualification* says that in principle any semantically meaningful syntactic class can be the body of a block [Ten]. We shall later encounter other examples of helpful organisational principles. As remarked in [Ten] many programming languages essentially force one construct to do jobs better done by several; for instance it is common to try to get something of the effect of both sequential and simultaneous definition. A little thought should convince the reader that there are essentially just the three interesting way of putting definitions together.

**Example**: Consider the expression

$$\begin{array}{l}\textbf{let } x = 3 \\ \textbf{in let } x = 5 \ \& \ y = 6 * x \\ \quad \textbf{in } x + y\end{array}$$

According as & is ; or **and** or **in**, the expression has the values $35 = 5+(6\times5)$ or $23 = 5+(6\times3)$ or $33 = 3 + (6 \times 5)$.

### Static Semantics

We will define the set DV($d$) of *defining variables* of a definition $d$ and also FV($d/e$), the set of *free variables* of a definition $d$ or expression $e$.

| | **nil** | $x = e$ | $d_0; d_1$ | $d_0$ **and** $d_1$ | $d_0$ **in** $d_1$ | |
|---|---|---|---|---|---|---|
| DV | $\emptyset$ | $x$ | $V_0 \cup V_1$ | $V_0 \cup V_1$ | $V_1$ | where $V_i$ is $\mathrm{DV}(d_i)$ |
| FV | $\emptyset$ | $FV(e)$ | $V_0 \cup (V_1 \backslash \mathrm{DV}(d_0))$ | $V_0 \cup V_1$ | $V_0 \cup (V_1 \backslash \mathrm{DV}(d_0))$ | where $V_i$ is $\mathrm{FV}(d_i)$ |

For expressions the definition of free variables is the same as before except for the case

$$\mathrm{FV}(\textbf{let } d \textbf{ in } e) = \mathrm{FV}(d) \cup (\mathrm{FV}(e) \backslash \mathrm{DV}(d))$$

Because of the restriction on simultaneous definitions not all expressions or definitions are well-formed - for example consider **let** $x = 3$ **and** $x = 6$ **in** $x$. So we also define the well-formed ones by means of rules for a predicate W($d/e$) on definitions and expressions.

**Rules**:

- **Definitions**

  **Nil:** $\mathrm{W}(\textbf{nil})$

  **Simple:** $\mathrm{W}(e) \Rightarrow \mathrm{W}(x = e)$

  **Sequential:** $\mathrm{W}(d_0), \mathrm{W}(d_1) \Rightarrow \mathrm{W}(d_0; d_1)$

  **Simultaneous:** $\mathrm{W}(d_0), \mathrm{W}(d_1) \Rightarrow \mathrm{W}(d_0 \textbf{ and } d_1)$ (if $\mathrm{DV}(d_0 \cap DV(d_1) = \emptyset)$

  **Private:** $\mathrm{W}(d_0), \mathrm{W}(d_1) \Rightarrow \mathrm{W}(d_0 \textbf{ in } d_1)$

- **Expressions**

  **Constants:** $\mathrm{W}(m)$

  **Variables:** $\mathrm{W}(x)$

  **Binary Op.:** $\mathrm{W}(c_0), \mathrm{W}(e_1) \Rightarrow \mathrm{W}(e_0 \; bop \; e_1)$

  **Definitions:** $\mathrm{W}(d), \mathrm{W}(e) \Rightarrow \mathrm{W}(\textbf{let } d \textbf{ in } e\})$

### Dynamic Semantics

It is convenient to inrtoduce some new notation to handle environments. For purposes of displaying environments consider, for exmaple, $\rho : \{x, y, z\}$, where $\rho(x) = 1$, $\rho(y) = 2$, $\rho(z) = 3$. We will also write $\rho$ as $\{x = 1, y = 2, z = 3\}$ and drop the set btackets when desired; this situation makes it clearer that environments can be thought of as assertions.

Next for any $V_0$,$V_1$ and $\rho_0$:$V_0$, $\rho_1$:$V_1$ we define $\rho = \rho_0[\rho_1]$:$V_0 \cup V_1$ by:

$$\rho(x) = \begin{cases} \rho_1(x) & (x \in V_1) \\ \rho_0(x) & (x \in V_0 \backslash V_1) \end{cases}$$

We now have the nice $\rho[x = m]$ to replace the less readable $\rho[m/x]$. Finally for any $\rho_0$:$V_0$, $\rho_1$:$V_1$ with $V_0 \cap V_1 = \emptyset$ we write $\rho_0$, $\rho_1$ for $\rho_0 \cup \rho_1$. Of course this is equale to $\rho_0[\rho_1]$, and also to $\rho_1[\rho_0]$, but the extra notation makes it clear that it is required that $V_0 \cap V_1 = \emptyset$.

The expression configurations are parameterised on $V$ by:

$$\Gamma_V = \{e \mid \mathrm{W}(e), \mathrm{FV}(e) \subseteq V\}$$

and of course

$$\mathrm{T}_V = N$$

And our transition relation, $\rho \vdash_V e \longrightarrow e'$, is defined only for $\rho : V$, and $e$, $e'$ in $\Gamma_V$.

For definitions the idea is that just as an expression is evaluate to yield values so is a definition *elaborated* to yield a “little” environment (for its defining variables). For example, given $\rho = \{x = 1, y = 2, z = 3\}$ the definition $x = 5 + x + z$; $y = x + y + z$ is elaborated to yield $\{x = 9, y = 14\}$. In order to make this work we add another cluse to the definition of Def

$$d ::= \rho$$

What this means is that the abstract syntax of declaration configurations allows environments; It does not mean that the abstract syntax of declarations does so.

In a sense we slipped a similar trich in under the carpet when we allowed numbers as expressions. Strictly speaking we should only have allowed literals and then allowed numbers as part of the configurations and given rules for evaluating literals to numbers. Similar statements hold for other kinds of literals. However, there seemed little point in forcing the reader through this tedious procedure.

Returning to definitions we now add clauses for free & defining variables:

$$\mathrm{FV}(\rho) = \emptyset$$
$$\mathrm{DV}(\rho) = V \qquad (\text{if } \rho : V)$$

and also add for any $\rho$ that $\mathrm{W}(\rho)$ holds, and for any $V$ that

$$\Gamma_V = \{d \mid \mathrm{W}(d), \mathrm{FV}(d) \subseteq V\}$$

and

$$\mathrm{T}_V = \{\rho\}$$

and consider for $V$ and $\rho$: $V$ and $d$, $d' \in \Gamma_V$ the transition relation

$$\rho \vdash_V d \longrightarrow d''$$

which means that, given $\rho$, one step of the elaboration of $d$ yields $d'$.

**Example**: We shall expect to see that:

$$\begin{aligned} x = 1, y = 2, z = 3 \vdash x = (5 + x) + z; y = (x + y) + z \\ &\longrightarrow^* \{x = 9\}; y = (x + y) + z; \\ &\longrightarrow^* \{x = 14\} \\ &\longrightarrow \{x = 9, y, = 14\} \end{aligned}$$

**Rules**:

- **Expressions**: As before but with a change for definitions:
- **Definitions**: informally, to evaluate $e_1 =$ let $d$ in $e_0$ in the environment $\rho$

  1. Elaborate $d$ in $\rho$ yielding $\rho_0$.
  2. Change $\rho$ to $\rho' = \rho[\rho_0]$.
  3. Evaluate $e$ in $\rho'$ yielding $m$.

  Then the evaluation of $e_1$ yield $m$. Formally we have:

  1. $\dfrac{\rho \vdash_V d \longrightarrow d'}{\rho \vdash_V \textbf{ let } d \textbf{ in } e \longrightarrow \textbf{ let } d' \textbf{ in } e}$
  2. $\dfrac{\rho[\rho_0] \vdash_{V \cup V_0} e \longrightarrow e'}{\rho \vdash_V \textbf{ let } \rho_0 \textbf{ in } e \longrightarrow \textbf{ let } \rho_0 \textbf{ in } e'}$ (where $\rho_0 : V_0$)
  3. $\rho \vdash_V \textbf{ let } \rho_0 \textbf{ in } m \longrightarrow m$

  The first two cases are self-explanatory.

  **Nil:** $\rho \vdash_V \textbf{nil} \longrightarrow \emptyset$

  **Simple:**
  1. $\rho \vdash_V e \longrightarrow e'' \quad \Rightarrow \quad \rho \vdash_V x = e \longrightarrow x = e'$
  2. $\rho \vdash_V x = m \longrightarrow \{x = m\}$

  **Sequential:** informally to elaborate $d_0; d_1$ given $\rho$.
  1. Elaborate $d_0$ in $\rho$ yielding $\rho_0$
  2. Elaborate $d_0$ in $\rho_0$ yielding $\rho_0$

  Then the elaboration of $d_0; d_1$ yields $\rho_0[\rho_1]$. Formally we have:

1. $\dfrac{\rho \vdash_V d_0 \longrightarrow d_0'}{\rho \vdash_V \ d_0; d_1 \longrightarrow \ d_0'; d_1}$
2. $\dfrac{\rho[\rho_0] \vdash_{V \cup V_0} d_1 \longrightarrow d_1'}{\rho \vdash_V \rho_0; d_1 \longrightarrow \rho_0; d_1'}$ (where $\rho_0 : V_0$)
3. $\rho \vdash_V \rho_0; \rho_1 \longrightarrow \ \rho_0[\rho_1]$

**Stimultaneous:** informally to elaborate $d_0$ **and** $d_1$ given $\rho$

1. Elaborate $d_0$ in $\rho$ yielding $\rho_0$.
2. Elaborate $d_0$ in $\rho$ yielding $\rho_1$.

Then the elaboration of $d_0$ **and** $d_1$ yields $\rho_0, \rho 1$ if that is defined. Formally

1. $\rho \vdash_V d_0 \longrightarrow d_0' \quad \Rightarrow \quad \rho \vdash_V d_0 \textbf{ and } d_1 \longrightarrow d_0' \textbf{ and } d_1$
2. $\rho \vdash_V d_0 \longrightarrow d_0' \quad \Rightarrow \quad \rho \vdash_V d_0 \textbf{ and } d_1 \longrightarrow \rho_0 \textbf{ and } d_1'$
3. $\rho \vdash_V \rho_0 \textbf{ and } \rho_1 \longrightarrow \rho_0 \cup \rho_1$

**Private:** informally to elaborate $d_0$ **in** $d_1$ given $\rho$

1. Elaborate $d_0$ in $\rho$ yielding $\rho_0$.
2. Elaborate $d_1$ in $\rho$ yielding $\rho_1$.

Then the elabolation of $d_0$ **in** $d_1$ yields $\rho_1$. Formally

1. $\rho \vdash_V d_0 \longrightarrow d_0' \quad \Rightarrow \quad \rho \vdash_V d_0 \textbf{ in } d_1 \longrightarrow d_0' \textbf{ in } d_1$
2. $\rho \vdash_{V \cup V_0} d_0 \longrightarrow d_0' \quad \Rightarrow \quad \rho \vdash_V d_0 \textbf{ in } d_1 \longrightarrow \rho_0 \textbf{ in } d_1'$
   (where $\rho_0 : V_0$)
3. $\rho \vdash_V \rho_0 \textbf{ in } \rho_1 \longrightarrow \rho_0$

**Example**:

$$
\begin{aligned}
x = 1, y = 2, z = 3 \vdash x = (5 + x) + z; y = (x + y) + z & \\
\overset{\textbf{SEQ1}}{\longrightarrow} x = (5 + 1) + z; y = (x + Y) + z & \text{ (using SIM1)} \\
\overset{\textbf{SEQ1}}{\longrightarrow} x = 9; y = (x + y) + z & \text{ (using SIM1)} \\
\overset{\textbf{SEQ1}}{\longrightarrow} \{x = 9\}; y = (x + y) + z & \text{ (using SIM1)} \\
\overset{\textbf{SEQ1}}{\longrightarrow} \{x = 9\}; y = (x + y) + z & \\
\overset{\textbf{SEQ1}}{\longrightarrow} \{x = 9\}; \{y = 14\} & \\
\overset{\textbf{SEQ1}}{\longrightarrow} \{x = 9, y = 14\}. &
\end{aligned}
$$

The reader is encouraged here (and generally too) to work out examples for all the other constructs.

## 3.4 Type-Checking and Definitions

New problems arise in static semantics when we consider type-checking and definitions. For example one cannot tell whether or not such an expression as $x$ **or** tt or $x + x$ is well-typed with out knowing what the type of $x$ is and that depends on the context of its occurrence. We Will be ables to solve these problems by introducing static environments $\alpha$ to give this type information and giving rules to establish properties of the form

$$\alpha \vdash_V e : \tau$$

As usual we work by considering an example language.

- **Basic Sets**

  **Types:** $\tau \in \mathrm{Types} = \{\mathrm{int}, \mathrm{bool}\}$

  **Numbers:** $m, n \in \mathrm{N}$;

  **Truthvalues:** $t \in \mathrm{T}$;

  **Variables:** $x, y, z \in \mathrm{Var}$;

  **Binary Operations:** $bop \in \mathrm{Bop} = \{+, -, *, =, \mathbf{or}\}$.

- **Derived Sets**

  **Constants:** $con \in \mathrm{Con}$ where $con ::= m/t$

  **Definitions:** $d \in \mathrm{Def}$ where

  $$d ::= \mathbf{nil} \mid x : \tau = e \mid d_0; d_1 \mid d_0 \ \mathbf{and} \ d_1 \mid d_0 \ \mathbf{in} \ d_1$$

  **Expressions:** $e \in \mathrm{Exp}$ where

  $$e ::= con \mid x \mid \sim e \mid e_0 \ bop \ e_1 \mid \mathbf{if} \ e_0 \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 \mid \\ \mathbf{let} \ d \ \mathbf{in} e$$

**Static Semantics**

The definitions of $\mathrm{DV}(d)$ and $\mathrm{FV}(d)$ are as before as is $\mathrm{FV}(e)$ just adding that

$$\mathrm{FV}(\mathbf{if} \ e_0 \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2) = \mathrm{FV}(e_0) \cup \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2)$$

We now need *type environments* over $V$. These form the set

$$\mathrm{TEnv}_V = V \longrightarrow \mathrm{Types}$$

and the set $\mathrm{TEnv}_V = \sum_V \mathrm{TEnv}_V$ is ranged over by $\alpha$ and $\beta$ and we write $\alpha : V$ for $\alpha \in \mathrm{TEnv}_V$. Of course all the notation $\alpha[\beta]$ and $\alpha$, $\beta$ extends without change from ordinary environments to type environments. Now for every $V$ and $\alpha{:}V$, $\tau$ and $e$ with $\mathrm{FV}(e) \subseteq V$ we give rules for the property

$$\alpha \vdash_V e : \tau$$

meaning that given $\alpha$ the expression $e$ is well-formed and has type $\tau$. This will involve us in giving similar rules for constants ans also for every $V$ and $\alpha : V$, $\beta$ and definition $d$ with $\mathrm{FV}(d) \in V$, for the property

$$\alpha \vdash_V d : \beta$$

meaning that given $\alpha$ the definition $d$ is well-formed and yields the type environment $\beta$.

**Examples**: 1. $y = \mathrm{bool} \vdash (\mathbf{let} x : \mathrm{int} = 1 \mathbf{in} (x = x) \mathbf{or} y) : \mathrm{bool}$

2. $y = \mathrm{bool} \vdash (x : \mathrm{int} = \mathbf{if}\ y\ \mathbf{then}\ 0\ \mathbf{else}\ 1; y : int = x + 1) : \{x = \mathrm{int}, y = \mathrm{int}\}$

**Rules**:

- **Constants:**

  **Numbers:** $\alpha \vdash_V m : \mathrm{int}$

  **Truthvalues:** $\alpha \vdash_V t : \mathrm{bool}$

- **Expressions:**

  **Constants:** $\alpha \vdash_V con : \tau \quad \Rightarrow \quad \alpha \vdash_V con : \tau$ (this makes sense!)

  **Variables:** $\alpha \vdash_V x : \alpha(x)$

  **Negation:** $\alpha \vdash_V e : \mathrm{bool} \quad \Rightarrow \quad \alpha \vdash_V \sim e : \mathrm{bool}$

  **Binary Operations:** $\dfrac{\alpha \vdash_V e_0 : \tau_0 \alpha \vdash_V e_1 : \tau_1}{\alpha \vdash_V e_0 \text{ bop } e_1 : \tau}$ (if $\tau = \tau_0\ \tau_{bop} \tau_1$)

  **Conditional:** $\alpha \vdash_V e_0 : \mathrm{bool}, \alpha \vdash_V e_1 : \tau, \alpha \vdash_V e_2 : \tau$
  $\Rightarrow \quad \alpha \vdash_V \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau$

  **Definition:** $\dfrac{\alpha \vdash_V d : \beta, \alpha[\beta] \vdash_{V \cup V_0} e : \tau}{\alpha \vdash_V \mathbf{let}\ d\ \mathbf{in}\ e : \tau}$ (where $\beta : V_0$)

Note that this allows the type of variables to be redefined.

**Definitions**:

**Nil:** $\alpha \vdash_V \mathbf{nil} : \emptyset$

**Simple:** $\alpha \vdash_V e : \tau \quad \Rightarrow \quad \alpha \vdash_V (x : \tau = e) : \{x = \tau\}$

**Sequential:** $\dfrac{\alpha \vdash_V d_0 : \beta_0, \alpha[\beta_0] \vdash_{V \cup V_0} d_1 : \beta_1}{\alpha \vdash_V (d_0; d_1) : \beta_0[\beta_1]}$ (where $\beta : V_0$)

**Simultaneous:** $\dfrac{\alpha \vdash_V d_0 : \beta_0 \ \ \alpha \vdash_V d_1 : \beta_1}{\alpha \vdash_V (d_0\ \mathbf{and}\ d_1) : \beta_0, \beta 1}$ (if $\mathrm{DV}(d_0) \cap \mathrm{DV}(d_1) = \emptyset$)

**Private:** $$\frac{\alpha \vdash_V d_0 : \beta_0, \alpha[\beta_0] \vdash_{V \cup V_0} d_1 : \beta_1}{\alpha \vdash_V (d_0 \textbf{ in } d_1) : \beta_1} \qquad (\text{where } \beta_0 : V_0)$$

It is hoped that these rules are self-explanatory. It is useful to define for any $V$ and $\alpha : V$ and $e$ with $\mathrm{FV}(e) \in V$ the property of being well-formed

$$\mathrm{W}_V(e, \alpha) \equiv \exists \tau.\ \alpha \vdash_V e : \tau$$

and also for any $V$, $\alpha : V$ and $d$ with $\mathrm{FV}(d) \subseteq V$ the property of being well-formed

$$\mathrm{W}_V(d, \alpha) \equiv \exists \beta.\ \alpha \vdash_V d : \beta.$$

**Dynamic Semantics**

If $x$ has type $\tau$ in environment $\alpha$ hen in the corresponding $\rho$ it should be the case that $\rho(x)$ also has type $\tau$; that is if $\tau = \mathrm{int}$, then we should have $\rho(x) \in \mathrm{N}$ and otherwise $\rho(x) \in \mathrm{T}$. To this end for any $V$ and $\alpha : V$ and $\rho : V \longrightarrow \mathrm{N} + \mathrm{T}$ we define:

$$\begin{aligned} \rho : \alpha \equiv \forall x \in V.\, & (\alpha(x) = \mathrm{int} \supset \rho(x) \in N) \\ \wedge\, & (\alpha(x) = \mathrm{bool} \supset \rho(x) \in T) \end{aligned}$$

and put $\mathrm{Env}_\alpha = \{\rho : V \longrightarrow N + T \mid \rho : \alpha\}$. Note that if $\rho_0 : \alpha_0$ and $\rho_1 : \alpha_1$ then $\rho_0[\rho_1] : \alpha_0[\alpha_1]$ and so too that (if it makes sense) $(\rho_0, \rho_1) : (\alpha_0, \alpha_1)$.

**Configurations**: We separate out the various syntactic categories according to the possible type environments.

- **Expressions:** form every $\alpha : V$ we put $\Gamma_\alpha = \{e \mid w_V(e, \alpha)\} and \mathrm{T}_\alpha = \mathrm{N} + \mathrm{T}$.
- **Definitions:** we add the production $d ::= \rho$ as before (but with $rho$ ranging over the $\sum_\alpha \mathrm{Env}_\alpha$) and then for every $\alpha : V$ we put $\Gamma_\alpha = \{d \mid \mathrm{W}_V(d, \alpha)\}$ and $\mathrm{T}_\alpha = \rho$.

**Transition relations**:

- **Expressions:** For every $\alpha : V$ we have the relation where $\rho : \alpha$ and $e, e' \in \Gamma_\alpha$:

  $$\rho \vdash_\alpha e \longrightarrow e'$$

- **Definitions:** For every $\alpha : V$ we have the relation where $\rho : \alpha$ and $d, d' \in \Gamma_\alpha$:

  $$\rho \vdash_\alpha d \longrightarrow d'$$

**Rules**: The rules are much as usual but with the normal constraints that all mentioned expressions and definitions be configurations and environments be of the right type-environment. here are three examples which should make the others obvious.

- **Expressions**:

**Definition 2:** $$\frac{\rho[\rho_0] \vdash_{\alpha[\alpha_0]} e \longrightarrow e'}{\rho \vdash_\alpha \textbf{ let } \rho_0 \textbf{ in } e \longrightarrow \textbf{let } \rho_0 \textbf{ in } e'} \qquad (\text{where } \rho_0 : \alpha_0)$$

- **Definitions**:

**Simple 2:** $\rho \vdash_\alpha x = \text{con} \longrightarrow \{x = con\}$

**Sequential 2:** $$\frac{\rho[\rho_0] \vdash_{\alpha[\alpha_0]} d_1 \longrightarrow d_1'}{\rho \vdash_\alpha \rho_0 : d_1 \longrightarrow \rho_0; d_1'} \qquad (\text{where } \rho_0 : \alpha_0)$$

**Example**:

$$\begin{array}{rl}
\{x + tt, y = 5\} \vdash_{\{x=\text{bool},y=\text{int}\}} & \textbf{let private}(x : \text{int} = 1 \textbf{ and } y : \text{int} = 2) \\
& \textbf{with in } z : int = x + y \\
& \textbf{in if } x \textbf{ then } y + z \textbf{ else } y \\
\longrightarrow^3 & \textbf{let private } \{x = 1, y = 2\} \\
& \textbf{with in } z : \text{int} = x + y \\
& \textbf{in if } x \textbf{ then } y + z \textbf{ else } y \\
\longrightarrow^4 & \textbf{let private } \{x = 1, y = 2\} \\
& \textbf{with in } \{z = 3\} \\
& \textbf{in if } x \textbf{ then } y + z \textbf{ else } y \\
\longrightarrow & \textbf{let } \{z = 3\} \textbf{ in if } x \textbf{ then } y + z \textbf{ else } y \\
\longrightarrow^2 & \textbf{let } \{z = 3\} \textbf{ in } y + z \\
\longrightarrow^4 & 8.
\end{array}$$

**Declarations in Imperative Languages**

The ideas so far developed transfer to imperative languages where we will speak of declarations (of identifiers) rather than definitions (of variables). Previously we have used stores for imperative languages and environments for applicative ones, although mathematically they are the same - associations of values to identifiers/variables. It now seems appropriate, however, to use both environments ans stores; the former shows what does not vary and the latter what does vary when commands are executed.

It is also very convenient to change the definitions of stores by introducing an (arbitrary infinite set, Loc, of locations (= references = cells) and taking for any $L \subseteq \text{Loc}$

$$\text{Stores}_L = L \longrightarrow \text{Values}$$

and

$$\text{Stores} = \sum_L \text{Stores}_L \qquad ( = \text{Loc} \longrightarrow_{\text{fin}} \text{Values})$$

and putting

$$\text{Env} = \text{ld} \longrightarrow_{\text{fin}} (\text{Values} + \text{Loc})$$

The idea is that if in some environment $\rho$ we have an identifier $x$ whose values should not vary the $\rho(x)$ = that value; otherwise $\rho(x)$ is a location, $l$, and given a store $\sigma : L$ (with $l$ in $L$) then $\sigma l$ is the value held in the location $l$ (its *contents*). In the first case we talk of *constant* identifiers and in the second we talk of *variable* identifiers. The former are introduced by constant declatarions like

$$\textbf{const}\ \ x = 5$$

and the *latter* by variable declarations like

$$\textbf{var}\ \ x = 5$$

In all cases declarations will produce new (little) environments, just as before. The general form of transitions will be:

$$\rho \vdash_l \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle$$

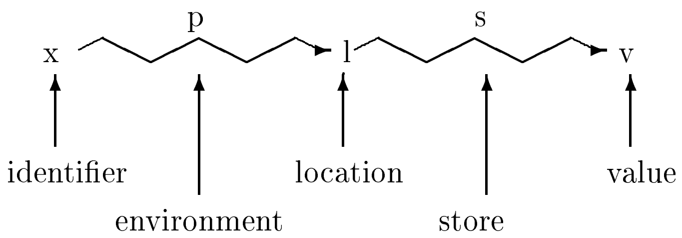
where $\rho$ is the elaboration environment and $\sigma$, $\sigma'$ are the *stores*. So, for example we will have

$$\rho \vdash_l \langle \textbf{const}\ x = 5, \sigma\rangle \longrightarrow \langle \{x = 5\}, \sigma\rangle$$

and

$$\rho \vdash_l \langle \textbf{var}\ x = 5, \sigma\rangle \longrightarrow \langle \{x = l\}, \sigma[l = 5]\rangle \qquad (*)$$

where $l$ is a certain "new" location. Locations can be thought of as "abstract addresses" where we do not really want to commit ourselves to any machine architecture, but only to the needed logical properties. Abetter way to think of a location is as an *individual* or *object* which has *lifetime* (=extent); it is created in a transition such as (*) and its lifetime continues either throughout the entire computation (execution sequence) or until it is *deleted* (= disposed of) (the declaration being achieved either through such mechanisms as block exit or through explicit storage management primitives in the language). Throughout its lifetime it has a (varying) contents, generally an ordinary mathematical values (or perhaps other locations). It is generally referred to by some identifier and is then said to be the *L-value* (or left-hand value) of the identifier and its contents, in some state, is the *R-value*(right-hand value) of the identifier, in that state. The lifetime of the location is related to, but logically distinct from the scope of the identifier. Thus we have a two-level picture



The L/R value terminology comes from considering assignment statements

$$x := y$$

where on the left we think of $x$ as referring to a location and on the right as referring to a value. Indeed we analyse the effect of assignment as changing the contents of the location to the R-value of $y$:

$$\rho \vdash \langle x := y, \sigma \rangle \longrightarrow \sigma[\rho x = \sigma(\rho y)]$$

This is of course a more complicated analysis of assignment than in Chapter2. the L/R terminology is a little inappropriate in that some programming languages write their assignments in the opposite order and also in that not all occurrences on the left of an assignment are references to L-values. The general idea of locations and separation of environments and stores comes from the Scott-Strachey tradition (e.g. [Gor, Ten, Led]); it is also reminiscent of ideas of individuals in modal logic [Hug]. In fact we do not need locations for most of the problems we encounter in the rest of later concepts such as

- Static binding of the same global variables in different procedure bodies (storage sharing).
- Call-by-reference (aliasing problems).
- Arrays (location expressions).
- Reference types (anonymous references).

On the other hand it would be interesting to see how far one can get without locations and to what extent programming languages would suffer from their excision (see [Don][Rey]). One can argue that it is the concept of location that distinguishes imperative from applicative languages.

We now make all this precise by considering a suitable mini-language.

**Syntax**:

- **Basic Sets**:

| | |
|---|---|
| **Types:** | $\gamma \in Types = \{\text{bool}, \text{int}\}$ |
| **Numbers:** | $m, n \in \text{N}$ |
| **Truthvalues:** | $t \in \text{T}$ |
| **Identifiers:** | $x, y, z \in \text{Id}$ |
| **Binary Operations:** | $bop \in \text{Bop}$ |

- **Derived Sets**

| | |
|---|---|
| **Constants:** | $con \in \text{Con}$ where $con ::= m \mid t$ |
| **Expressions:** | $e \in \text{Exp}$ where |

$$e ::= con \mid x \mid \sim e \mid e_0 \text{ bop } e_1 \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$$

**Declarations:** $d \in$ Dec where

$$e ::= \mathbf{nil} \mid \mathbf{const}\ x; \gamma = e \mid \mathbf{var}\ x : \gamma = e \mid d_0; d_1 \mid$$
$$d_0\ \mathbf{and}\ d_1 \mid d_0\ \mathbf{in}\ d_1$$

**Commands:** $c \in$ Com where

$$e ::= \mathbf{nil} \mid x := e \mid c_0; c_1 \mid \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid$$
$$\mathbf{while}\ e\ \mathbf{do}\ c \mid d; c$$

**Note**: On occasion we write **begin** $c$ **end** for $(c)$. That is **begin** ... **end** act as command parentheses, and have no particular semantic significance. However, their use can make scopes more apparent.

The whole of out discussion of defining applied and free and bound occurrences caries over to commands and is illustrated by the command in figure 2.

**var** $x$ : bool = $tt$ ;

**var** $y$ : int = **if** $x$ **then** 0 **else** $z$;

**const** $z$ : bool = if $\sim (x = 0)$ **then** $tt$ **else** $v$;

**begin**

$y$ : = if $x$ **then** 0 **else** $z$

$x$ : = $tt$ or $v$

**end**

**Bindings**

Note that left-hand variable occurrences in assignments are applied, not binding.

### Static Semantics

**Identifiers**: For **expressions** we need the set, FI($e$), of identifiers occurring freely in $e$ (defined as usual). For **declarations** we need the sets FI($d$) and DI($d$) of identifiers with free and defining occurrences in $d$; they are defined just like in the case of definitions and of course

$$\mathrm{FI}(\mathbf{const}\ x : \gamma = e) = \mathrm{FI}(\mathbf{var}\ x : \gamma = e) = \mathrm{FI}(e)$$
$$\mathrm{DI}(\mathbf{const}\ x : \gamma = e) = \mathrm{DI}(\mathbf{var}\ x : \gamma = e) = \{x\}$$

For commands we only need FI($c$) defined as usual plus $\mathrm{FI}(d; c) = \mathrm{FI}(c) \backslash \mathrm{DI}(d)$.

**Type Checking**: We take

$$\text{TEnv} = \text{Id} \longrightarrow_{\text{fin}} (\text{Types} + \text{Types} \times \{\text{loc}\}).$$

and write $\alpha : I$ for any $\alpha$ in TEnv with domain $I \subseteq Id$. The idea is that $\alpha(x) = \gamma$ means that $x$ denotes a value of type $\tau$, whereas $\alpha(x) = \tau$ loc ( $= \langle \tau, \text{loc} \rangle$) means that $x$ denotes a location which holds a value of type $\tau$.

**Assertions**:

- **Expression**: For each l and expression e with $\text{FI}(e) \subseteq I$ and type-environment $\alpha : I$ we define

  $$\alpha \vdash_I e : \tau$$

  meaning that given $\alpha$ the expression $e$ is well-formed and of type $\tau$.

- **Declarations**: Here for each $I$ and declaration $d$ with $\text{FI}(d) \subseteq I$ and type-environment $\alpha : I$ we define

  $$\alpha \vdash_l e : \beta$$

  meaning that given $\alpha$ the declaration $d$ is well-formed and yield the type-environment $\beta$.

- **Commands**: Here for eacg $I$ and command $c$ with $\text{FI}(C) \subseteq I$ and type-environment $\alpha : I$ we define:

  $$\alpha \vdash_I c$$

  meaning that given $\alpha$ the command $c$ is well-formed.

**Rules**:

- **Expressions**: As usual except for identifiers where:

  **Identifiers:** $\alpha \vdash_l x : \tau \qquad (\text{if } \alpha(x) = \tau \text{ or } \alpha x = \tau \text{ loc})$

- **Declarations**: Just like definitions before, except for simple ones:

  **Constants:** $$\frac{\alpha \vdash_l e : \tau}{\alpha \vdash_l \textbf{const } x : \tau = e : \{x = \tau\}}$$

  **Variables:** $$\frac{\alpha \vdash_l e : \tau}{\alpha \vdash_l \textbf{var } x : \tau = e : \{x = \tau\} \text{ loc}}$$

- **Commands**: The rules are similar to those in Chapter 2. We give an illustrative sample.

  **Nil:** $\alpha \vdash_l \textbf{nil}$

$$\textbf{Assignment:} \quad \frac{\alpha \vdash_l e; \tau}{\alpha \vdash_l x := e} \qquad (\text{if } \alpha(x) = \tau \text{ loc})$$

$$\textbf{Sequencing:} \quad \frac{\alpha \vdash_l c_0 \qquad \alpha \vdash_l c_1}{\alpha \vdash_l c_0; c_1}$$

$$\textbf{Blocks:} \quad \frac{\alpha \vdash_l c_0 \quad \alpha[\beta] \vdash_{l \cup l_0} c}{\alpha \vdash_l d; c} \qquad (\text{where } \beta : l_0)$$

**Dynamic Semantics**

Following the ideas on environments and stores we consider suitably typed locations and assume we have for eacs $\tau$ infinite sets

$$\mathrm{Loc}_\tau$$

which are disjoint and that (in order to create new locations) we have for each $l \subseteq \mathrm{Loc}_\tau$ a location $\mathrm{New}_\tau(l) \notin l$ (the *new* property).

**Note**: It is very easy to arrange these matters. Just put $\mathrm{Loc}_\tau = \mathrm{N} \times \{\tau\}$ and $\mathrm{New}_\tau(l) = \langle \mu m. \langle m, \tau \rangle \notin l, \tau \rangle$.

Now putting $\mathrm{Loc} = \bigcup_\sigma \mathrm{Loc}_\sigma$ we take for

$$\begin{aligned} \mathrm{Stores} = \{\sigma : L \subseteq \mathrm{Loc} \longrightarrow_{fin} Con \mid \forall\ & | \in \mathrm{Loc}_{\mathrm{int}} \cap L . \sigma(l) \in \mathrm{N} \\ \wedge \forall\ & | \in \mathrm{Loc}_{\mathrm{bool}} \cap L . \sigma(l) \in \mathrm{T} \end{aligned}$$

(as Con is the set of values). And we also take

$$\mathrm{Env} = \mathrm{Id} \longrightarrow_{\mathrm{fin}} \mathrm{Con} + \mathrm{Loc}$$

For any $\rho : l$ and $\alpha : l$ we define $\rho : \alpha$ by:

$$\begin{aligned} \rho : \alpha \equiv \forall x \in l(\alpha(x) = \mathrm{bool} \wedge \rho(x) in \mathrm{T}) \ & \vee (\alpha(x) = \mathrm{int} \wedge \rho(x) \in \mathrm{N}) \\ & \vee \exists \tau - 20(\alpha(x) = \tau \text{ loc } \wedge \rho(x) \in Loc_\tau \end{aligned}$$

**Transition Relations**:

- **Expressions:** For any $\alpha : I$ we set

$$\Gamma_\alpha = \{\langle e, \sigma \rangle \mid \exists \tau . \alpha \vdash_l e : \tau\}$$
$$T_\alpha = \{\langle con, \sigma \rangle\}$$

  and for any $\alpha : I$ we will define transition relations of the form

$$\rho \vdash_\alpha \langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$$

  where $\rho : \alpha$ and $\langle e, \sigma \rangle$ and $\langle e', \sigma' \rangle$ are in $\Gamma_\alpha$.

- **Declarations**: We extend Dec by adding the production

  $$d ::= \rho$$

  and putting $\mathrm{FI}(\rho) = \emptyset$ and $\mathrm{DI}(\rho) = I$ (where $\rho : I$), and putting $\alpha \vdash_I \rho : \beta$ (where $\rho : \beta$). Now for any $\alpha : I$ we take

  $$\Gamma_\alpha = \{\langle d, \sigma\rangle \mid \exists \beta \cdot \alpha \vdash_l d : \beta\} \qquad \text{and} \qquad \mathrm{T}_\alpha = <\rho, \sigma>$$

  and the transition relation has the form

  $$\rho \vdash_\alpha \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle$$

  where $\rho : \alpha$ and $\langle d, \sigma\rangle$ and $\langle d', \sigma'\rangle$ are in $\Gamma_\alpha$.

- **Commands**: For any $\alpha : l$ we take

  $$\Gamma_\alpha = \{\langle c, \sigma\rangle \mid \alpha \vdash_l c\} \cup \{\sigma\} \qquad \text{and} \qquad \mathrm{T}_\alpha = \{\sigma\}$$

  and the transition relation has the form

  $$\rho \vdash_\alpha \langle c, \sigma\rangle \longrightarrow \langle c', \sigma'\rangle \qquad (\text{or } \sigma')$$

  where $\rho : \alpha$ and $\langle c, \sigma\rangle$ and $\langle c', \sigma'\rangle$ (or $\sigma'$) are in $\Gamma_\alpha$.

**Rules**:

- **Expressions:** These should be fairly obvious and we just give some examples.

  **Identifiers:**
  1. $\rho \vdash_\alpha \langle x, \sigma\rangle \longrightarrow \langle con, \sigma\rangle \qquad (\text{if } \rho(x) = \text{con})$
  2. $\rho \vdash_\alpha \langle x, \sigma\rangle \longrightarrow \langle con, \sigma\rangle, \qquad (\text{if } \rho(x) = l \text{ and } \sigma(l) = \text{con})$

  **Conditional:**
  1. $$\frac{\rho \vdash_\alpha \langle e_0, \sigma\rangle \longrightarrow \langle e_0', \sigma'\rangle}{\rho \vdash_\alpha \langle \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \sigma\rangle \longrightarrow \langle \textbf{if } e_0' \textbf{ then } e_1 \textbf{ else } e_2\rangle}$$
  2. $\langle \textbf{if } \mathrm{tt} \textbf{ then } e_1 \textbf{ else } e_2, \sigma\rangle \longrightarrow \langle e_1, \sigma\rangle$
  3. $\langle \textbf{if } \mathrm{ff} \textbf{ then } e_1 \textbf{ else } e_2, \sigma\rangle \longrightarrow \langle e_2, \sigma\rangle$

- **Declarations:**

  **Nil:** $\rho \vdash_\alpha \langle \textbf{nil}, \sigma\rangle \longrightarrow \langle \emptyset, \sigma\rangle$

  **Constants:**
  1. $$\frac{\rho \vdash_\alpha \langle e_0, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}{\rho \vdash_\alpha \langle \textbf{const } x : \tau \ = e, \sigma\rangle \longrightarrow \langle \textbf{const } x : \tau = e', \sigma'\rangle}$$
  2. $\rho \vdash_\alpha \langle \textbf{const } x : \tau \ = \text{con}, \sigma\rangle \longrightarrow \langle \{x = \text{con}\}, \sigma\rangle$

  **Variables:** Informally to elaborate **var** $x : \tau = e$ from state $\sigma$ given $\rho$
  1. Evaluate $e$ from state $\sigma$ given $\rho$ yielding con.
  2. Get a new location $l$ and change $\sigma$ to $\sigma[l = \text{con}]$ and yield $\{x = l\}$

Formally

1. $$\frac{\rho \vdash_\alpha \langle e_0, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}{\rho \vdash_\alpha \langle \mathbf{var}\ x : \gamma\ = e, \sigma\rangle \longrightarrow \langle \mathbf{var}\ x; \gamma = e', \sigma'\rangle}$$
2. $\rho \vdash_\alpha \langle \mathbf{var}\ x : \tau\ = \text{con}, \sigma\rangle \longrightarrow \langle \{x = l\}, \sigma[l = \text{con}]\rangle$
   (where $\sigma : L$ and $l = \text{New}_\tau(L \cap Loc_\tau)$)

**Sequential:**

1. $$\frac{\rho \vdash_\alpha \langle d_0, \sigma\rangle \longrightarrow \langle d_0', \sigma'\rangle}{\rho \vdash_\alpha \langle d_0; d_1, \sigma\rangle \longrightarrow \langle d_0'; d_1, \sigma'\rangle}$$
2. $$\frac{\rho[\rho_0] \vdash_{\alpha[\alpha_0]} \langle d_0, \sigma\rangle \longrightarrow \langle d_0', \sigma'\rangle}{\rho \vdash_\alpha \langle \rho_0; d_1, \sigma\rangle \longrightarrow \langle \rho_0; d_1', \sigma'\rangle} \qquad \text{(where } \rho_0 : \alpha_0)$$
3. $\rho \vdash_\alpha \langle \rho_0; \rho_1, \sigma\rangle \longrightarrow \langle \rho_0[\rho_1], \sigma\rangle$

**Private:**

1./2. Like Sequential

3. $\rho \vdash_\alpha \langle \rho_0\ \mathbf{in}\ \rho_1, \sigma\rangle \longrightarrow \langle \rho_1, \sigma\rangle$

**Simultaneous:**

1. Like Sequential
2. $$\frac{\rho \vdash_\alpha \langle d_0, \sigma\rangle \longrightarrow \langle d_0', \sigma'\rangle}{\rho \vdash_\alpha \langle \rho_0\ \mathbf{and}\ d_1, \sigma\rangle \longrightarrow \langle \rho_0\ \mathbf{and}\ d_1', \sigma'\rangle}$$
3. $\rho \vdash_\alpha \langle \rho_0\ \mathbf{and}\ \rho_1, \sigma\rangle \longrightarrow \langle \rho_0 \cup \rho_1 \sigma\rangle$

**Note**: These definitions follow those for definitions very closely.

- **Commands:** On the whole the rules for commands are much like those we have already seen in Chapter 2.

**Nil:** $\rho \vdash_\alpha \langle \mathbf{nil}, \sigma\rangle \longrightarrow \sigma$

**Assignment:**
$$\frac{\rho \vdash_\alpha \langle e, \sigma\rangle \longrightarrow^* \langle con, \sigma'\rangle}{\rho \vdash_\alpha \langle x := e, \sigma\rangle \longrightarrow \sigma' l = \text{con}}$$
(where $\rho(x) = l$, and if $l \in L$ where $\sigma : L$)

**Composition:** 1./2. Like Chapter 2, but with $rho$.

**Conditional, While:** Like Chapter 2, but with $rho$.

**Blocks:** Informally to execute $d$; c from $\sigma$ given $\rho$

1. Elaborate $d$ from $\sigma$ given $\rho$ yielding $\rho_0$ and a store $\sigma'$.
2. Execute $C$ from $\sigma$ given $\rho[\rho_0]$ yielding $\sigma''$. Then $\sigma''$ is the result of the execution.

1. $$\frac{\rho \vdash_\alpha \langle d_0, \sigma\rangle \longrightarrow \langle d_0', \sigma'\rangle}{\rho \vdash_\alpha \langle d_0; c, \sigma\rangle \longrightarrow \langle d_0'; c, \sigma'\rangle}$$
2. $$\frac{\rho[\rho_0] \vdash_{\alpha[\alpha_0]} \langle c, \sigma\rangle \longrightarrow \langle c', \sigma'\rangle}{\rho \vdash_\alpha \langle \rho_0; c, \sigma\rangle \longrightarrow \langle \rho_0; c', \sigma\rangle'} \qquad (\rho_0 : \alpha_0)$$
3. $$\frac{\rho[\rho_0] \vdash_{\alpha[\alpha_0]} \langle c, \sigma\rangle \longrightarrow \sigma'}{\rho \vdash_\alpha \langle \rho_0; c, \sigma\rangle \longrightarrow \sigma'}$$

In the above we have not connected up $\rho$ and $\sigma$. In principle it could happen either that

1. There is an $l$ in the range of $\rho$ but not in the domain of $\sigma$. This is an example of a **dangling** reference. They are also possible in relation to a configuration such as $\langle c, \sigma \rangle$ where $l$ occurs in $c$ (via some $\rho$) but not in the domain of $\sigma$.

2. There is an $l$ not in the range of $\rho$ but in the domain of $\sigma$. And similarly wrt $c$ and $\sigma$, etc. This is an example of an *inaccessible* reference. However, we easily show that if for example that the language has no *storage insecurities*. As easy way to obtain a language which is not secure is to add the command

   $$c ::= \text{dispose}(x)$$

   with the dynamic semantics

   $$\rho \vdash_\alpha \langle \text{dispose}(x), \sigma \rangle \longrightarrow \sigma \backslash l \qquad (\text{where } l = \rho(x))$$

   (and $\sigma \backslash l = \sigma \backslash \{\langle l, \sigma(l) \rangle\}$) (and obvious static semantics). One might wish to add an error rule for attempted assignments to dangling references.

On the other hand according to out semantics we do have inaccessible references. For example a block exit

$$\begin{aligned} \rho \vdash \langle \mathbf{var}\ x : \text{bool} = \text{tt}, \mathbf{begin\ nil\ end}, \sigma \rangle &\longrightarrow \langle \{x = l\}; \mathbf{nil}, \sigma[l = \text{tt}] \rangle \\ &\longrightarrow \sigma[l = \text{tt}] \end{aligned}$$

Another example is provided by sequential or private definitions, e.g.

$$\begin{aligned} \rho \vdash \langle \mathbf{var}\ x : \text{bool} = \text{tt}; \mathbf{var} x : \text{bool} = \text{tt}, \sigma \rangle &\longrightarrow \langle \{x = l_1\}; \mathbf{var}\ x : \text{bool} = \text{tt}, \sigma[l_1 = \text{tt}] \rangle \\ &\longrightarrow \langle \{x = l_1\}; \{x = l_2\}, \sigma[l_1 = \text{tt}, l_2 = \text{tt}] \rangle \\ &\longrightarrow \langle \{x = l_1\}, \sigma[l_1 = \text{tt}, l_2 = \text{tt}] \rangle \end{aligned}$$

and again

$$\begin{aligned} \rho \vdash \langle \mathbf{var}\ x : \text{bool} = \text{tt} \mathbf{in\ var} y : \text{bool} = \text{tt}, \sigma \rangle &\longrightarrow^* \langle \{x = l_1 \ \mathbf{in}\ y = l_2\}, \sigma[l_1 = \text{tt}, l_2 = \text{tt}] \rangle \\ &\longrightarrow \langle \{y = l_2\}, \sigma[l_1 = \text{tt}, l_2 = \text{tt}] \rangle \end{aligned}$$

It is not clear whether inaccessible references should be allowed. They can easily be avoided, at the cost of complicating the definitions, by "pruning" them away as they are created, a kind of logical garbage collection. We prefer here to leave them in, for the sake of simple definitions; the do not, unlike dangling references, cause any harm.

The semantics for expressions is a little more complicated than necessary in that if $\rho \vdash \langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ then $\sigma = \sigma'$; that is there are no *side-effects*. However, the extra generality will prove useful. For example suppose we had a production:

$$\begin{aligned} e ::= {} & \mathbf{begin}\ c \\ & \mathbf{result}\ e \end{aligned}$$

To evaluate **begin** $c$ **result** $e$ from $\sigma$ given $\rho$ one first executes $c$ from $\sigma$ given $\rho$ yielding $\sigma'$ and then evaluates $e$ from $\sigma'$ given $\rho$. The transition rules would, of course, be:

$$\frac{\rho \vdash_\alpha \langle c, \sigma\rangle \longrightarrow \langle c', \sigma'\rangle}{\rho \vdash_\alpha \langle \mathbf{begin}\ c\ \mathbf{result}\ e, \sigma\rangle \longrightarrow \langle \mathbf{begin}\ c'\ \mathbf{result}\ e, \sigma'\rangle}$$
$$\frac{\rho \vdash_\alpha \langle c, \sigma\rangle \longrightarrow \sigma'}{\rho \vdash_\alpha \langle \mathbf{begin}\ c\ \mathbf{result}\ e, \sigma\rangle \longrightarrow \langle e, \sigma'\rangle}$$

(and the static semantics is obvious).

With this construct one also has now the possibility of side-effects during the elaboration of definitions; previously we had instead that if

$$\rho \vdash_V \langle d, \sigma\rangle \longrightarrow \langle d', \sigma''\rangle$$

then $\sigma' \mid_L = \sigma$ where $\sigma : L$.

We note some other important constructs. The principle of qualification suggests we include expression blocks:

$$\begin{aligned} e ::= {} & \mathbf{let}\ d \\ & \mathbf{in}\ e \end{aligned}$$

with evident static semantics and the rules.

$$\frac{\rho \vdash_\alpha \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle}{\rho \vdash_\alpha \langle \mathbf{let}\ d\ \mathbf{in}\ \sigma\rangle \longrightarrow \langle \mathbf{let}\ d'\ \mathbf{in}\ \sigma'\rangle}$$
$$\frac{\rho \vdash_{\alpha[\alpha_0]} \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}{\rho \vdash_\alpha \langle \mathbf{let}\ \rho_0\ \mathbf{in}\ e, \sigma\rangle \longrightarrow \langle \mathbf{let}\ \rho_0\ \mathbf{in}\ e', \sigma'\rangle} \qquad (\text{where } \rho_0 : \alpha_0)$$
$$\rho \vdash_\alpha \langle \mathbf{let}\ \rho_0\ \mathbf{in}\ con, \sigma\rangle \longrightarrow \langle con, \sigma\rangle$$

As another kind of atomic declaration consider

$$d ::= x == y$$

meaning that $x$ should refer to the location referred to by $y$ (in $\rho$). The relevant static semantics will, of course, be:

$$\mathrm{DI}(x == y) = \{x\}; \qquad \mathrm{FI}(x == y) = \{y\}$$
$$\alpha \vdash_l x == y : \{x = \tau\ \mathrm{loc}\} \qquad (\text{if } \alpha(y) = \tau\ \mathrm{loc})$$

and the dynamic semantics is:

$$\rho \vdash_\alpha \langle x == y, \sigma\rangle \longrightarrow \langle x = l, \sigma\rangle \qquad (\text{if } \rho(y) = l)$$

This construct is an example where it is hard to do without locations; more complex versions allowing the evaluation of expressions to references will be considered in the next chapter.

It can be important to allow initialisation commands in declarations such as

$$
\begin{aligned}
d ::= d&\\
&\textbf{initial}\\
&\qquad c\\
&\textbf{end}
\end{aligned}
$$

and the static semantics is:

$$\mathrm{DI}(d\ \textbf{initial}\ c\ \textbf{end}\ ) = \mathrm{DI}(d); \qquad \mathrm{FI}(d\ \textbf{initial}\ c\ \textbf{end}) = \mathrm{FI}(d) \cup (\mathrm{FI}(c) \backslash \mathrm{DI}(d))$$

and

$$\frac{\alpha \vdash_l d : \beta \qquad \alpha[\beta] \vdash_{l \cup l_0} c}{\alpha \vdash_l d\ \textbf{initial}\ c\ \textbf{end} : \beta} \qquad (\text{if } \beta : l_0)$$

However, we may wish to add other conditions(like the drastic $\mathrm{FI}(c) \subseteq \mathrm{DI}(d)$) to avoid side-effects. The dynamic semantics is:

$$
\begin{array}{ll}
\dfrac{\rho \vdash_\alpha \langle d, \sigma \rangle \longrightarrow, \langle d', \sigma' \rangle}{\rho \vdash_\alpha \langle d\ \textbf{initial}\ c\ \textbf{end}, \sigma \rangle \longrightarrow \langle d'\ \textbf{initial}\ c\ \textbf{end}, \sigma' \rangle} & \\
\dfrac{\rho \vdash_{\alpha[\alpha_0]} \langle c, \sigma \rangle \longrightarrow \langle d', \sigma' \rangle}{\rho \vdash_\alpha \langle \rho_0\ \textbf{initial}\ c\ \textbf{end}, \sigma \rangle \longrightarrow \langle \rho_0\ \textbf{initial}\ c', \sigma' \rangle} & (\text{where } \rho 0 : \alpha 0) \\
\dfrac{\rho \vdash_{\alpha[\alpha_0]} \langle c, \sigma \rangle \longrightarrow sigma'}{\rho \vdash_\alpha \langle \rho_0\ \textbf{initial}\ c\ \textbf{end}, \sigma \rangle \longrightarrow \langle \rho_0, \sigma' \rangle} &
\end{array}
$$

In the exercises we consider a dual idea of *declaration finalisation* commands which are executed after the actions associated with the scope rather than before the scope of the declaration.

Finally, we stand back a little and look at the various classes of *values* associated with our language.

- **Expressible Values:** These are the values of expressions. In our language this set, EVal, is just the set, Con, of constants.

- **Denotable Values:** These are the values of identifiers in environments. Here the set, DVal, is the set Con + Loc of constants and locations. Note, that $\mathrm{Env} = \mathrm{Id} \longrightarrow_{\mathrm{fin}} \mathrm{DVal}$.

- **Storeable Values:** These are the values of locations in the store. Here, the set, SVal, is the set Con of constants. Note, that Stores is the set of type-respecting finite maps from Loc to SVal.

Thus we can consider the sets EVal, DVal, SVal of expressible, denotable and storeable values; languages can differ greatly in what they are their relationship to each other [Str]. Other classes of values – e.g. writeable ones – may also be of interest.

## 3.5 Exercises

**1.** It is possible to formalise the notion of occurrence. An occurrence is a sequence $l = m_1 \ldots m_n (n \geq 0)$ of nonzero integers. For any expression, $e$, (say in the first language of Chapter 3) and occurrence, $l$, one has the expression $e^l = \mathrm{Occ}(e, l)$ occurring in $e$ at $l$ (it may not be defined). For example

$$\mathrm{Occ}(e, \varepsilon) = e$$

$$\mathrm{Occ}(\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1, m \frown l) = \begin{cases} \mathrm{Occ}(x, l) & (m = 1) \\ \mathrm{Occ}(e_0, l) & (m = 2) \\ \mathrm{Occ}(e_1, l) & (m = 3) \\ \text{undefined} & (\text{otherwise}) \end{cases}$$

Define $\mathrm{Occ}(e, l)$ in general. Define $\mathrm{FO}(x, e) =$ the set of free occurrences of $x$ in $e$ and also the sets $\mathrm{AO}(x, e)$ and $\mathrm{BO}(x, e)$ of applied and binding occurrences of $x$ in $e$. For any $l$ in $\mathrm{BO}(x, e)$ define $\mathrm{Scope}(l) =$ the set of applied occurrences of $x$ in the scope of $l$; for any bound occurrence, $l$, of $x$ in $e$ (i.e. $l$ in $[\mathrm{AO}(x, e) \cup \mathrm{BO}(x, e)] \backslash \mathrm{FO}(x, e)$, define $\mathrm{binder}(l)$ the unique occurrence in whose scope $l$ is.

**2.** Repeat exercise 1 for the other languages in Chapter 3 (and later chapters!).

**3.** Ordinary mathematical language also has binding constructions. Notable are such examples as integration and summation.

$$\int_0^y \int_1^x f(n)\, dn\, dx \qquad \text{and} \qquad \sum_{n \geq 0} a_n x^n$$

Define mathematical expression language with these constructs and then define free variables and occurrences etc, just as in exercise 1.

**4.** The language of predicate logic also contains binders. Given a syntax for arithmetic expressions (say) we can define formulae by:

$$F ::= e = e \mid e > e \mid \ldots \mid \neg F \mid F \vee F \mid F \wedge F \mid F \supset F \mid \forall x F \mid \exists x F$$

where $\wedge, \vee, \supset$ mean logical and, or and implies and to assert $\forall x F$ means that for all $x$ have $F$ and to assert $\exists x F$ means that we have $F$ for some $x$. Repeat the work of exercise 3 for predicate logic. To what extent is it feasible to construct an operational semantics for the languages of exercise 3 and 4? How would it help to only consider finite sums, *display* $\sum_{a \leq n \leq b} e$ and quantifications $\forall x \leq b.F$ and piecewise approximation?

**5.** Can you specify the location of dynamic errors? Thus starting from $c, \sigma$ suppose we reach $c', \sigma'$ and the next action is (for example) division by zero; then we want to specify an error occurred as some occurrence in the original command $c$. [Hint, add a labelling facility, $c ::= L :: c$ and tradition rules for it, and start not from $c$ bur a labelled version in which the occurrences are used for labels].

**6.** Define the behaviour and equivalence of definitions and expressions of th second language of this chapter; prove that the program constructs respect equivalence. Establish of refute each of the following suggested equivalences

$$\begin{array}{lll} d_0 \ \mathbf{and}\ (d_1 \ \mathbf{and}\ d_2) & \equiv & (d_0 \ \mathbf{and}\ d_1) \ \mathbf{and}\ d_2 \\ d_0 \ \mathbf{and}\ d_1 & \equiv & d_1 \ \mathbf{and}\ d_2 \\ d_0 \ \mathbf{and\ nil} & \equiv & d_0 \\ d_0 \ \mathbf{and\ nil} & \equiv & \mathbf{nil} \end{array}$$

and similar ones for private and sequential definition.

**7.** Show that the following right-deftributive law

$$d_0 \ \mathbf{in}\ (d_1 \ \mathbf{and}\ d2) \quad \equiv \quad (d_0 \ \mathbf{in}\ d_1) \ \mathbf{and}\ (d_0 \ \mathbf{and}\ d_2)$$

holds. What about the left-distributive law? What about other such laws? Show that $d_0$ **in** $(x = e) \equiv x =$ **let** $d_0; d_1 \equiv d_0$ **in** $(d_1$ **and** $d_V)$ where $V = \mathrm{DV}(d_0)\backslash\mathrm{DV}(d_1)$ and where for any $V = \{x_1, \ldots, x_n\}$ we put $d_V = x_1 = x_1$ **and** $\ldots\ldots$ **and** $x_n = x_n$. Conclude that any $d$ can be put, to within equivalence, in the form $x_1 = e_1$ **and** $\ldots$ **and** $x_n = e_n$.

**8.** Show that **let** $d_0; d_1 \in e \ \equiv$ **let** $d_0$ **in** (**let** $d_1$ **in** $e$). Under what general conditions do we have $d_0; d_1 \ \equiv \ d_1; d_0$? When do wh have $d_0; d_1 \ \equiv \ d_0$ **in** $d_1$? When do we have **let** $d_0; d_1$ **in** $e \ \equiv$ **let** $d_0$ **in** $d_1$ **in** $d_0; e$?

**9.** It has been said that in blocks like **let** $d_0$ **in** $e$ all free variables of $e$ should be bound by $d$ for reasons of programming readibility. Introduce *strict* blocks **let** $d_0$ **in** $e$ and $d_0$ in $d_1$ where it is required that $\mathrm{FV}(e)$ (resp. $\mathrm{FV}(d_1)) \subseteq \mathrm{FV}(d_0)$. Show that the non-strict blocks are easily defined in terms of the strict ones. [Hint, use simultaneous definitions and the $d_V$ of exercise 7]. Investigate Equivalences for the strict constructions.

**10.** Two expressions (of the first language of the present chapter) $e$ and $e'$ are $\alpha$ - equivalent - written $e \equiv_\alpha e'$ - if they are identical "up to renaming of bound variables". For example

$$\mathbf{let}\ x = e\ \mathbf{in\ let}\ y = e'\ \mathbf{in}\ x + y \equiv_\alpha \mathbf{let}\ y = e\ \mathbf{in\ let}\ x = e'\ \mathbf{in}\ y + x$$

but **let** $x = e$ **in** $x + y \not\equiv_\alpha$ **let** $y = e$ **in** $y + y$. Define $\alpha$ - equivalence.[Hint, for a definition by structural induction to show **let** $x = e_0$ **in** $e_1 \equiv_\alpha$ **let** $y = e'_0$ **in** $e'_1$ it is necessary to show some relation between $e_1$ and $e'_1$. So define $\pi : e \equiv_\alpha e'$ where $\pi : \mathrm{FV}(e) \cong \mathrm{FV}(e')$ is a bijection; this relation means $e$ is $\alpha$ - equivalent to $e'$ up to the renaming, $\pi$, of the free variables]. Show that $e \equiv_\alpha e'$ implies $e \equiv e'$. Show that for any $e$ there is an $e'$ with $e \equiv_\alpha e'$ and no bound variable of $e'$ in some specified finite set and no variable of $e'$ has more than one binding occurrence.

**11.** Define for the first language of the present chapter the substitution of an expression $e$ for a variable $x$ in the expression $e'$ - written $[e/x]e'$; in the substitution process no free variable of $e'$ should be captured by a binding occurrence in $e'$, so that some systematic renaming of bound variables will be needed. For example we could not have

$$[x/y]\ \textbf{let}\ x = e\ \textbf{in}\ x + y = \ \textbf{let}\ x = [x/y]\ e\ \textbf{in}\ x + x$$

but could have

$$[x/y]\ \textbf{let}\ x = e\ \textbf{in}\ x + y = \ \textbf{let}\ z = [x/y]\ e\ \textbf{in}\ z + x$$

where $z \neq x$. Show the following

$$\begin{array}{lll}
\textbf{let}\ x = e\ \textbf{in}\ e' \ \equiv_\alpha\ \textbf{let}\ y = e\ \textbf{in}\ [y/x]e' & & (\text{if } y \notin \mathrm{FV}(e')) \\
{[e/x][e'/y]e''} \ \equiv_\alpha\ [[e/x]e'/y][e/x]e'' & & (\text{if } x \neq y) \\
{[e/x][e'/y]e''} \ \equiv_\alpha\ [[e/x]e'/x]e'' & & \\
{[e/x]e'} \ \equiv_\alpha\ e' & & (\text{if } x \notin \mathrm{FV}(e')) \\
\mathrm{FV}([e/x]e') \ = \mathrm{FV}(e) \cup (\mathrm{FV}(e') \backslash \{x\}) & & \\
{[e/x]e'} \ \equiv\ \textbf{let}\ x = e\ \textbf{in}\ e'. & &
\end{array}$$

**12.** By using substitution we could avoid the use of environments in the dynamic semantics of the first language of the present chapter. The transition relation would have the form $e \longrightarrow e'$ for *closed* $e, e'$ (no free variables) and the rules would be as usual for binary operations, none (needed) for identifiers, and $\textbf{let}\ x = e_0\ \textbf{in}\ e_1 \longrightarrow [e_0/x]e_1$. Show this gives the same notion of behaviour for closed expressions as the usual semantics.

**13.** Extend the work of exercises 10, 11 and 12 to the second language of the present chapter.

**14.** It is possible to have iterative constructs in applicative languages. Tennent has suggested the construct

$$e = \ \textbf{for}\ x = e_0\ \textbf{to}\ e_1\ \textbf{op}\ bop\ \textbf{on}\ e_2$$

So that for example if $e_0 = 0$ and $e_1 = n$ *op* $= +$ *and* $e_2 = x * x$ then $e = \sum\limits_{0 \leq x \leq n} x * x$.
Give the operational semantics of this construct.

**15.** It is even possible to use definitions to obtain analogues of while loops.Consider the definition construct

$$d = \textbf{while}\ e\ \textbf{do}\ d$$

So that

$$\begin{array}{l}
\textbf{let}\ \ \textbf{private}\, x = 1\ \mathit{and}\ y = 1 \\
\qquad \textbf{within}\ \ \textbf{while}\ y \neq n \\
\qquad\qquad\qquad \textbf{do}\ x = x * y\ \textbf{and}\ y = y + 1 \\
\textbf{in}\ x
\end{array}$$

computers $n!$ for $n \geq 1$. Give this construct a semantics; show that the construct of exercise 14 can be defined in terms of it. Is the new construct a "good idea"?

**16.** Consider the third language of the present chapter. Show that the type-environments generated by definitions are *determined* by defining by $SI$ a partial function $DTE$: Definitions $\longrightarrow$ TEnv and then proving that for any $\alpha, V, d, \beta$:

$$\alpha \vdash_V d : \beta \quad \Rightarrow DTE(d) \text{ is defined and equal to } \beta.$$

**17.** Give a semantics to a variant of the third language in which the types of variables are not declared and type-checking is dynamic.

**18.** Change the fourth language of the present chapter so that the atomic declarations have the more usual forms:

$$\textbf{const}\ x = e \qquad \text{and} \qquad \textbf{var}\ x : \tau$$

Can you type-check the resulting language? To what extent can you impose in the static semantics the requirement that variables should be initialised before use? Give an operational semantics following one of the obvious alternatives regarding initialisation aat declaration:

1. The variable is initialised to a conventional value (e.g. 0/ff), or an unlikely one (e.g. the maximum integer available/?).
2. The variable is not initialised at declaration. (*Hint*, use undefined maps for stores or (equivalently) introduce a special UNDEF vallue into the integers (and another for truthvallues). In this case show how to specify the error of access before initialisation. Which alternative do you prefer?

**19.** In PL/I identifiers canbe declared to be "EXTERNAL"; as such they take their value from an external environment - and so the declaration is an applied occurrence - but they have local scope - and so the declaration is also a binding occurrence. For example consider the following fragment in an extension of our fourth minilanguage (not PL/II) (where we allow $d ::= \textbf{external}\ x : \tau$):

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{external}\, x : \text{int};} \\
\multicolumn{4}{l}{\textbf{begin}} \\
 & x := 2; & & \\
; & \textbf{var}\ x : \text{int}; & & \\
 & \textbf{begin} & & \\
 & & x := 1; & \\
 & & \textbf{external}\, x : \text{int}; & \\
 & & \textbf{begin} & \\
; & & & y := x \\
 & & \textbf{end} & \\
 & \textbf{end} & & \\
\multicolumn{4}{l}{\textbf{end}}
\end{array}
$$

this sets $y$ equal to 2. Give a semantics to external declarations.

**20.** In PL/I variables can be declared without storage allocation begin make until explicitly requested. Thus a program fragment like

$$\begin{array}{ll} \textbf{var} & x : \mathrm{int} \\ \textbf{begin} & \\ & x := 1; \mathrm{allocate}(x) \\ \textbf{end} & \end{array}$$

would result in a dynamic error under that interpretation of variable declaration. Give a semantics to this idea.

**21.** In the programming language EUCLID it is possible to declare identifiers as *pervasive*, meaning that no holes are allowed in their scope - they cannot be redeclared within their scope. Formulate an extension of the imperative language of this chapter which allows pervasive declarations and give it a static semantics. Are they any problems with its dynamic semantics?

**22.** Formalise Dijkstra's ideas on scope as presented in section 10 of his book, A Discipline of Programming (Prentice-Hall, 1976). To do this define and give a semantics to a variant of the fourth minilanguage which incorporates his ideas in as elegant a way as you can manage.

**23.** Suppose we have two flavoours of variable declaration

$$\textbf{local var}\ x : \tau \qquad \text{and} \qquad \textbf{heap var}\ x : \tau$$

(cf PL/I, Algol 68). From an implementation point of view local variables are allocated space on the stack and heap ones on the heap; from a semantical point of view the locations are disposed of on block exit (i.e. they live until the end of the variable's scope is reached) or never (unless explicitly disposed of). Formalise the semantics for these ideas. Does replacing local by heap make any difference to a program's behaviour? If not, find some language extensions for which it does.

**24.** Add to the considerations of exercise 23 the possibility

$$\textbf{static var}\ x : \tau$$

Here, the locations are allocated as part of the static semantics (of FORTRAN / COBOL / PLI).

**25.** Consider the *finalisation* construct $d = d_0 \textbf{final}\ c$. Informally to elaborate this from an environment $\rho$ one elaborates $d_0$ obtaining $\rho_0$ but then *after* the actions (whether elaboration, execution or evaluation) involved in the scope of $d$ one executes $c$ in the environment $\rho' = \rho[\rho_0]$(equivalently, one executes $\rho'; ; c$). Give an operational semantics for an extension of the imperative language of the present chapter by a finalisation construct. [**Hint**, the elaboration of declarations should result in an environment and a command (with no free identifiers)]. Justify your treatment of the interaction of finalisation and the various compound definition forms.

**26.** How far can you go in treating the constructs of the imperative language of this chapter (or later ones) without using locations? One idea would be for declarations to produce couples $< \rho, \sigma >$ of environments and stores (in the sense of Chapter 2). where $\rho : l_1, \sigma : l_2$ and $l_1 \cap l_2 = \phi$. What problems arise with the declaration $x == y$?

**27.** Formalise the notion of accessibility of a location and of a *dangling* location by defining when given an environment $\rho$ and a configuration $\langle c, \sigma \rangle$ (or $\langle d, \sigma \rangle$ or $\langle e, \sigma \rangle$) a location, $l$, is accessible. Define the notion of *lifetime* with respect to the imperative language of the present chapter. Would it be best to define it so that the lifetime of a location ended exactly when it was no longer accessible or dangling? Using your definition formulate and prove a theorem, for the immmmperative language, relating scope and lifetime.

**28.** Locations can be considered as "dynamic place holders" (in the execution sequence) just as we considered identifiers as "static place holders"(in program text). Draw some arrow diagrams for locations in execution sequences to show their creation occurrences analogous to those drawn in this chapter to show binding occurrences.

**29.** Define $\alpha$-equivalence for the imperative programming language of the present chapter (see exercise 10). One can consider $c \equiv_\alpha c'$ as saying that $c$ and $c'$ are equivalent up to choice of static place holders. Define a relation of *location equivalence* between couples of environments and configurations, written $\rho, \gamma \equiv_l \rho', \gamma'$ (where $\gamma$ is an expression, command or declaration configuration); It should mean that the couples are equivalent uup to choice of locations (dynamic place holders). For example

$$\begin{array}{ll} \{x = l_1\}, \langle \{y = l_2\}; x := x + y, \{l_1 = 3, l_2 = 4\} \rangle & \equiv_l \\ \qquad \{x = l_2\}, \langle \{y = l_1\}; x := x + y, \{l_2 = 3, l_1 = 4\} \rangle & \end{array}$$

holds.

**30.** Define the behaviour of commands, expressions and declarations and define an equivalence relation $\equiv_l$ between behaviours which should reflect equality of behaviours up to choice of dynamic place holders. Prove, for example, that

$$(\mathbf{var}\ x : int = 1; \mathbf{var}\ y : int = 1) \quad \equiv_l \quad (\mathbf{var}\ y : int = 1; \mathbf{var}\ x : int = 1)$$

even although the two sides do not have identical behaviours. Investigate the issues of exercises 10, 11, and 12 using $\equiv_l$.

## 3.6 Remarks

The idea of structuring definitions and declarations seem to go back to Landin [Lan] and Milne and Strachey [Mil]. The idea of separating environments and stores, via locations, can also be found in [Mil]. The concepts of scope, extent, environments, stores and their mathematical formulations seem to be due to Burstall, Landin, McCarthy, Scott and Strachey. [I do not want to risk exact credits, or exclude others ... ] For another account of these matters see [Sto]. The ideas of 3.5 on static semantics where the constraints are clearly con-text-sensitive in general were formulated in line with the general ideas on dynamic semantics. In fact, they are simpler as it is only needed to establish properties of phrases rather than having relations between them. lt is hoped that the method is easy to read and in line with one's intuition. There are many other methods for the purpose and for a survey with references, see [Wil]. It is also possible to use the techiques of denotational semantics for this purpose [Gor, Sto]. Our method seems particularly close to the production systems of Ledgard and the extended attribute grammars used by Watt; one can view, in such formulae as $\alpha \vdash_V d : \beta$, the turnstile symbols $\alpha$ and $V$ as inherited attributes and $\beta$ as a synthesized attribute of the definition $d$; obviously too the type-environments $\alpha$ and $\beta$ are nothing but symbol tables. It would be interesting to compare the methods on a formal basis. As pointed out in exercise 26 one can go quite far without using locations. Donahue also try to avoid them in [Don]. In a first version of our ideas we also avoided them, but ran into unpleasantly complicated systems when considering shared global variables of function bodies. As pointed out in exercise 12 one can try to avoid environments by using substitutions; it is not clear how far one can go in this direction (which is the usual one in syntactic studies of the $\lambda$-calculus). However, we have made a definite decision in these notes to stick to the Scott-Strachey tradition of environments. Note that in such rules as

$$\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 \longrightarrow [e_0/x]e_1$$

there is no offense against the idea of syntax-directed operational semantics. It is just that substitution is a rather "heavy" primitive and one can argue that the use of environments is closer to the intuitions normally used for understanding programming languages. (One awful exception is the Algol-60 call-by-name mechanism).

## 3.7 Bibliography


[Ack] W.B. Ackerman (1979) *Data Flow languages* Proc. 1979, AFIPS

[Don] J.E. Donahue (1977) *Locations Considered Unnecessary* Acta Inf. 8, pp.221-242

[Gor1] M. Gordon, R. Milner and C. Wadsworth (1979) *Edinburgh LCF*, Vol. 78, SLNCS

[Gor2] M. Gordon (1979) *The Denotational description of Programming Languages* Springer-Verlag

[Hin] J.R. Hindley, B. Lercher and J.P. Seldin (1974) *introduction to Combinatry Logic* CUP

[Hug] G.E. Hughes and M.J. Cresswell (1977) *An introduction to Modal Logic*

[Lan1] P.J. Landin (1966) *The Mechanical Evalution of Expressions Computer Journal* Vol. 6, pp. 308-320

[Lan2] P.J. Landin (1965) *A Correspondence between Algol 60 and Church7s Lamdba-Notation* CACM, Vol. 8

[Led] H. Ledgard and M. Marcotty (1981) *The Programming Language Landscape Science Research Associates*

[Mil] R.E. Milne and C. Strachey (1976) *A Theory of Programming Language Semantics* Chapman and Hall

[Pra] D. Prawitz (1971) *Ideas and Results in Proof Theory* Proc. 2nd Scand. Logic Congress, North Holland

[Rey] J.C. Reynolds (1978) *Syntactic Control of Interference* 5th POPL. ACM, pp. 39-46

[Str] C. Strachey (1974) *The Varieties of Programming Languages* PRG-10, Prog. Res. Group, University of Oxford

[Sto] J.E. Stoy (1977) *Denotational Semantics* MIT Press

[Wil] M.H. Williams (1981) *Methods for Specifying Static Semantics Computer Languages* Vol.6, pp. 1, 17

# Chapter 4

# Functions, Procedures and Classes

In this chapter we consider various mechanisms allowing various degrees of abbreviation and abstraction in programming languages. The idea of abbreviating the repeated use of some expressions by using definitions or declarations of identifiers was considered in chapter three; if we apply the same choice to commands we arrive at (parameterless) procedures (= subroutines). It is very much more useful to abstract many similar computations together, different ones being obtained by varying the values of *parameters*. In this way we obtain functions from expressions and procedures from commands.

Tennent's *Principle of Abstraction* declares that the same thing can be done with any semantically meaningful category of phrases. Applying the idea to definitions of declarations we obtain a version of the *class* concept, introduced by SIMULA and recently taken up in many modern programming languages. (If we just use identifiers to stand for definitions or declarations we obtain the simpler but still most useful idea of *module*).

*Calling* (= invoking) abstractions with *actual* parameters (their *arguments*) for the *formal* ones appearing in their definition results in appropriate computations whether evaluations, executions or elaborations of the *bodies* of their definitions. We will explain this by allowing abstraction identifiers to denote *closures* which record their formal parameters and bodies. Invocations will be explained in terms of computations of blocks chosen in terms of Tennent's Principle of Correspondence which declares that in principle to every parameter mechanism there corresponds an appropriate definition or declaration mechanism. For example if we define

$$f(x : \mathrm{int}) : \mathrm{in} = x + 1$$

then the elaboration results in the environment

$$f = \lambda x : int.\, x + 1 : \mathrm{int}$$

To invoke $f$ in an expression, say $f(5)$, we just evaluate the expression block

$$\begin{array}{l} \mathbf{let}\, x : \mathrm{int} = 5 \\ \mathbf{in}\ \ x + 1 \end{array}$$

Note that his block exists by virtue of Tennent's Principle of Qualification.

Below we use these ideas to consider an applicative programming language with (possibly recursive) definitions of functions of several arguments. We then consider an imperative language where we consider both functions and procedures and use the Principle of Correspondence to obtain the parameter mechanisms of call-by-constant and call-by-value. Other parameter mechanisms are easily handled using the same ideas (some explicitly in the text and others in exercises); let us mention call-by-reference, call-by-result, call-by-value-result, call-by-name and call-by-text. Next we consider higher order functions and procedures. Finally we use the Principles of Abstraction and Correspondence to handle modules and classes; this needs no new ideas although some of the type-checking issues are interesting.

## 4.1 Functions in Applicative Languages

We begin with the simplest case where it is possible to define functions of one argument (unary) functions. Let us consider throughout extensions of the second applicative language of Chapter 3. Add the following kind of function definitions:

$$d ::= f(x : \tau_0) : \tau_1 = e$$

and function calls

$$e ::= f(e)$$

where $f$ is another letter we will use to range over variables (but reserving its use to contexts where functions are expected).

### Static Semantics

This is just as before as regards free and defining variables with the extensions

$$\begin{array}{l} \mathrm{FV}(f(x : \tau_0) : \tau_1 = e) = \mathrm{FV}(e)\backslash\{x\} \\ \mathrm{DV}(f(x : \tau_0) : \tau_1 = e) = \{f\} \\ \mathrm{FV}(f(e)) = \{f\} \cup \mathrm{FV}(e) \end{array}$$

It is convenient to consider types a little more systematically than before. Just as we have expressive and denotable value (EVal and DVal) we now introduce the sets of ETypes and Dtypes, *expressible and denotable types* (ranged over by $et$ and $dt$ respectively) where

$$\begin{array}{l} et ::= \tau \\ de ::= \tau \mid \tau_0 \rightarrow \tau_1 \end{array}$$

More complex expressible types will be needed later; denotable types of the form $\tau_0 \rightarrow \tau_1$ will be used for functions which take arguments of type $\tau_0$ and deliver *results* of type $\tau_1$. Later we will want also sets of *storeable* types and other such sets. Now we take

$$\mathrm{Tenv} = \mathrm{Var} \longrightarrow_{\mathrm{fin}} \mathrm{DTypes}$$

ranged over, as before, by $\alpha$ and $\beta$ and give rules for the predicates

$$\alpha \vdash_V e : et$$

where $\alpha : V$ and $\mathrm{FV}(e) \subseteq V$, and

$$\alpha \vdash_V d : \beta$$

where $\alpha : V$ and $\mathrm{FV}(d) \subseteq V$. These rules are just as before with the evident extensions for function calls and definitions:

**Function Call:** $$\frac{\alpha \vdash_V e : et_0}{\alpha \vdash_V f(e) : et_1} \quad (\text{if } \alpha(f) = et_0 \to et_1)$$

**Function Definitions:** $$\frac{\alpha \vdash_V e : \tau_1}{\alpha \vdash_V f(x : \tau_0) : \tau_1 = e : \{\tau_0 \to \tau_1\}}$$

**Dynamic Semantics**

We introduce the set, Closures, of closures (ranged over by $d$)

$$\mathrm{Closures} = \{\lambda x : et_0 . e : et_1 \ \mid\ \{x = et_0\} \vdash_{\{x\}} e : et_1\}$$

and define the set of denotable values by

$$\mathrm{DVal} = \mathrm{Con} + \mathrm{Closures}$$

and then we define, as usual,

$$\mathrm{Env} = \mathrm{Var} \longrightarrow_{\mathrm{fin}} \mathrm{DVal}$$

and add the following production to the definition for the category of definitions

$$d ::= \rho$$

(and put for $\rho : V$, $\mathrm{DV}(\rho) = V$ and $\mathrm{FV}(\rho) = \emptyset$).

It is important to note that what is meant here is that the sets Dec, Exp, Closures, DVal and Env are begin defined mutually recursively. For example the following is an integer expression

$$\begin{array}{l}
\mathbf{let}\, f = \lambda x : \mathit{int} \\
\qquad (\mathbf{let}\{y = 3, g = \lambda y : \mathrm{bool}.\ {\sim} y : \mathrm{bool}\} \\
\qquad \mathbf{in\ if}\ g(ff)\ \mathbf{then}\ x\ \mathbf{else}\ y) : \mathit{int} \\
\quad \mathbf{and} w = 5 \\
\mathbf{in}\ \ f(2) + w
\end{array}$$

There is no more harm in such recursions than in those found in context-free grammars; a detailed discussions is left to elsewhere.

Note to that closures have in an obvious sense on free variables. This raises the puzzle of what we intend to do about the free variable in function definitions. In fact in elaborating such definitions we will bind the free variables to their values in the elaboration environment. This is known as *static binding* (= binding of free variables determined by their textual occurrence), and will be followed throughout these ones. The alternative of delaying binding until the function is called, and then using the calling environment, is known as *dynamic binding*, and is considered in the exercises.

To extend the static semantics we type denotable values defining the predicate for *dval* in DVal ad *dt* in DType

$$dval : dt$$

and for $\rho : V$ in Env and $\alpha : V$ in TEnv define

$$\rho : \alpha$$

by the rules

**Constants:** $m : \mathrm{int} \qquad t : \mathrm{bool}$

**Closures:** $(\lambda x : et_0.\ e : et_1) : et_0 \to et_1$

**Environments:** $\dfrac{\forall x \in V.\ \rho(x) : \alpha(x)}{\rho : \alpha} \qquad (\text{where } \rho : V,\ \alpha : V)$

and add the rule for environments considered as definitions

**Environments:** $\dfrac{\rho : \alpha}{\alpha \vdash_V \rho : \beta}$

With all this we now easily extend the old dynamic semantics with the usual transition relations

$$\rho \vdash_\alpha e \longrightarrow e'$$
$$\rho \vdash_\alpha d \longrightarrow d'$$

by rules for function calls and definition.

- **Function Calls**:

  $$\rho \vdash_\alpha f(e_0) \longrightarrow \mathbf{let}\ x : et_0 = e_0\ \mathbf{in}\ e \qquad (\text{if } \rho(f) = \lambda x : et_0.\ e : et_1\ )$$

  This rule is just a formal version of the Principle of Correspondence for the language under consideration.

- **Function Definition**:

  $$\rho \vdash_\alpha f(x : \tau_0) : \tau_1 = e \longrightarrow \{f = \lambda x : \tau_0.\ (\mathbf{let}\ \rho\lceil V \mathbf{in} e) : \tau_1\} \qquad (\text{where } V = \mathrm{FV}(e)\backslash\{x\})$$

**Example**: We write $f(x : \tau_0) : \tau_1 = e$ for the less readable $f = \lambda x : \tau_0.\ e : \tau 1$ (and miss out $\tau_0 and/or \tau_1$ when they are obvious). Consider the expression

$$\begin{array}{l} e \stackrel{def}{=} \textbf{let}\, \text{double}(x : \text{int}) : \text{int} = 2 * x \\ \quad\ \ \textbf{in}\ \text{double}(\text{double}(2)) \end{array}$$

we have

$$\emptyset \vdash_{\emptyset} e \longrightarrow \textbf{let}\ \rho \in \text{double}(\text{double}(2))$$

where $\rho \stackrel{def}{=} \{\text{double}(x) = 2 * x\}$ and now note the computation

$$\begin{array}{l} \rho \vdash \text{double}(\text{double}(2)) \longrightarrow \textbf{let}\ x : \text{int} = \text{double}(2)\ \textbf{in}\ \text{double}(2) \\ \quad \longrightarrow\ \ \longrightarrow \textbf{let}\ x : \text{int} = (\textbf{let}\ x\ \text{int} = 2\ \textbf{in}\ 2 * x) \\ \qquad\quad \textbf{in} 2 * x \\ \quad \longrightarrow^3 \textbf{let}\ x : \text{int} = 4\ \textbf{in}\ 2 * x \\ \quad \longrightarrow^3 8 \end{array}$$

and so

$$\emptyset \vdash e \longrightarrow^* 8$$

Our function calls are *call-by-value* in the sense that the argument is evaluated before the body of the function. On the other hand it is evaluated just after the function call; a slight variant effects the evaluation before.

- **Function Call (Amended)**

  1. $$\frac{\rho \vdash e \longrightarrow e'}{\rho \vdash_V f(e) \longrightarrow f(e')}$$
  2. $\rho \vdash f(\text{con}) \longrightarrow \textbf{let}\ x : \tau_0 = \text{con}\ \textbf{in}\ e \qquad (\text{if } f(x : \tau_0) = e \text{ is in } \rho)$

This variant has no effect on the result of our computations (prove this!) although it is not hard to define imperative languages where there could be a difference (because of side-effects). Another important possibility – call-by-name – is considered below and in the exercises.

We now consider how to extend the above to definitions of functions of several arguments such as

$$\max(x : \text{int}, y : \text{int}) : \text{int} = \textbf{if}\ x \geq y\ \textbf{then}\ x\ \textbf{else}\ y$$

Intending to use the Principle of Correspondence to account for function calls we expect such transitions as

$$\rho \vdash \max(3, 5) \longrightarrow \begin{array}{l} \textbf{let} x : \text{int}, y : \text{int} = 3, 5 \\ \textbf{in if}\ x \geq y\ \textbf{then}\ x\ \textbf{else}\ y \end{array}$$

and there fore simultaneous simple definitions. To this end we adopt a "minimalist" approach adding two syntactic classes to the applicative language of the last chapter.

**Formals**: This is the set Forms ranged over by $form$ and given by

$$form ::= \cdot \mid x : \tau, form$$

**Actual Expressions**: This is the set $AcExp$ ranged over by $ae$ where

$$ae ::= \cdot \mid e, ae$$

Then we extend the category of definitions allowing more simple definitions and function definitions

$$d ::= form = ae \mid f(form) : \tau = e$$

and adding function calls to the stock of expressions

$$e ::= f(ae)$$

To obtain a conventional notation $x : \tau, \cdot$ and $e, \cdot$ are written $x : \tau$ and $e$ respectively and $f()$ replaces $f(\cdot)$. In a "maximalist" solution we could include actual expressions as expressions and allow corresponding "tuple" types and as types of identifiers and function results; see exercise 2.

### Static Semantics

Formals give rise to defining variable occurrences

$$\mathrm{DV}(\cdot) = \emptyset \qquad \mathrm{DV}(x : \tau, form) = \{x\} \cup \mathrm{DV}(form)$$

Then we have free variables in actual expressions

$$\mathrm{FV}(\cdot) = \emptyset \qquad \mathrm{FV}(e, ad) = \mathrm{FV}(e) \cup \mathrm{FV}(ae)$$

and for the new kinds of definitions

$$\begin{aligned}&\mathrm{FV}(form = ae) = \mathrm{FV}(ae) \qquad \mathrm{DV}(form = ae) = \mathrm{DV}(form)\\ &\mathrm{FV}(f(form) : \tau = e) = \mathrm{FV}(e) \backslash \mathrm{DV}(form)\\ &DV(f(form) : \tau = e) = \{f\}\end{aligned}$$

and for function calls, $\mathrm{FV}(f(ae)) = \{f\} \cup \mathrm{FV}(ae)$.

Turning to types we now have ETypes, AcETypes (ranged over by $aet$) and DTypes where

$$et ::= \tau \qquad aet ::= \cdot \mid \tau, aet \qquad dt ::= et \mid aet \to et$$

Then with TEnv = Var $\longrightarrow_{\text{fin}}$ DTypes as always we have the evident predicates

$$\alpha \vdash_V e : et \qquad \alpha \vdash_V ae : aet \qquad \alpha \vdash_V d : \beta$$

Formals give positional information and type environments. So we define $T$ : Formals $\longrightarrow$ AcETypes by

$$\mathrm{T}(\cdot) = \cdot \qquad \mathrm{T}(x : \tau, form) = \tau, \mathrm{T}(form)$$

and give rules for the predicate $form : \beta$

1. $\cdot : \emptyset$

2. $form : \beta \quad \Rightarrow \quad (x : \tau, form) : \{x = \tau\}, \beta(\text{if} x \notin \mathrm{DV}(form))$

Note that it is here the natural restriction of no variable occurring twice in a formal is made.

Here are the rules for the other predicates:

**Function Calls:** $\dfrac{\alpha \vdash_V ae : aet}{\alpha \vdash_V f(ae) : et} \qquad (\text{if } \alpha(f) = aet \to et)$

**Definitions:** $\dfrac{form : \beta \ \alpha \vdash_V ae : aet}{\alpha \vdash_V (form = ae) : \beta} \qquad (\text{where } aet = T(form))$

$$\frac{form : \beta \ \alpha[\beta] \vdash_{V \cup V_0} e : t}{\alpha \vdash_V (f(form) : \tau = e) : \{f = aet \longrightarrow \tau\}}$$
(where $\beta : V_0 \text{and} aet = T(form)$)

**Actual Expr.:** $\alpha \vdash_V \cdot : \cdot$

$$\frac{\alpha \vdash_V e : et \quad \alpha \vdash_V ae : aet}{\alpha \vdash_V e, ae : et, aet}$$

**Dynamic Semantics**

We proceed much as before as regards closures, denotable values and environments

$$\begin{aligned}
&\text{Closures} = \{\lambda form.\ e : et \mid \exists \beta V.\ form : \beta \text{ and } \beta : V \text{ and } \beta \vdash_V e : et\} \\
&\text{DVal} = \text{Con} + \text{Closures} \\
&\text{Env} = \text{Var} \longrightarrow_{\text{fin}} \text{DVal} \\
&d ::= \rho
\end{aligned}$$

with the free and defining variables of $\rho$ as usual and extend the static semantics by defining the predicates $dval : dt$ and $\rho : \alpha$ much as before.

As regards transition rules we will naturally define $\rho \vdash_V e \longrightarrow e'$ and $\rho \vdash_V d \longrightarrow d'$ and, for actuals, $\rho \vdash_V ae \longrightarrow ae'$. The terminal actual configurations are the "actual constants"-tuples of constants given by the rules

$$acon ::= \cdot \mid con, acon$$

As for formals they give rise to environments in the content of a value for the corresponding actuals and so we begin with rules for the predicate

$$acon \vdash form : \rho$$

1. $\cdot \vdash \cdot : \emptyset$

2. $\dfrac{acon \vdash form : \rho}{con, acon \vdash (x : \tau, form) : \rho \cup \{x = con\}}$

While this is formally adequate enough it does seem odd to use values rather than environments as dynamic contexts.

The other rules should now be easy to understand.

**Function Calls:** $\rho \vdash_\alpha f(ae) \longrightarrow \textbf{let}\ form = ae\ \textbf{in}\ e \qquad (\text{if } \rho(f) = \lambda form.\ e : et)$

**Definitions Simple:**

$$\frac{\rho \vdash_\alpha ae \longrightarrow ae'}{\rho \vdash form = ae \longrightarrow form = ae'}$$

$$\frac{acon \vdash form : \rho_0}{\rho \vdash form = acon \longrightarrow \rho_0}$$

**Function:** $\rho \vdash_\alpha f(form) : \tau = e \longrightarrow \{f = \lambda form\ \textbf{let}\ \rho\lceil V\ \textbf{in}\ e : \tau\}$
(where $V = \mathrm{FV}(e) \backslash \mathrm{DV}(form)$)

**Actual Expr.:** $\rho \vdash_\alpha e \longrightarrow e' \quad \Rightarrow \quad \rho \vdash_\alpha e, ae \longrightarrow e', ae$
$\rho \vdash_\alpha ae \longrightarrow ae' \quad \Rightarrow \quad \rho \vdash_\alpha con, ae \longrightarrow con, ae'$

**Example**: We calculate the maximum of $2+3$ and $2 \times 3$. Let $\rho_0 = \{\max(x : \mathrm{int}, y : \mathrm{int}) : in = \textbf{let}\ \emptyset\ \textbf{in}\ \textbf{if} x \geq y \textbf{then} x \textbf{else} y\}$. Then we have

$$\begin{aligned}
&\emptyset \vdash \{\textbf{let}\ \max(x : \mathrm{int}, y : \mathrm{int}) : \mathrm{int} =\ \textbf{if}\ x \geq y\ \textbf{then}\ x\ \textbf{else}\ y\}\ \textbf{in}\ \max(2+3, 2*3)\\
&\quad \longrightarrow \textbf{let}\ \rho_0\ \textbf{in}\ \max(2+3, 2*3)\\
&\quad \longrightarrow \textbf{let}\ \rho_0\ \textbf{in}\ \textbf{let}\ x : \mathrm{int}, y : \mathrm{int} = 2+3, 2 \times 3\ \textbf{in}\ \textbf{let}\ \emptyset\ \textbf{in}\ (\textbf{if}\ x \geq y\ \textbf{then}\ x\ \textbf{else}\ y)\\
&\quad \longrightarrow^* \textbf{let}\ \rho_0\ \textbf{in}\ \textbf{let}\ \{x = 5, y = 6\}\ \textbf{in}\ \textbf{let}\ \emptyset\ \textbf{in}\ (\textbf{if}\ x \geq y\ \textbf{then}\ x\ \textbf{else}\ y\\
&\quad \longrightarrow^* \textbf{let}\ \rho_0\ \textbf{in}\ \textbf{let}\ \{x = 5, y = 6\}\ \textbf{in}\ \textbf{let}\ \emptyset\ \textbf{in}\ 6\\
&\quad \longrightarrow^3 6.
\end{aligned}$$

as one sees that

$$\rho_0 \vdash x : \mathrm{int}, y : \mathrm{int} = 2+3, 2 \times 3 \longrightarrow^* \{x = 5, y = 6\}$$

### Recursion

It will not have escaped the readers attention that no matter how interesting our applicative language maybe it is useless as there is no ability to prescribe interesting computations. For example we do not succeed in defining the factorial function by

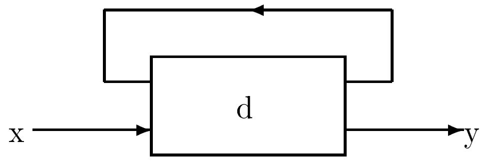
$$d \stackrel{fact}{=} fact(n : \mathrm{int}) : \mathrm{int} = \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times fact(n-1)$$

as the "fact" on the right will be taken from the environment of $d_{fact}$ and not understood recursively. (Of course the imperative languages are interesting owing to the possibility of loops; note too exercise 3, 14, 15.

Clearly we need to introduce recursion. Syntactical we just postulate a unary operator on definitions (and later on declarations)

$$d ::= \textbf{rec} d$$

Thus $\textbf{recd}_{\textbf{fact}}$ will define the factorial function. In terms of imports and exports $\textbf{rec}d$ imports all imports of $d$ other than exports which provide the rest of the imports to $d$; the exports of $\textbf{rec}d$ are those of $d$. In other words $\textbf{let} X = \mathrm{FV}(d) \backslash \mathrm{DV}(d) and Y = \mathrm{DV}(d) and R = \mathrm{FV}(d) \cap \mathrm{DV}(d)$. Then $X$ is the set of imports of $\textbf{rec}d$ and $Y$ is the set of its exports with $R$ begin defined recursively. Diagrammatically we have




A Recursive Declaration: **rec** $d$

The unary recursion operator gives a very flexible way to make recursive definitions since the $d$ in **rec**$d$ can take many forms other than simple function definitions like $f(x : \tau_1 \ldots) : \tau = e$. *Simultaneous recursive* definitions are written

$$\begin{array}{ll}\mathbf{rec} & f(\ldots) = \ldots f \ldots g \ldots \mathbf{and} \ldots \mathbf{and} \\ & g(\ldots) = \ldots f \ldots g \ldots\end{array}$$

A *narrow scope* form of sequential recursive definitions is

$$\begin{array}{l}\mathbf{rec}\ f(\ldots) = \ldots f \ldots g \ldots\ ;\ \ldots\ ; \\ \mathbf{rec}\ g(\ldots) = \ldots f \ldots g \ldots\ ;\end{array}$$

where the $g$ in the definition of $f$ is taken from the environment but the $f$ in the definition of $g$ is the recursively defined one. A *wide scope* form is obtained by writing

$$\begin{array}{ll}\mathbf{rec} & f(\ldots) = \ldots f \ldots g \ldots\ ; \ldots\ ; \\ & g(\ldots) = \ldots f \ldots g \ldots\end{array}$$

which is equivalent to the simultaneous definition unless $f = g$ for example.

### Static Semantics

For free and defining variables we note that

$$\begin{array}{l}\mathrm{FV}(\mathbf{rec} d) = \mathrm{FV}(d) \backslash \mathrm{DV}(d) \\ \mathrm{DV}(\mathbf{rec} d) = \mathrm{DV}(d)\end{array}$$

We keep TEnv and DTypes, ETypes and AcETypes as before. The natural rule for recursive declarations is

$$\frac{\alpha[\beta\lceil V_0] \vdash_{V \cup V_0} d : \beta}{\alpha \vdash_V \mathbf{rec} d : \beta} \qquad (\text{where } V_0 = \mathrm{FV}(d) \cap \mathrm{DV}(d))$$

However, this is not easy to use in a top-down fashion as given **rec**$d$ and $\alpha$ one would have to guess $\beta$. But, as covered by exercise 11, it would work. It is more convenient to use the fact that in $\alpha \vdash_V d : \beta$ the elaborated $\beta$ does not depend on $\alpha$ but is uniquely determined by $d$, the $\alpha$ only begin used to check the validity of $\beta$. We make this explicit by defining **two** predicates for definitions. First for any $V$ and $d$ with $\mathrm{FV}(d) \subseteq V$ and $\beta$ we define

$$\vdash V d : \beta$$

and secondly for any $\alpha : V$ and $d$ with $\mathrm{FV}(d) \subseteq V$ we define

$$\alpha \vdash_V d$$

The first predicate can be read as saying that if $d$ is a valid definition then it will have type $\beta$; the second says that given $\alpha$ then $d$ is valid. The other predicates will be as before

$$\alpha \vdash_V e : et \qquad \alpha \vdash_V ae : aet \qquad form : \beta$$

**Rules**:

- **Definitions:**

**Nil:**

1. $\vdash_V \mathbf{nil} : \emptyset$
2. $\alpha \vdash_V \mathbf{nil}$

**Simple:**

1. $\dfrac{form : \beta}{\vdash_V form = ae : \beta}$
2. $\dfrac{form : \beta \ \alpha \vdash_V ae : T(form)}{\alpha \vdash_V form = ae}$

**Function:**

1. $\vdash_V f(form) : \tau = e : T(form) \longrightarrow \tau$
2. $\dfrac{form : \beta \ \alpha[\beta] \vdash_{V \cup V_0} e : \tau}{\alpha \vdash_V f(form) : \tau = e}$ (where $\beta : V_0$)

**Sequential:**

1. $\dfrac{\vdash_V d_0 : \beta_0 \ \vdash_V d_1 : \beta_1}{\vdash_V d_0; d_1 : \beta_0[\beta_1]}$
2. $\dfrac{\alpha \vdash_V d_0 \ \vdash_V d_0 : \beta \ \alpha[\beta] \vdash_{V \cup V_0} d_1}{\alpha \vdash_V d_0; d_1}$ (where $\beta : V_0$)

**Simultaneous:**

1. $\dfrac{\vdash_V d_0 : \beta_0 \ \vdash_V d_1 : \beta_1}{\vdash_V d_0 \ \mathbf{and} \ d_1 : \beta_0, \beta 1}$
2. $\dfrac{\alpha \vdash_V d_0 \ \alpha \vdash_V d_1}{\alpha \vdash_V d_0 \ \mathbf{and} \ d_1}$ (if $\mathrm{DV}(d_0) \cap \mathrm{DV}(d_1) = \emptyset$)

**Private:**

1. $\dfrac{\vdash_V d_1 : \beta_1}{\vdash_V d_0 \mathbf{in} d_1 : \beta_1}$
2. $\dfrac{\alpha \vdash_V d_0 \ \vdash_V d_0 : \beta_0 \ \alpha[\beta_0] \vdash_{V \cup V_0} d_1}{\alpha \vdash_V d_0 \mathbf{in} d_1}$ (where $\beta_0 : V_0$)

**Recursion:**

1. $\dfrac{\vdash_V d : \beta}{\vdash_V \mathbf{rec} d : \beta}$
2. $\dfrac{\vdash_{V \cup V_0} d : \beta \ \alpha[\beta \lceil V_0] \vdash_{V \cup V_0} d}{\alpha \vdash_V \mathbf{rec} d}$ (where $V_0 = \mathrm{FV}(d) \cap \mathrm{DV}(d)$)

The other rules are as before except for expression blocks:

$$\frac{\vdash_V d : \beta \ \alpha \vdash_V d \alpha[\beta] \vdash_{V \cup V_0} e : et}{\alpha \vdash_V \mathbf{let} d \mathbf{in} e : et} \qquad \text{(where } \beta : V_0\text{)}$$

**Example**: Consider the definition

$$\begin{aligned} d = \mathbf{rec}\, f(x:\mathrm{int}) : \mathrm{int} &= g(x)\mathbf{and} \\ g(x:in)\ \ : \mathrm{int} &= f(x) \end{aligned}$$

Here as $\vdash_{f,g} f(x:\mathrm{int}) : \mathrm{int} = g(x) : \{f = \mathrm{int} \to \mathrm{int}\}$, etc. we have

$$\vdash_{\emptyset} d : \{f = \mathrm{int} \to \mathrm{int}, g = \mathrm{int} \longrightarrow \mathrm{int}\}.$$

Then to see that $\emptyset \vdash_{\emptyset} d$ one just shows that $\{f = \mathrm{int} \to \mathrm{int}, g = \mathrm{int} \to \mathrm{int}\} \vdash_{f,g} d_0$ (Where $\mathbf{rec} d_0 = d$). This example also shows why it is needed to explicitly mention the result (= output) of functions.

## Dynamic Semantics

Before discussing our specific proposal we should admit that this seems, owing to a certain clumsiness and its somewhat unnatural approach, to be a possible weak point in our treatment of operational semantics.

At first sight one wants to get something of the following effect with recursive definitions

$$\frac{\rho[\rho_0\lceil V_0] \vdash_{\alpha\cup\alpha_0} d \longrightarrow^* \rho_0}{\rho \vdash_{\alpha} \mathbf{rec} d \longrightarrow^* \rho_0} \qquad \text{(where } \rho_0 : \mathrm{DV}(d) \text{ and for suitable } \alpha_0 : V_0)$$

Taken literally this is not possible. For example put $d = f(x : \mathrm{int}) : \mathrm{int} = f(x)$ and suppose $\rho_0(f) = d$. Then for $V = \emptyset$ and $\rho \longrightarrow \emptyset$ we would have

$$\rho_0 \vdash_{\{f\}} f(x:\mathrm{int}) : \mathrm{int} = f(x) \longrightarrow \{f = \lambda x : \mathrm{int}(\mathbf{let}\rho_0\mathbf{in} f(x)) : \mathrm{int}\}$$

and so we would have $d = \lambda x : \mathrm{int}.\ (\mathbf{let}\rho_0\mathbf{in} f(x)) : \mathrm{int}$ which is clearly impossible as $d$ cannot occur in itself (via $\rho_0$). Of course it is just in finding solutions to suitable analogues of this equation that the Scott-Strachey approach finds one of its major achievements.

Let us try to overcome the problem by not trying to guess $\rho_0$ but trying to elaborate $d$ without any knowledge of the values of the recursively defined identifiers. Thus in our example we first elaborate the body

$$\emptyset \vdash_{\emptyset} f(x:\mathrm{int}) : \mathrm{int} = f(x) \longrightarrow \{f = \lambda x : \mathrm{int}.\ (\mathbf{let}\emptyset f(x)) : \mathrm{int}\}$$

and let $\rho_0$ be the resulting "environment". Note that we no longer have closures as there can be free variables in the abstractions. So we know that for any imported value of $f$ that $\rho_0$ gives the corresponding export. But in $\mathbf{rec} d$ the imports and the exports must be the same, that is $f = \rho(f)$ in some recursive sense and we can take $f \stackrel{def}{=} \mathbf{rec}\rho_0$. To get a closure we now take the all important step of binding $f$ to $\mathbf{rec}\rho_0$ in $\rho_0$ and take the elaboration of $\mathbf{rec} d$ to be

$$\rho_1 = \{f(x:\mathrm{int}) : \mathrm{int} = \mathbf{let}\ \mathbf{red}\rho_0\ \mathbf{in}\ (\mathbf{let}\ \emptyset\ \mathbf{in} f(x)) : \mathrm{int}\}$$

What we have done is unwound the recursive definition by one step and bound into the body instructions for further unwinding. Indeed it will be the case that

$$\vdash \mathbf{rec}\rho_0 \longrightarrow \rho_1$$

and so when we *call* $f(e)$ we will arrive at the expression

$$\begin{array}{l}\mathbf{let}\ x : \mathrm{int} = e \\ \ \mathbf{in}\ \mathbf{let}\ rec\rho_0 \\ \quad\ \mathbf{in}\ \mathbf{let}\ \emptyset \\ \qquad \mathbf{in}\ f(x) : \mathrm{int}\end{array}$$

Then we will evaluate the argument $e$, then we will unwind the definition once more (in preparation for the next call!), then we will evaluate the body. This is perhaps not too bad; in the usual operational semantics of recursive definitions (see exercise 7) one first evaluates the argument, then unwinds the definition for the *present* call and then evaluates the body. Thus we have simply performed in advance one step of the needed unwindings during the elaboration. Some further comments on this are given in exercise

Let us now turn our attention to the formal details, the changes form previously mostly concern allowing free variables in closures, and we define

$$\mathrm{Abstracts} = \{\lambda form.\ e : et\}$$

and put

$$\mathrm{DVal} = \mathrm{Con} + \mathrm{Abstracts}$$

and

$$\mathrm{Env} = \mathrm{Var} \longrightarrow_{\mathrm{fin}} \mathrm{DVal}$$

and add

$$d ::= \rho$$

To extend the static semantics we define $\mathrm{FV}(dval)$ by

$$\mathrm{FV}(con) = \emptyset \qquad \mathrm{FV}(\lambda form.\ e : et) = \mathrm{FV}(e)\backslash \mathrm{DV}(form)$$

and then for $\rho : V$

$$\mathrm{DV}(\rho) = V \qquad \text{and} \qquad \mathrm{FV}(\rho) = \bigcup_{x \in V} \mathrm{FV}(\rho(x))$$

Now we define predicates $\vdash_V dval : dt$ and $\alpha \vdash_V dval$ by

**Constants:** 1. $\vdash_V m : \mathrm{int}$

2. $\alpha \vdash_V m$
3. $\vdash_V t : \text{bool}$
4. $\alpha \vdash_V t$

**Abstracts:** 1. $\vdash_V \lambda form.e : et : T(form) \to et$

2. $$\frac{form : \beta\ \alpha[\beta] \vdash_{V \cup V_0} e : et}{\alpha \vdash_V \lambda form.\ e : et} \qquad \text{(where } \beta : V_0\text{)}$$

Then the rules for environments $\rho : V$

1. $$\frac{\forall x \in V.\ \vdash_W \rho(x) : \beta(x)}{\vdash_W \rho : \beta}$$

2. $$\frac{\forall x \in V.\ \vdash_W \rho(x)}{\alpha \vdash_W \rho}$$

Turing to the transition relations we define for $\alpha : V$ and $\beta : W$, with $W \subseteq V$ and $\rho : \alpha\lceil W$, and $e$, $e'$, in $\Gamma_\alpha$ (as before)

$$\rho \vdash_\alpha e \longrightarrow e'$$

and keep the same set $\Gamma_\alpha$ of terminal expressions. Similarly we define $\rho \vdash_\alpha ae \longrightarrow ae'$ and $\rho \vdash_\alpha d \longrightarrow d'$.

The rules are formally the same as before except that for $\rho : W$ conditions of the form $\rho(f) = \ldots$ are understood to mean that $f \in W$ and $\rho(f) = \ldots$ and similarly for $\rho(x) = \ldots$ (this affects looking up the values of variables and function calls).

We need rules for recursion:

1. $$\frac{\rho|_{V_0} \vdash_{\alpha[\alpha_0]} d \longrightarrow d'}{\rho \vdash_\alpha \mathbf{rec} d \longrightarrow \mathbf{rec} d'}$$
(where $V_0 = \text{DV}(D) \cap \text{FV}(d)$ and taking $\beta$ from the requirement that $\vdash d : \beta$ we have $\alpha_0 = \beta \backslash V_0$)

2. $\rho \vdash \mathbf{rec}\ \rho_0 \longrightarrow \{x = con \mid x = con in \rho_0\}$
$\cup \{f(form) : \tau = \mathbf{let\ rec}\ \rho_0 \backslash \text{DV}(form) \mathbf{in}\ e \mid f(form) : \tau = e\ in\ \rho_0\}$

In other words we first elaborate $d$ without knowing anything about the variables and then from the resulting $\rho_0$ we yield $\rho_0$ altered to bind its free variables by $\mathbf{rec}\ \rho_0$. Here are a couple of examples. More can be found in the exercises.

**Example 1.**: Consider the traditional definition of factorial

$$d = \text{fact}(x : \text{int}) : \text{int} = \mathbf{if}\ x = 0\ \mathbf{then}\ a\ \mathbf{else}\ x \times \text{fact}(x - 1)$$

Then for any suitable $\rho$ and $\alpha$ we have

$$\rho \backslash \{\text{fact}\} \vdash_{\alpha[\alpha_0]} \text{fact}(x : \text{int}) : \text{int} = \ldots \longrightarrow \rho_0 \qquad \text{(with } \alpha_0 \text{ as given above)}$$

where $\rho_0 = \{\mathrm{fact}(x : \mathrm{int}) : \mathrm{int} = \mathbf{let}\emptyset\mathbf{in}\ldots\}$ (and from now on we omit the tedious "**let** $\emptyset$ **in**"). Then we have

$$\rho \vdash_\alpha \mathbf{rec}d \longrightarrow \mathbf{rec}\rho_0 \longrightarrow \rho_1$$

where $\rho_1 = \{\mathrm{fact}(x) = \mathbf{let\ rec}\rho_0\mathbf{in}\ldots\}$

To compute fact(0) we look at the derivation

$$\begin{aligned}
\emptyset \vdash_\emptyset \mathbf{let\ rec}\ d\ \mathbf{in}\ \mathrm{fact}(0) &\longrightarrow^* \mathbf{let}\ \rho_1\ \mathbf{in}\ \mathrm{fact}(0)\\
&\longrightarrow \mathbf{let}\ \rho_1\ \mathbf{in}\ \mathbf{let}\ x : \mathrm{int} = 0\\
&\qquad\qquad\qquad \mathbf{in\ let\ rec}\ \rho_0\\
&\qquad\qquad\qquad\quad \mathbf{in}\ \ldots\\
&\longrightarrow^* \mathbf{let}\ \rho_1\ \mathbf{in\ let}\ \{x = 0\}\ \mathbf{in}\\
&\qquad\qquad\qquad \mathbf{let}\ \rho_1\\
&\qquad\qquad\qquad \mathbf{in\ if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ \ldots\\
&\longrightarrow^3 1
\end{aligned}$$

Equally for fact(1) we have

$$\begin{aligned}
\emptyset \vdash_\emptyset \mathbf{let\ rec}\ d\ \mathbf{in}\ \mathrm{fact}(1) &\longrightarrow^* \mathbf{let}\ \{x = 1\}\ \mathbf{in\ let}\ \rho_1\mathbf{in}\\
&\qquad\quad \mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \times \mathrm{fact}(x-1)\\
&\longrightarrow^* \mathbf{let}\ \{x = 1\}\ \mathbf{in\ let}\ \rho_1\ \mathbf{in}\ x \times \mathrm{fact}(x-1)\\
&\longrightarrow^* \mathbf{let}\ \{x = 1\}\ \mathbf{in\ let}\ \rho_1\ \mathbf{in}\ 1 \times [\mathbf{let}\ x : \mathrm{int} = x - 1\ \mathbf{in\ rec}\ \rho_0\ \mathbf{in}\ldots]\\
&\longrightarrow^* \mathbf{let}\ \{x = 1\}\ \mathbf{in\ let}\ \rho_1\ \mathbf{in}\ 1 \times [\mathbf{let}\ x = 0\ \mathbf{in\ rec}\ \rho_0\ \mathbf{in}\ldots]\\
&\longrightarrow 1
\end{aligned}$$

**Example 2.**: It is allowed to define integers or truth-values recursively. For example consider $d = (\mathbf{red}x = x + 1)$. To elaborate $d$ given $\rho = \{x = 1\}$ we must elaborate $x = x + 1$ from $\rho\backslash\{x\} = \emptyset$ and that elaboration sticks as we must evaluate $x + 1$ in the empty environment. It could be helpful to specify a dynamic error in this case. Again the elaboration of

$$d = \mathbf{rec}\ x = \mathrm{fact}(0)\ \mathbf{and}\ \mathrm{fact}(x : \mathrm{int}) : \mathrm{int} = \ldots$$

does not succeeds, intuitively, we need to know the value of fact before the elaboration – which produces this value – has finished. On the other hand simple things like the elaboration of $\mathbf{rec}x = 5$ do succeed. If desired we could have specified in the static semantics that only recursive function definitions were allowed.

## 4.2 Procedures and Functions

We now consider abstractions in imperative languages. Abstract of expressions give rise to functions, as before, but now with the possibility of side-effects as in:

$$\mathbf{function}\ f(\mathbf{var}\ x : \mathrm{int}) : \mathrm{int} =$$

> **begin**
> $\quad y := y + 1$
> **result** $x + y$

In several programming languages the bodies of functions are commands, but are treated, via special syntactic devices, as expressions – see exercise 12. We take a straightforward view where the bodies are (clearly) expressions. Abstracts of commands give rise to procedures as in:

> **function** $f($**var** $x : \mathrm{int})$
> $\quad$**begin**
> $\quad\quad y := x + y$
> $\quad$**end**

which may also have side-effects and indeed are often executed for their side-effects. To see why we write **var** in the formal parameter let us see how the Principle of Correspondence allows us to treat a procedure call. First the above declaration, $d$, will be elaborated thus

$$\rho \vdash_\alpha \langle d, \sigma\rangle \longrightarrow \langle\{p(\mathbf{var}\ x : \mathrm{int}) = \{y = l\};\ y := x + y\}, \sigma\rangle$$

where $l = \rho(y)$. Then the procedure call $p(e)$ in the resulting environment $\rho'$ will look like this

$$\rho' \vdash_\alpha \langle p(e), \sigma\rangle \longrightarrow \langle \mathbf{var}\ x : \mathrm{int} = e;\ \mathbf{begin}\{y = 1\};\ y := x + y\ \mathbf{end}, \sigma\rangle$$

And we see the reason for writing **var**... is to get an easy correspondence with our previous declaration mechanism. The computation now proceeds by evaluation $e$, finding a new location $l'$, making $l'$ refer to the value of $e$ in the state and then executing the body of the procedure with $x$ bound to $l'$. This is very clearly nothing else but the classical call-by-value. Constant declarations will give rise to a call-by-constant parameter mechanism.

We begin by working these ideas out in the evident extension of the imperative language of Chapter 3. Then we proceed to other parameter mechanisms by considering the corresponding declaration mechanisms. (Many real languages will not possess such a convenient correspondence; one way to deal with their parameter mechanisms would be to add the corresponding declaration mechanisms when defining the set of possible configurations.)

For the extension we drop the **const** $x : \tau = e$ and **var**$x : \tau = e$ productions and add:

**Expressions:** $e ::= \mathbf{let}\ d\ \mathbf{in}\ e \mid f(ae) \mid \mathbf{begin}\ c\ \mathbf{result}\ e$

**Actual Expr.:** $ae ::= \cdot \mid e, ae$

**Declarations:** $d ::= form = ae \mid \mathbf{function}\ f(form) : \tau = e \mid \mathbf{procedure}\ p(form)\ c \mid \mathbf{rec}\ d$

**Formals:** $form ::= \cdot \mid \mathbf{const}\ x : \tau, form \mid \mathbf{var}\ x : \tau, form$

**Commands:** $c ::= p(ae)$

## Static Semantics

We have the following sets of identifiers with the evident definitions and meanings: $\mathrm{FI}(e)$, $\mathrm{FI}(ae)$, $\mathrm{FI}(d)$, $\mathrm{DI}(d)$, $\mathrm{DI}(form)$, $\mathrm{FI}(c)$. For example

$$\begin{aligned}&\mathrm{FI}(\mathbf{procedure}\, p(form)c) = \mathrm{FI}(c)\backslash\mathrm{DI}(form)\\ &\mathrm{DI}(\mathbf{procedure}\, p(form)c) = \{p\}\\ &\mathrm{FI}(p(ae)) = \{p\} \cup \mathrm{FI}(ae)\end{aligned}$$

Turning to types we define ETypes, AcETypes and DTypes; these are as before except that both locations and procedures are denotable, causing a change in DTypes.

$$\begin{aligned}&et ::= \tau\\ &aet ::= \cdot \mid \tau, aet\\ &dt ::= et \mid et\,\mathbf{loc} \mid aet \longrightarrow et \mid aet\,\mathbf{proc}\end{aligned}$$

and of course $\mathrm{TEnv} = \mathrm{Id} \longrightarrow_{\mathrm{fin}} \mathrm{DTypes}$. We also need $\mathrm{T}(form) \in \mathrm{AcETypes}$ with the evident definition

| | $\cdot$ | **const** $x : \tau, form$ | **var** $x : \tau, form$ |
|---|---|---|---|
| T | $\cdot$ | $\tau, aet$ | $\tau, aet$ |

Then we define the expected predicates

$$\alpha \vdash_I e : et \qquad \alpha \vdash_I ae : aet \qquad \vdash_I d : \beta \qquad \alpha \vdash_I d\, form : \beta \qquad \alpha \vdash_I c$$

We give some representative rules:

**Procedure Declarations:**

1. $\vdash_I \mathbf{procedure}\, p(form)c; \{p = T(form)\mathbf{proc}\}$

2. $$\frac{form : \beta \quad \alpha[\beta] \vdash_{I \cup I_0} c}{\alpha \vdash_I \mathbf{procedure}\, p(form)c} \qquad \text{(where } \beta : I_0)$$

**Formals:**

1. $\cdot \mid \emptyset$

2. $$\frac{form : \beta}{const x : \tau, form : \{x = \tau\} \uplus \beta} \qquad (\text{if } x \notin I_0 \text{ where } \beta : I_0)$$

3. $$\frac{form : \beta}{\mathbf{var}x : \tau, form : \{x = \tau\mathbf{loc}\} \uplus \beta} \qquad (\text{if } x \notin I_0 \text{ where } \beta : I_0)$$

**Procedure Calls:**

1. $$\frac{\alpha \vdash_I ae : aet}{\alpha \vdash_I p(ae)}$$
(if $\alpha(p) = aet\,\mathbf{proc}$)

## Dynamic Semantics

We begin with environments, abstracts and denotable values. First the set, Abstracts (ranged over by abs), is

$$\text{Abstracts} = \{\lambda form.\ e : et\} \cup \{\lambda form.\ c\}$$

then

$$\text{DVal} = \text{Con} + \text{Loc} + \text{Abstracts}$$

Where Loc is the set $\text{Loc}_{\text{int}} \cup \text{Loc}_{\text{bool}}$ of Chapter 3 and

$$\text{Env} = \text{Id} \longrightarrow_{fin} \text{DVal}$$

and we add the production

$$d ::= \rho$$

and all the above is to be interpreted recursively as usual.

Then $\text{DI}(\rho)$ and $\text{FI}(\rho)$ are defined. Next we define the evident predicates

$$\vdash_I dval : dt \qquad \alpha \vdash_I dval \qquad \vdash_I \rho : \beta \qquad \alpha \vdash_I \rho : \beta$$

as expected; for example

**Procedure Abstracts:**

1. $\vdash_I \lambda form.c : T(form)\mathbf{proc}$

2. $$\frac{form : \beta \qquad \alpha[\beta] \vdash_{I \cup I_0} c}{\alpha \vdash_I \lambda form.\ c} \qquad \text{(where } \beta : I_0\text{)}$$

**Transition Relations**: Turning to the transition relations we first need the set of stores

$$\text{Stores} = \{\sigma : L \in \text{Loc} \longrightarrow_{\text{fin}} \text{Con} \mid \sigma \text{ respects types}\}$$

– the same as in Chapter 3.

- **Expressions:** We have

  $$\Gamma_\alpha = \{\langle e, \sigma\rangle \mid \exists et.\ \alpha \vdash_I e : et\} \qquad (\text{for } \alpha : I)$$

  and

  $$\text{T}_\alpha = \{\langle con, \sigma\rangle\}$$

  and the evident relation

  $$\rho \vdash_\alpha \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle$$

- **Actual Expressions:** we have

  $$\Gamma_\alpha = \{\langle ae, \sigma\rangle \mid \exists aet.\ \alpha \vdash_I ae : aet\} \qquad (\text{for } \alpha : I)$$

  and

  $$\mathrm{T}_\alpha = \{\langle acon, \sigma\rangle\}$$

  where *acon* is in AeCon, as before. And we have the relation

  $$\rho \vdash_\alpha \langle ae, \sigma\rangle \longrightarrow \langle ae', \sigma'\rangle$$

- **Declarations:** We have

  $$\Gamma_\alpha = \{\langle d, \sigma\rangle \mid \alpha \vdash_I d\} \qquad (\text{for } \alpha : I)$$

  and

  $$\mathrm{T}_\alpha = \{\langle \rho, \sigma\rangle \mid \langle \rho, \sigma\rangle \in \Gamma_\alpha\}$$

  and the relation

  $$\rho \vdash_\alpha \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle$$

- **Formals:** We define

  $$acon, L \vdash from : \rho, \sigma$$

  meaning that in the context of an actual expression constant *acon* and given an existing set, $L$, of locations the formal (part of a declaration) form yields a new (little) environment $\rho$ and store $\sigma$.

- **Commands:** We have

  $$\Gamma_\alpha, \mathrm{T}_\alpha$$

  and

  $$\rho \vdash_\alpha \langle c, \sigma\rangle \longrightarrow \langle c', \sigma'\rangle \qquad (\text{or } \sigma')$$

  as usual.

**Rules**: The rules are generally just those we already know and only the new points are covered.

**Declarations Simple:** 1. $$\frac{\rho \vdash_\alpha \langle ae, \sigma\rangle \longrightarrow \langle ae', \sigma'\rangle}{\rho \vdash_\alpha \langle form = ae, \sigma\rangle \longrightarrow \langle form = ae', \sigma'\rangle}$$

2. $$\frac{acon, L \vdash form : \rho_0, \sigma_0}{\rho \vdash_\alpha \langle form = ae, \sigma\rangle \longrightarrow \langle \rho_0, \sigma \cup \sigma_0\rangle} \quad \text{(where } \sigma : L)$$

**Procedures:** $\rho \vdash_\alpha \langle \textbf{procedure}\ p(form)c, \sigma\rangle \longrightarrow \langle \{p = \lambda form \rho \backslash I; c\}, \sigma\rangle$
(where $I = \mathrm{FI}(c) \backslash \mathrm{DI}(form)$)

**Recursive:** 1. $$\frac{\rho \backslash R \vdash_{\alpha[\alpha_0]} d \longrightarrow d'}{\rho \vdash_\alpha \textbf{rec}\ d \longrightarrow \textbf{rec}\ c}$$
(where if $\vdash_{\mathrm{FI}(d)} d : \beta$ then $R = \mathrm{FI}(d) \cap \mathrm{DI}(d)$ and $\alpha = \beta|_R$)

2. $\rho \vdash_\alpha \textbf{rec}\ \rho_0 \longrightarrow \rho_1$
where
$$\begin{aligned}\rho_1 = {} & \{x = \mathrm{con} \mid x = \mathrm{con} \text{ in } \rho_0\} \cup \\ & \{x = l\ \textbf{loc} \mid x = l\ \textbf{loc} \text{ in } \rho_0\} \cup \\ & \{f(form) : et = \textbf{let rec}\ \rho_0 \backslash I\ \textbf{in}\ e \mid \\ & \quad f(form) : et = e \text{ in } \rho_0 \text{ and } I = \mathrm{DI}(form)\} \cup \\ & \{p(form).\ \textbf{rec}\ \rho_0 \backslash I;\ c \mid \\ & \quad p(form)c \text{ in } \rho_0 \text{ and } I = \mathrm{DI}(form)\}\end{aligned}$$

**Formals Empty:** $\cdot, L \vdash \cdot : \emptyset, \emptyset$

**Declarations Empty:** $$\frac{acon, L \vdash form : \rho_0, \sigma_0}{(\mathrm{con}, acon), L \vdash \textbf{const}\ x : \tau, form : \rho_0 \cup \{x = \mathrm{con}\}, \sigma_0}$$

**Variable:** $$\frac{acon, L \cup \{I\} \vdash form : \rho_0, \sigma_0}{(\mathrm{con}, acon), L \vdash \textbf{var}\ x : \tau, form : \{x = I\} \cup \rho_0, \{l = \mathrm{con}\} \cup \sigma_0}$$
(where $l = \mathrm{New}_\tau(L \cap \mathrm{Loc}_\tau)$)

**Example**: The following program demonstrates the use of private variables shared between several procedures. This provides a nice version of Algol's own variables and anticipates the facilities provided by classes and abstract data types. Consider the command

$$\begin{aligned} c = {} & \textbf{private var}\ x : \mathrm{int} = 1 \\ & \textbf{within} \textbf{procedure}\ \mathrm{inc}()\ x := x + 1 \\ & \qquad \textbf{procedure}\ \mathrm{dec}()\ \textbf{if}\ x > 0\ \textbf{then}\ x : x - 1\ \textbf{else}\ nil; \\ & \textbf{begin} \\ & \quad \mathrm{inc}();\ \mathrm{dec}() \\ & \textbf{end}\end{aligned}$$

First look at the declaration part, $d$:

$$\begin{aligned} & \rho \vdash \langle d, \sigma\rangle \longrightarrow \langle \textbf{private}\ \{x = 1\}\ \textbf{within procedure}\ \mathrm{inc}() \ldots;\ \textbf{procedure}\ \mathrm{dec}() \ldots, \sigma[I = 1]\rangle \\ & \longrightarrow \langle \textbf{private}\ \{x = 1\}\ \textbf{within}\{\mathrm{inc}() = \{x = 1\}; -\}; \textbf{procedure}\ \mathrm{dec}() \ldots, \sigma[I = 1]\rangle \\ & \longrightarrow^3 \langle \textbf{private}\ \{x = 1\}\ \textbf{within}\{\mathrm{inc}() = \{x = 1\}; -, \mathrm{dec}()\{x = 1\}; \ldots, \sigma[I = 1]\rangle \\ & \longrightarrow \langle \{\mathrm{inc}()\{x = 1\}; -, \mathrm{dec}()\{x = 1\}; \ldots, \sigma[I = 1]\rangle \\ & \quad = \langle \rho, \sigma[I = 1]\rangle, \text{say}\end{aligned}$$

So we see that

$$\rho \vdash \langle c, \sigma \rangle \longrightarrow^* \langle \rho_0; (\mathrm{inc}(); \mathrm{dec}()), \sigma[I = 1] \rangle$$

and so we should examine the computation:

$$\begin{aligned}
&\rho[\rho_0] \vdash \langle \mathrm{inc}(); \mathrm{dec}(), \sigma[I = 1] \rangle \\
&\qquad \longrightarrow \langle (\{x = 1\}; x := x + 1); \mathrm{dec}(), \sigma[I = 1] \rangle \\
&\qquad \longrightarrow^* \langle \mathrm{dec}(), \sigma[I = 2] \rangle \\
&\qquad \longrightarrow \langle \{x = 1\}; \mathbf{if}\ x > 0\ \mathbf{then}\ x := x - 1\ \mathbf{else\ nil}, \sigma[I = 2] \rangle \\
&\qquad \longrightarrow^* \langle \sigma[I = 1] \rangle.
\end{aligned}$$

## 4.3 Other Parameter Mechanisms

Other parameter mechanisms can be considered in the same manner. The general principle is to admit more ways to declare identifiers (as discussed above) and to admit more ways of evaluating expressions (and/or actual expressions). The latter is needed because actual expressions can be evaluated to various degrees when abstracts are called. One extreme is absolutely no evaluation (see exercise for this call-by-text mechanism). We shall first consider call-by-name in the context of our applicative language which we regard as evaluating the argument to the extent of binding the call-time environment to it; this well-known idea differs from the official ALGOL-60 definition and is discussed further in exercise 15.

Then we consider call-by-reference in the context of our imperative language where the argument is evaluated to produce a reference. Other mechanisms are considered in the exercises. Note that in call-by-name for example the actual parameter may be further evaluated during computation of the body of the abstract. It is even possible to have mechanisms (e.g. variants of call-by-result) where some or all of the evaluation is delayed until *after* the computation of the body of the abstract.

### Call-by-Name

Syntactically it is only necessary to add another possibility for the formal parameters to the syntax of our applicative language

$$form ::= \mathbf{name}\ x : \tau, form$$

### Static Semantics

The sets of defining variables of **name** $x : \tau$, form is clearly $\{x\} \cup \mathrm{DV}(form)$. Regarding types we add

$$\begin{aligned}
&aet ::= \tau\ \mathbf{name},\ aet \\
&dt ::= et \mid \tau\ \mathbf{name} \mid aet \to et
\end{aligned}$$

The definition of the type $\mathrm{T}(form)$ of a formal needs the new clause

$$\mathrm{T}(\mathbf{name}\ x : \tau, form) = \tau\ \mathbf{name},\ \mathrm{T}(form)$$

Here are the new predicate rules

**Formals:** $form : \beta \Longrightarrow (\mathbf{name} x : \tau, form) : \{x = \tau \mathbf{name}\} \cup \beta (\text{if } x \notin \mathrm{DV}(form))$

**Expressions Variables:** $\alpha \vdash_V x : \tau \qquad (\text{if } \alpha(x) = \tau\ \mathbf{name})$

This rule expresses the fact that if $x$ is a call-by-name formal parameter as in **name** $x : \tau$ then in the calling environment its denotation can be evaluated to a value of type $\tau$.

**Actual Expr.:** $\dfrac{\alpha \vdash_V e : et \qquad \alpha \vdash_V ae : aet}{\alpha \vdash_V e, ae : et\mathbf{name}, aet}$

It is important to note that this rule is in *addition* to the previous rule. So given $\alpha$ an actual expression can have several different types; these are needed as the same expressions can correspond to formals of different types, and that will require different kinds of evaluation.

**Example**: Consider these two expressions

$$\begin{array}{l}\mathbf{let\ function}\ fred\ (x : \mathrm{int}, \mathbf{name}\ y : \mathrm{int}) : \mathrm{int} = x + y \\ \mathbf{in}\ fred(u + v, u - v)\end{array}$$

and

$$\begin{array}{l}\mathbf{let\ function}\ fred\ (\mathbf{name}\ x : \mathrm{int}, y : \mathrm{int}) : \mathrm{int} = x + y \\ \mathbf{in}\ fred(u + v, u - v)\end{array}$$

in the first case we need the fact that $\alpha \vdash u + v, u - v : \mathrm{int}, \mathrm{int}\mathbf{name}$ and in the second that $\alpha \vdash u + v, u - v : \mathrm{int}\ \mathbf{name}, \mathrm{int}$ (where $\alpha = \{u = \mathrm{int}, v = \mathrm{int}\}$).

### Dynamic Semantics

Clearly we must add a new component to the set of denotable values, corresponding to the new denotable types $\tau$ **name**

$$\mathrm{DVal} = \mathrm{Con} + \mathrm{NExp} + \mathrm{Abstracts}$$

where we need $\mathrm{NExp} = \{e : \mathbf{name}\ \tau\}$ to allow free variables in the expressions because of the possibility of recursive definitions. For example consider

$$\begin{array}{l}\mathbf{rec\ name}\ x : \mathrm{int} = f(5)\ \mathbf{and} \\ \quad f(x : \mathrm{int}) = \ldots f \ldots\end{array}$$

The extension to the definition of $\mathrm{FV}(dval)$ is, of course, clear

$$\mathrm{FV}(e : \mathbf{name}\ \tau) = \mathrm{FV}(e)$$

For the predicates $\vdash_V dval : dt$ and $\alpha \vdash_V dval$ we add the rules

$$\vdash_V (e : \mathbf{name} : \tau) : \tau\ \mathbf{name} \qquad \frac{\alpha \vdash e : \tau}{\alpha \vdash e : \mathbf{name}\ \tau}$$

**Transition Relations**: For expressions and definition we refine the usual $\rho \vdash_\alpha e \longrightarrow e'$ and $\rho \vdash_\alpha d \longrightarrow d'$ a little, parameterizing also on the set of variables where definition is currently available in the environment (the others will be in the process of begin recursively defined). So for $\alpha : V$ and $W \subseteq V$ we will define the relations

$$\rho \vdash_{\alpha,W} e \longrightarrow e' \qquad \text{and} \qquad \rho \vdash_{\alpha,W} d \longrightarrow d'$$

where $\rho : \alpha|_W$ and $e, e' \in \Gamma^{\text{exp}}_{\alpha,W}$ and $d, d' \in \Gamma^{\text{def}}_{\alpha,W}$ where

$$\begin{aligned} \Gamma^{\text{exp}}_{\alpha,W} &= \{e \mid \exists et.\ \alpha \vdash_V e : et\} \\ \Gamma^{\text{def}}_{\alpha,W} &= \{d \mid \alpha \vdash_V d\} \end{aligned}$$

We also have the evident $\mathrm{T}^{\text{exp}}_{\alpha,W}$ and $\mathrm{T}^{\text{def}}_{\alpha,W}$.

For formals we have the predicate

$$ae \vdash_{\mu,W} form : \rho_0$$

where $ae \in \mathrm{T}_{\mu,W}$ and $\mu = \mathrm{M}(\mathrm{T}(form))$..

For actual expressions the result desired will depend on the context and we introduce an apparatus of different *evaluation modes*. The set Modes of modes is ranged over by $\mu$ given by

$$\mu ::= \cdot \mid \mathbf{value}, \mu \mid \mathbf{name}, \mu$$

Each actual expression type, $aet$, has a mode, $\mathrm{M}(aet)$ where

$$\begin{aligned} &\mathrm{M}(\cdot) = \cdot \\ &\mathrm{M}(\tau, aet) = \mathbf{value}, M(aet) \\ &\mathrm{M}(\tau, \mathbf{name},\ aet) = \mathbf{name}, M(aet) \end{aligned}$$

We define transition relations $\rho \vdash_{\alpha,W,\mu} ae \longrightarrow ae'$ which are also parameterized on modes. The set of configurations is, for $\alpha : V$, $W \subseteq V$ and mode $\mu$

$$\Gamma_{\alpha,W,\mu} = \{ae \mid \exists aet.\ \alpha \vdash_V ae : aet and \mathrm{M}(aet) = \mu\}$$

and we define the set $\mathrm{T}_{\alpha,W,\mu}$ of terminal actual expressions by some rules of the form $\vdash_{\mu,W} \mathrm{T}(ae)$

1. $\vdash_{\cdot,W} T(\cdot)$

2. $$\frac{\vdash_{\mu,W} T(ae)}{\vdash_{(\mathbf{value},\mu),W} T(con, ae)}$$

3. $$\frac{\vdash_{\mu,W} T(ae)}{\vdash_{(\mathbf{name},\mu)}), W} T(e, ae) \text{ (if } \mathrm{FV}(e) \cap W = \emptyset)$$

It is rule 3 which introduces the need for $W$., insisting that all variables are bound, except, possibly, for those begin recursively define.

The transition relation is defined for $\rho : \alpha\lceil W$ and $ae, ae' \in \Gamma_{\alpha,W,\mu}$ and has the form $\rho \vdash_{\alpha,W,\mu} ae \longrightarrow ae'$ . The apparatus of modes gives types what might also be called metatypes and this may be a useful general idea. The reader should not be confused with one normal usage of the term mode as synonymous with type.

**Transition Rules**:

- **Expressions:** These are the same as before except for identifiers:

**Identifiers:**

1. $\rho \vdash x \longrightarrow \text{con} \qquad (\text{if } \rho(x) = \text{con})$
2. $\rho \vdash x \longrightarrow x \longrightarrow e \qquad (\text{if } \rho(x) = e : \textbf{name}\ \tau)$

**Actual Expr. Value Mode:**

1. $$\frac{\rho \vdash_{\alpha,W} e \longrightarrow e'}{\mu \vdash_{\alpha,(\textbf{value},\mu),W} e, ae \longrightarrow e', ae}$$
2. $$\frac{\mu \vdash_{\alpha,\mu,W} ae \longrightarrow ae'}{\mu \vdash_{\alpha,(\textbf{value},\textbf{mu}),W} con, ae \longrightarrow con, ae'}$$

**Name Mode:**

1. $\rho \vdash_{\alpha,(\textbf{name},\mu),W} e, ae \longrightarrow (\textbf{let}\ \mu\lceil \text{FV}(e)\ \textbf{in}\ e), ae$
2. $$\frac{\rho \vdash_{\alpha,\mu,W} ae \longrightarrow æ'}{\rho \vdash_{\alpha,(\textbf{name},\mu),W} e, ae \longrightarrow e, ae'} \qquad (\text{if FV}(e) \cap W = \emptyset)$$

- **Definitions:** Here we need a rule which ensures that the actual expressions are evaluated in the right mode. Otherwise the rules are as before.

**Simple:**

1. $$\frac{\rho \vdash_{\alpha,mu,W} ae \longrightarrow ae'}{\rho \vdash_{\alpha,W} form = ae \longrightarrow form = ae'}$$
(where $\mu = \text{M}(\text{T}(form))$)
2. $$\frac{ae \vdash form : \rho_0}{\rho \vdash_{\alpha,W} form = ae \longrightarrow \rho_0}$$
(if $ae \in \text{T}_{\alpha,\mu,W}$ where $\mu = \text{M}(\text{T}(form))$

**Formals:**

1. $\cdot \vdash_{\cdot,W} \cdot : \emptyset$
2. $$\frac{ae \vdash_{\mu,W} form : \rho}{con, ae \vdash_{(\textbf{value},\mu),W} (x : \tau, form) : \{x = con\} \cup \rho}$$
3. $$\frac{ae \vdash_{\mu,W} form : \rho}{e, ae \vdash_{(\textbf{name},\mu),W} (\textbf{name}\ x : \tau, form) : \{x = e : \textbf{name}\ \tau\} \cup \rho}$$

**Example**: The main difference between call-by-name and call-by-value in applicative languages is that call-by-name may terminate where call-by-value need not. For example consider the expression

$$\begin{aligned} e = \textbf{let}\ & f(\textbf{name}\ x : \text{int}) : \text{int} = 1\ \textbf{and} \\ & \quad \textbf{rec}\ g(x : \text{int}) : \text{int} = g(x) \\ & \textbf{in}\ \ f(g(2)) \end{aligned}$$

Then

$$\rho \vdash e \longrightarrow^* \textbf{let}\ \rho_0\ \textbf{in}\ f(g(2))$$

where $\rho_0 = \{f(\mathbf{name}\, x : \text{int}))\text{int} = 1, g(x : \text{int}) : \text{int} = \ldots\}$

So we look at

$$\begin{aligned}\rho_0 \vdash f(g(2)) &\longrightarrow \mathbf{let\ name}\, x : \text{int} = g(2)\mathbf{in}\, 1\\ &\longrightarrow^* \begin{array}{l}\mathbf{let}\, \{x = \mathbf{let}\, g(x : \text{int}) : \text{int} = \ldots\}\\ \mathbf{in}\, 1\end{array}\\ &\longrightarrow 1\end{aligned}$$

On the other hand if we change the formal parameter of $f$ to be call-by-value instead, then, as the reader may care to check, the evaluation does not terminate.

### Call-by-Reference

We consider a variant (the simplest one!) where the actual parameter must be a variable (identifier denoting a location). In other languages the actual parameter could be any of a wide variety of expressions which are evaluated to produce a location; these might include conditionals and function calls. This would require a number of design decisions on the permitted expressions and on how the type-checking should work. For lack of time rather than any intrinsic difficulty we leave such variants to exercise 17. Just note that it will certainly be necessary to rethink expression evaluation; this should either be changed so that evaluation yields a natural value (be it location or primitive value) or else different evaluation modes should be introduced.

Syntactically we consider an extension to our imperative language

$$form ::= \mathbf{loc}\, x : \tau, form.$$

### Static Semantics

Clearly we have DI($\mathbf{loc} x : \tau, form) = \{x\} \cup DI(form)$. For types we add another actual expression type

$$aet ::= \tau\mathbf{loc}, aet$$

and

$$T(\mathbf{loc}\, x : \tau, form) = \tau\mathbf{loc}, aet$$

and we have the rule

$$\frac{form : \beta}{\mathbf{loc} x : \tau, form : \{x = \tau\mathbf{loc}, \beta\}} \qquad (\text{if } x \notin I \text{ where } \beta : I)$$

- **Actual Expressions**: These are as before with the addition

$$\frac{\alpha \vdash_V ae : aet}{\alpha \vdash_V x, et : \tau\ |bf loc, aet} \qquad (\text{if } alpha(x) = \tau\ \mathbf{loc})$$

It is here that we insist that actual reference parameters must be variables. As in the case of call-by-name the type of an actual expression is not determined by its environment alone, but by its context as well. (A more honest notation might be $\alpha, aet \vdash_{V\, ae}$ rather than $\alpha \vdash_V\ ae : aet$.)

**Dynamic Semantics**

It is not necessary to change the definitions of DVal (or Env or Dec) as locations are already include. However, we allow locations in AcExp and AeCon

$$ae ::= I, ae$$
$$acon ::= I, acon$$

and clearly $\mathrm{FV}(I, ae) = \mathrm{FV}(ae)$ and we have the rule

$$\frac{\alpha \vdash_I ae : aet}{\alpha \vdash_I I, ae : \tau\ \mathbf{loc}, aet} \qquad (I \in Loc_\tau)$$

**Transition Rules**: We have relations $\rho \vdash_\alpha \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle$, $\rho \vdash_\alpha \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle$ and $\rho \vdash_\alpha \langle c, \sigma\rangle \longrightarrow \langle c', \sigma'\rangle$ (or $\sigma'$) and a predicate $acon, L \vdash form : \rho, \sigma$ as before. For actual expressions we proceed as with call-by-name and introduce a set, Mode, of evaluation modes

$$\mu ::= \cdot \mid \mathbf{val}, \mu \mid \mathbf{loc}, \mu$$

with the evident definition of $M(aet) \in Mode$ and put for $\alpha : I$ and $\mu$,

$$\Gamma_{\alpha,\mu} = \{\langle ae, \sigma\rangle \mid \exists aet.\ \alpha \vdash_I ae : aet and \mu = \mathrm{M}(aet)\}$$
$$\mathrm{T}_{\alpha,\mu} = \{\langle acon, \sigma\rangle \in \Gamma_{\alpha,\mu}\}$$

and will define the transition relation for $\rho : \alpha$ and $\mu$

$$\rho \vdash_{\alpha,\mu} \langle ae, \sigma\rangle \longrightarrow \langle ae', \sigma'\rangle$$

**Rules**:

- **Expressions: Actual Expressions**

  **Vaule Mode:**

  1. $$\frac{\rho \vdash_\alpha \langle e, \sigma\rangle \longrightarrow \langle e'\sigma'\rangle}{\rho \vdash_{\alpha,(\mathbf{val},\mu)} \langle (con, ae), \sigma\rangle \longrightarrow \langle (e', ae), \sigma'\rangle}$$
  2. $$\frac{\rho \vdash_\alpha \langle ae, \sigma\rangle \longrightarrow \langle ae', \sigma'\rangle}{\rho \vdash_{\alpha,(\mathbf{val},\mu)} \langle (con, ae), \sigma\rangle \longrightarrow \langle (con, ae'), \sigma'\rangle}$$

  **Ref. Mode:**

  1. $\rho \vdash_{\alpha,(\mathbf{loc},\mu} \langle (x, ae), \sigma\rangle \longrightarrow \langle (I, ae), \sigma\rangle$ (if $\rho(x) = I$)
  2. $$\frac{\rho \vdash_{\alpha,\mu} \langle ae, \sigma\rangle \longrightarrow \langle ae', \sigma'\rangle}{\rho \vdash_{\alpha,(\mathbf{loc},\mu)} \langle (I, ae), \sigma\rangle \longrightarrow \langle (I, ae'), \sigma'\rangle}$$

- **Definitions:**

  **Simple:**

  1. $$\frac{\rho \vdash_{\alpha,\mu} \langle ae, \sigma\rangle \longrightarrow \langle ae', \sigma'\rangle}{\rho \vdash_\alpha \langle form = ae, \sigma\rangle \longrightarrow \langle form = ae', \sigma'\rangle}$$
     (if $\mu = \mathrm{M}(\mathrm{T}(form))$)
  2. $$\frac{acon, L \vdash form : \rho_0, \sigma_0}{rho \vdash_\alpha \langle form = acon, \sigma\rangle \longrightarrow \langle \rho_0, \sigma \cup \sigma_0\rangle} \qquad (\text{where } \sigma : L)$$

**Formals:** We Just add a rule for declaration-by-reference (= location)

$$\frac{acon, L \vdash form : \rho_0, \sigma_0}{(I, acon), L \vdash (\mathbf{loc} x : \tau, form) : \{x = I\} \cup \rho_0, \sigma_0}$$

**Note**: All we have done is to include the construct $x == y$ of Chapter 3 in our simple declarations.

- **Commands:** No new rules are needed.

Clearly our discussion of binding mechanisms is only a start, even granting the ground covered in the exercises. I hope the reader will have been lead to believe that a more extensive coverage is feasible. What is missing is a good guiding framework to permit a systematic coverage.

## 4.4 Higher Types

Since we can define or declare abstractions, such s functions and procedures, Tennents Principle of Correspondence tells us that we can allow abstractions themselves as parameters of (other) abstractions. The resulting abstractions are said to be of *higher types* (the resulting functions are often called *functionals*. For example the following recursive definition is of a function to apply a given function, $f$, to a given argument, $x$, a given number, $t$, of times:

$$\begin{array}{l} \mathbf{rec}\ \mathrm{Apply}(f : \mathrm{int} \longrightarrow in, x : \mathrm{int}, t : \mathrm{int}) : \mathrm{int} = \\ \qquad \mathbf{if}\ t = 0\ \mathbf{then}\ x\ \mathbf{else}\ \mathrm{Apply}(f, f(x), t-1) \end{array}$$

We will illustrate this idea by considering a suitable extension of the imperative language of this chapter (but neglecting call-by-reference). Another principle would be to allow any denotable type to be an expressible type; this principle would allow locations or functions and procedures as expressions and, in particular, as results of functions (by the Principle of Abstraction). For example we could define an expression (naturally, called an *abstraction*)

$$\lambda form.e$$

that would be an abbreviation for the expression **let** $f(form) : \tau = e$**in** $f$. For a suitable $\tau$, depending on the context, it might, more naturally, be written as: *function* form. $e$; such functions (and other similar abstractions) are often termed anonymous. Then the following function would output the composition of two given functions

$$\begin{array}{l} \mathrm{Compose}(f : \mathrm{int} \rightarrow \mathrm{int}, g : \mathrm{int} \rightarrow \mathrm{int}) : \mathrm{int} \rightarrow \mathrm{int} \\ \qquad = \lambda x : \mathrm{int}.\ f(g(x)) \end{array}$$

In this way we obtain (many) versions of the typed $\lambda$-calculus. A number of problems arise in imperative languages where functions are not denotable, but only references to them. In the definition of Compose one will have locally declared references to functions as the denotations of $f$ and $g$; if these are disposed of upon termination of the function call one will have a dangling reference. Just the same thing happens, but in an even more bare-faced way, if we allow locations as outputs

$$\textbf{function}\ f() : \text{int}\ \textbf{loc}\ =\ \textbf{var}\ x : \text{int} = 5\ \textbf{in}\ x$$

At any rate we will leave these issues to exercises, begin moderately confident they can be handled along the lines we have developed.

Now, let us turn to our language with higher types. We extend the syntax by including the category AcETypes of actual expression types:

$$aet ::= \cdot \ \mid\ \tau, aet\ \mid\ (aet \longrightarrow \tau), aet\ \mid\ aet\ \textbf{proc}, aet$$

and then add to the stock of formals

$$form ::= \textbf{function}\ f : aet \rightarrow \tau, form\ \mid\ \textbf{procedure}\ p : aet, form$$

It is clear how this allows functions and procedures of higher type to be defined; they are passed as arguments via identifiers that denote them.

**Static Semantics**

Clearly

$$\begin{array}{ll} \mathrm{DI}(\textbf{function} f : aet \longrightarrow \tau, form) = \{f\} \cup \mathrm{DI}(form) & \text{and} \\ \mathrm{DI}(\textbf{procedure} f : aet, form) = \{p\} \cup \mathrm{DI}(form) & \end{array}$$

The definition of $\mathrm{T}(form)$ in AcETypes is also evident and we note

$$\begin{array}{l} \mathrm{T}(\textbf{function}\ f : aet \rightarrow \tau, form) = (aet \rightarrow \tau), T(form) \\ \mathrm{T}(\textbf{procedure}\ p : aet, form) = aet\ \textbf{proc}, \mathrm{T}(form) \end{array}$$

As for the predicate $form : \beta$ we first note the definition of the set, DTypes, of denotable types:

$$dt ::= et\ \mid\ et\ \textbf{loc}\ \mid\ aet \longrightarrow et\ \mid\ aet\ \textbf{proc}$$

The rules are fairly clear and we just note the procedure case:

$$\frac{form : \beta}{\textbf{procedure}\ p : aet, form : \{p = aet\ \textbf{proc} \uplus \beta\}} \qquad (\text{if } p \notin I \text{ where } \beta : I)$$

Turning to the other predicates we only need to add a rule for actuals:

$$\frac{\alpha \vdash_I ae : aet}{\alpha \vdash_I x, ae : dt, aet}$$

(where $dt = \alpha(x)$ is either of the form $aet \rightarrow et$ or $aet$ **proc**)

**Example**: Try type-checking the following imperative version of Apply in the environment $\{x = \mathrm{int}\}$

> **function** double($x$ : int) : int $= 2 \times x$
> **rec function** apply(**function**$f$ : int $\to$ int, $x$ : int, $t$ : int) : int =
>     **let var** result : int $= x$ **in**
>     **begin while** $t > 0$ **do** $x := f(x);\ t := t - 1$
>     **result** result;
> $x :=$ apply(double, $x, x$)

## Dynamic Semantics

Once more there is no need to change (the form of) the definitions of DVal or Env or Dec. We must now allow abstracts within actual expressions and also AcCon

$$ae ::= (\lambda form.\ e : \tau), ae \ \mid\ (\lambda form.\ c), ae$$
$$acon ::= (\lambda form.\ e : \tau), acon \ \mid\ (\lambda form.\ c), acon$$

with the evident extensions to the definitions of $\mathrm{FI}(ae)$ and $\alpha \vdash_I ae : aet$.

**Transition Relations**: In the following $\alpha : I$ and $j \subseteq I$.

- **Expressions:** We define configurations and terminal configurations and terminal configurations as usual; for the transition relation we define for $\rho : \alpha|_J$

  $$\rho \vdash_{\alpha,J} \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle$$

- **Actual Expressions:** We take

  $$\Gamma_{\alpha,J} = \{\langle ae, \sigma\rangle \ \mid\ \mathrm{FI}(ae) \subseteq I\}$$

  and

  $$\mathrm{T}_{\alpha,J} = \{\langle acon, \sigma\rangle \ \mid\ \mathrm{FI}(acon) \cap J = \emptyset\}$$

  and for $\rho : \alpha|_J$ the relation

  $$\rho \vdash_{\alpha,J} \langle ae, \sigma\rangle \longrightarrow \langle ae', \sigma'\rangle$$

- **Declarations:** We define $\Gamma_{\alpha,J}$, $\mathrm{T}_{\alpha,J}$ in the evident way, and the transition relation $\rho \vdash_{\alpha,J} \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle$

- **Commands:** Again the configurations, the terminal configurations and the transition relation are of the evident forms.

- **Formals:** We will define the predicate $acon,\ L \vdash_J form : \rho_0, \sigma_0$ where $\mathrm{FI}(acon) \cap J = \emptyset$.

**Rules**: **Expressions**, **Declarations**, **Commands** as before.

- **Actual Expressions:** As before, plus

$$\rho \vdash_{\alpha,J} \langle (x, ae), \sigma\rangle \longrightarrow \langle (abs, ae), \sigma\rangle \qquad (\text{if } \rho(x) = abs \in \text{Abstracts})$$

$$\frac{\rho \vdash_{\alpha,J} \langle ae, \sigma\rangle \longrightarrow \langle ae', \sigma'\rangle}{\rho \vdash_{\alpha,J} \langle (abs, ze), \sigma\rangle \longrightarrow \langle (abs, ae'), \sigma'\rangle}$$

- **Formals:** We just need two more rules

$$\frac{acon, L \vdash_J form : \rho, \sigma}{((\lambda form.\ e : \tau), acon), L \vdash_J \textbf{ function } f : aet \to \tau, form : \{f = \lambda forme : \tau\} \uplus \rho, \sigma}$$

$$\frac{acon, L \vdash_J form : \rho, \sigma}{((\lambda form.\ c), acon), L \vdash_J \textbf{procedure } p : aet, form : \{p = \lambda form.c\} \uplus \rho, \sigma}$$

As a matter of fact the $J$'s are not needed, but we obtain finer control over the allowable actual expression configurations. This can be useful in extensions of our language where abstractions are allowed.

## 4.5 Modules and Classes

There is a certain confusion of terminology in the area of modules and classes. Rather than enumerate the possibilities let me say what I mean here. First these is a *Principle of Denotation* which says that one can in principle use an identifier to denote the value of any syntactic phrase – where "value" is dellbertely ambiguous and may indicate various degrees of "evaluation". For expressions this says we can declare constants (in imperative languages) but also allows declaration by name or by text and so on; for commands it means we can have parameterless subroutines. For declarations we take it as meaning one can declare identifiers as modules, and they will denote the environment resulting from the elaboration. (There is a corresponding *Principle of Storeability* which the reader will spot for himself; it is anything but clear how useful these principles are!)

Applying the principle of abstraction to declarations on the other hand we obtain what we call classes. Applying a class to actual arguments gives a declaration which can be used to supply a denotation to a module identifier; then we say the module is an instance of the class. (Of course everything we say here applies just as well to applicative languages; by now, however, it is enough just to consider one case!)

A typical example is providing a random integer facility. Let $d_{\text{rand}}$ be the declaration

**private**
    **var** $a$ = seed **mod** $d$
**within**
    **function** draw() : int

**begin** $a := a \times m$ **mod** $d$
**result** $a/d$
**end**

This would declare a function, draw, providing a random integer with its own private variable – insuccessible from the outside. If one wanted to declare and use two random integers, just declare two modules

**module** X : draw : $\cdot \to$ int = $d_{\text{rand}}$
**module** Y : draw : $\cdot \to$ int = $d_{\text{rand}}$
**begin**
  $\ldots X.draw() \ldots Y.draw()$

Thus draw is an *attribute* of both X and Y and the syntax X. draw selects the attribute (in general there is more than one).

When one wnts some parameterization and/or desires to avoid writing out $d_{\text{rand}}$ several times, one can declare and use a *class*

**class** random(**const** seed : int, **const** $d$ : int) : draw : $\cdot \to$ int; $d_{\text{rand}}$
**begin**
  $\vdots$
  **module** X : draw : $\cdot \to$ int = random(5, 2);
  **module** Y : draw : $\cdot \to$ int = random(2, 3);
  **begin** ... X.draw() ... Y.draw() ... **end**
  $\vdots$
**end**

Finally we note that it is possible to use the compound forms of declarations to produce similar effects on classes. For example a version of the SIMULA classprefixing idea is available.

**class** CLASS1($form1$) ... ; ... ;
**class** CLASS2($form2$)—; —;
**class** PREFIXCLASS($form1, form2$) ... —;
    CLASS1($form1$); CLASS2($form2$)

Naturally we will also be able to use simultaneous and private and recursive class declarations (can you tell me some good examples of the use of these?). One can also easily envisage classes of higher types (classicals?), but we do not investigate this idea.

Here is our extension of the syntax of the imperative language of the present chapter (but no call-by-reference, or higher types).

- **Types:** We need the categories DTSpecs, AcETypes and DecTSpecs of denotable type specifications, actual expression types and declaration type specifications

$$dts ::= \tau \mid \tau\ \mathbf{loc} \mid aet \to \tau \mid aet\ \mathbf{proc} \mid dects \mid aet \to dects$$

$aet ::= \cdot \mid \tau, aet$
$dects ::= x : dts \mid x : dts, dexcts$

Clearly $dect$ will be the type of a module identifier and $aet \rightarrow dect$ will be the type of a class identifier.

- **Expressions:** We add five(!!) new categories of expressions, funcion, procedure, variable, module and class expressions, called FExp, PExp, VExp, MExp, CExp and ranged over by $fe$, $pe$, $ve$, $me$, $cle$, and given by the following productions (where we also allow $f$, $p$, $v$, $m$, $cl$, as metavariables over the set, Id, of identifiers)

  $fe ::= f \mid me.f$
  $pe ::= p \mid me.p$
  $ve ::= v \mid me.v$
  $me ::= m \mid me.m \mid cle(ae)$
  $cle ::= cl \mid me.cl$

  The definition of the set of expressions is extended by

  $e ::= me.x \mid fe(ae)$

  (and the second possibility generalizes expressions of the form $f(ae)$). The set of actual expressions is defined as before.

- **Commands:** We generalize commands of the forms $p(ae)$ and $x := e$ (i.e. procedure calls and assignment statements) by

  $c ::= pe(ae) \mid ve := e$

- **Declarations:** We add the following productions to the definition

  $d ::= \textbf{module}\ m : dects = d \mid \textbf{class}\ cl(form) : dects; d$

  Note that declaration types are used here to specify the types of the attributes of modules and classes. If we except recursive declarations this information is redundant, but it could be argued that it increases readability as the attribute types may be buried deep inside the declarations.

- **Formals:** The definition of these remains the same as we do not want class or module parameters.

**Note**: In this chapter we have essentially been following a philosophy of different expressions for different uses. This is somewhat inconsistent with previous chapters where we have merged different kinds of expressions (e.g. integer and boolean) and been context to separate them out again via the static semantics. By now the policy of this chapter looks a little ridiculous and it could well be better to merge everything together. However, the reader may have appreciated the variation.

**Static Semantics**

For the definitions of $\mathrm{FI}(fe), \ldots, \mathrm{FI}(cle)$ we do not regard the attribute identifiers as free (but rather as a different use of identifiers from all previous identifiers; their occurrences are the same as constant occurrences and they are thought of as standing for themselves). So for example $\mathrm{FI}(me)$ is given by the table

| | $m$ | $me.m$ | $cle(ae)$ |
|---|---|---|---|
| FI | $\{m\}$ | $\mathrm{FI}(me)$ | $\mathrm{FI}(cle) \cup \mathrm{FI}(ae)$ |

For the definions $f$ of $\mathrm{FI}(e)$, $\mathrm{FI}(c)$ we put

$\mathrm{FI}(me.x) = \mathrm{FI}(me)$
$\mathrm{FI}(fe(ae)) = \mathrm{FI}(fe) \cup \mathrm{FI}(ae)$
$\mathrm{FI}(pe(ae)) = \mathrm{FI}(pe) \cup \mathrm{FI}(ae)$
$\mathrm{FI}(ve := e) = \mathrm{FI}(ve) \cup \mathrm{FI}(e)$

and for $\mathrm{FI}(d)$ and $\mathrm{DI}(d)$ we have

| | **module** $m : dect = d$ | **class** $cl(for) : dect; d$ |
|---|---|---|
| FI | $\mathrm{FI}(d)$ | $\mathrm{FI}(d) \backslash \mathrm{DI}(for)$ |
| DI | $\{m\}$ | $\{d\}$ |

(We are really cheating somewhere here. For example the above scheme would not work if we added the reasonable production

$d ::= me$

as then with, for example, a command $m$; **begin** $\ldots x \ldots$ **end** the $x$ can be in the scope of the $m$ if the command is in the scope of a declaration of the form **module** $m : dect =$ **var** $x\mathrm{int} = \ldots; \ldots$

Thus it is nolonger possible todefine the free identifiers of a phrase in a context-free way. Let us agree to ignore the problem.)

- **Types:** We define (mutually recursively) the sets ETypes, FETypes, ..., CIETypes, DTYpes, TEnv of expression types, function expression types, ..., class expression types, denotable types and type environments by

  $et ::= \tau$

$fet ::= aet \to \tau$
$pet ::= aet$ **proc**
$vet ::= \tau$ **loc**
$met ::= \alpha$
$clet ::= aet \longrightarrow \alpha$
$dt ::= et \mid vet \mid fet \mid pet \mid met \mid clet$
$\mathrm{TEnv} = \mathrm{Id} \longrightarrow_{\mathrm{fin}} \mathrm{DTypes}$ (with $\alpha$ ranging over TEnv)

To see how the sets DTSpecs and DecTSpecs of denotable and declaration type specifications specify denotable and declaration types respectively, we define predicates

$$dts : dt \qquad \text{and} \qquad dects : \alpha$$

by the formulae

- **DTSpecs:**

1. $\tau : \tau$
2. $\tau\ \mathbf{loc} : \tau\ \mathbf{loc}$
3. $aet \to \tau : aet \to \tau$
4. $aet\ \mathbf{proc} : \mathbf{aet\ proc}$
5. $\dfrac{dects : \alpha}{dects : \alpha}$ (where the premise means proved from the rules for DecTSpecs)
6. $\dfrac{dects : \alpha}{aet \to dects : aet \to \alpha}$

- **DecTSpecs:**

1. $\dfrac{dts : \alpha}{(x : dts) : \{x = \alpha\}}$
2. $\dfrac{dts : \alpha \qquad dects : \beta}{(x : dts, dets) : \{x = \alpha\} \cup \beta}$ (if $x \notin I$ for $\beta : I$)

Next $\mathrm{T}(form) \in$ AcETypes is defined as before. Now we must define the predicates

$$\begin{array}{ll} \alpha \vdash_I e : et & \alpha \vdash_I fe : fet, \ldots, \alpha \vdash_I cle : clet, \alpha \vdash_I c, \\ \vdash_I d : \beta & \alpha \vdash_I d\ form : \beta \end{array}$$

The old rules are retained and we add new ones as indicated by the following examples.

- **Expressions:**

1. $\dfrac{\alpha \vdash_I me : \beta}{\alpha \vdash_I me.x : dt}$ (if $\beta(x) = dt$)
2. $\dfrac{\alpha \vdash_I fe : aet \to et, \alpha \vdash_I ae : aet}{\alpha \vdash_I fe(ae) : et}$

- **Function Expressions:**

  1. $\alpha \vdash_I f : ft \qquad (\text{if } \alpha(f) = ft \in \text{FTypes})$
  2. $\dfrac{\alpha \vdash_I me : \beta}{\alpha \vdash_I me.f : ft} \qquad (\text{if } \beta(f) = ft \in \text{FTypes})$

- **Class Expressions:**

  1. $\alpha \vdash_I cle : clet \qquad (\text{if } \alpha(cl) = clet \in \text{CIETypes})$
  2. $\dfrac{\alpha \vdash_I me : \beta}{\alpha \vdash_I me.cl : clet} \qquad (\text{if } \beta(cl) = clet \in \text{CIETypes})$

- **Commands:**

  1. $\dfrac{\alpha \vdash_I pe : aet\ \mathbf{proc} \qquad \alpha \vdash_I ae : clet}{\alpha \vdash_I pe(ae)}$
  2. $\dfrac{\alpha \vdash_I vet : \tau\ \mathbf{loc} \qquad \alpha \vdash_I e : \tau}{\alpha \vdash_I (vet := e)}$

- **Declarations:**
  **Modules**

  1. $\dfrac{dects : \beta}{(\mathbf{module}\ m : dects = d) : \{m = \beta\}}$
  2. $\dfrac{dects : \beta \qquad \alpha \vdash_I d : \beta}{\alpha \vdash_I \mathbf{module}\ m : dects = d}$

- **Classes:**

  1. $\dfrac{dects : \beta}{(\mathbf{class}\ cl(form) : dects; d) : \{cl = T(form) \longrightarrow \beta\})}$
  2. $\dfrac{dects : \beta\ form : \alpha_0 \qquad \alpha[\alpha_0] \vdash_{I \cup I_0} d : \beta}{\alpha \vdash_I \mathbf{class}\ cl(form) : dects : d} \qquad (\text{where } \alpha_0 : I_0)$

## Dynamic Semantics

First we define the sets FECon, ..., CIECon of function expression constants, ..., class expression constants by

$$
\begin{aligned}
&fecon ::= \lambda form.\ e : et\\
&pecon ::= \lambda form.\ c\\
&vecon ::= I\\
&mecon ::= \rho\\
&clecon ::= \lambda form.\ d : \beta
\end{aligned}
$$

and also add the productions

$$fe ::= fecon, \ldots, cle ::= clean \ \mid\ d$$

and define the sets DVal and Env of denotable values and environments by

$dval ::= con \mid vecon \mid fefon \mid pecon \mid clecon \mid mecon$
$\mathrm{Env} = \mathrm{Id} \longrightarrow_{\mathrm{fin}} \mathrm{DVal}$

and extend the definition of declarations by the production

$d ::= \rho$

These are mutually recursive definitions of a harmless kind. The extensions to the defintion of $\mathrm{FI}(fe), \ldots, \mathrm{FI}(de), \mathrm{FI}(d), \mathrm{DI}(d)$ are evident; for example $\mathrm{FI}(\lambda form.\ d : \beta = \mathrm{FI}(d) \backslash \mathrm{DI}(from)$.

We must also extend the definitions of $\alpha \vdash_I fe : fet, \ldots, \alpha \vdash_I cle : clet$ and $\vdash_I d : \beta$ and $\alpha \vdash_I d$ (the latter tow in the case $d = \rho$). The former extensions are obvious; for example

- **Class Abstracst:**

$$\frac{form : \alpha_0 \quad \alpha[\alpha_0] \vdash_{I \cup I_0} d : \beta}{\alpha \vdash_I (\lambda form.\ d : \beta)} \qquad \text{(where } \alpha_0 : I_0\text{)}$$

For the latter we have to define $\vdash_I decon : dt$ and this also presents little difficulty; for example

- **Class Abstracts:**

$$\vdash_I (\lambda form.\ d : \beta) : \mathrm{T}(form) \to \beta$$

Then we have the two rules

1. $$\frac{\forall x \in I_0 \quad \vdash_I \rho(x) : \beta(x)}{\vdash_I \rho : \beta} \qquad \text{(where } \rho : I_0\text{)}$$

2. $$\frac{\forall x \in I_0 \quad \alpha \vdash_I \rho(x) : \beta(x)}{\alpha \vdash_I \rho}$$

**Transition Relations**: The set Stores, is as before.

The configurations, final configurations and the transition relations for expressions, actual expressions and declarations are as before; for formals we have the same predicate as before. Now $fix\ \alpha : I$ and $\rho : \alpha|_J$ (for some $J \subseteq I$).

- **Function Expressions:** We take $\Gamma_\alpha = \{\langle fe, \sigma \rangle\ mid\ \exists fet.\ \alpha \vdash_I fe : fet\}$, $\mathrm{T}_\alpha = \{\langle fecon, \sigma \rangle \mid \exists fet.\ \alpha \vdash_I fecon : fet\}$ and the transition relation has the form $\rho \vdash_I \gamma \longrightarrow \gamma'$

The definitions for PExp, ..., CExp are the analogues of that for function expressions

**Rules** : • **Class Expressions:**

1. $\rho \vdash_\alpha \langle cl, \sigma \rangle \longrightarrow \langle clecon, \sigma \rangle \qquad (\text{if } \rho(cl) = clecon)$

2. $\dfrac{\rho \vdash_\alpha \langle me, \sigma\rangle \longrightarrow \langle me', \sigma'\rangle}{\rho \vdash_\alpha \langle me.cl, \sigma\rangle \longrightarrow \langle me'.cl, \sigma\rangle}$
3. $\rho \vdash_\alpha \langle \rho_0.cl, \sigma\rangle \longrightarrow \langle clecon, \sigma\rangle \qquad (\text{if } \rho_0(cl) = clecon)$

The rules for FExp, ..., MExp are similar except that in the last case we need also

1. $\dfrac{\rho \vdash_\alpha \langle cle, \sigma\rangle \longrightarrow \langle cle', \sigma'\rangle}{\rho \vdash_\alpha \langle cle(ae), \sigma\rangle \longrightarrow \langle cle'(ae), \sigma'\rangle}$
2. $\rho \vdash_\alpha \langle (\lambda from.\ d : \beta)(ae), \sigma\rangle \longrightarrow \langle \mathbf{private}\ from = ae\ \mathbf{within}\ d, \sigma\rangle$
3. $\dfrac{\rho \vdash_\alpha \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle}{\rho \vdash_\alpha \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle}$
   (where in the top line we mean a transition of decl.)

The new rules for expressions and commands should be clear; for example

- **Assignment:**
  1. $\dfrac{\rho \vdash_\sigma \langle ve, \sigma\rangle \longrightarrow \langle ve', sigma'\rangle}{\rho \vdash_\alpha \langle ve := e, \sigma\rangle \longrightarrow \langle ve' := e, \sigma'\rangle}$
  2. $\dfrac{\rho \vdash_\alpha \langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}{\rho \vdash_\alpha \langle I := e, \sigma\rangle \longrightarrow \langle I := e', \sigma'\rangle}$
  3. $\rho \vdash_\alpha \langle I := con, \sigma\rangle \longrightarrow \sigma[I = con]$

For declarations the new rules are

- **Modules:**
  1. $\dfrac{\rho \vdash_\alpha \langle d, \sigma\rangle \longrightarrow \langle d', \sigma'\rangle}{\rho \vdash_\alpha \langle \mathbf{module}\ m : dects = d, \sigma\rangle \longrightarrow \langle \mathbf{module}\ m : dects = d', \sigma'\rangle}$
  2. $\rho \vdash_\alpha \langle \mathbf{module}\ m : dects = \rho_0, \sigma\rangle \longrightarrow \langle \{m = \rho_0\}, \sigma\rangle$

- **Classes:**

  $displaystyle\rho \vdash_\alpha \ \mathbf{class}\ cl(form) : dects;\ d \longrightarrow \{cl = \lambda form.\ (\rho\backslash I)\ \mathbf{in}\ d\}$
  (where $I = \mathrm{DI}(form)$)

## 4.6 Exercises

**1.** Consider dynamic binding in the context of a simple applicative language so that, for example,

$\mathbf{let}\ x = 1;\ f(y) = x + y$
$\mathbf{in\ let}\ x = 2\ \mathbf{in}\ f(3)$

has value 5. What issues arise with type checking? Can you program iterations (e.g. factorial) without using recursive function definitions?

**2.** In a maximalist solution to the problem (in the applicative language) of neatly specifying functions of several arguments one could define the class of formal parameters by

$$form ::= \cdot \mid x : \tau \mid form, form$$

and merge expressions and actual expressions, putting

$$e ::= \cdot \mid e, e \mid f(e)$$

and amending the definition of definitions

$$d ::= form = e \mid f(form) : \tau = e$$

a) Do this, but effectively restrict the extension to the minimalist case by suitable choice of static semantics.

b) Allow the full extension.

c) Go further and extend the types available in the language by putting

$$\tau ::= \text{ int } \mid \text{ bool } \mid \tau, \tau \mid \cdot$$

thus allowing tuples to be denotable.

**3.** Consider the maximalist position in a simple imperative programming language.

**4.** Consider in a simple imperative language how to allow expressions on the left-hand of assignments:

$$e_0 := e_1$$

and even the boolean expression $e_0 \equiv e_1$ which is true precisely when $e_0$ and $e_1$ evaluate to the same reference. As well as discussing typechecking issues, try the two following approaches to expression evaluation:

a) Expressions are evaluated to their natural values which will be either locations or basic values.

b) Modes of evaluation are introduced, as in the text.

Extend the work to the maximalist position where actual expressions and expressions are merged, thus allowing simultaneous assignments.

**5.** Just as expressions are evaluated, and so on, formals are matched (to given actual values) to produce environments (= matchings). The semantics given above can be criticized as not being dynamic enough as the matching *process* is not displayed. Provide an answer to this; you may find configurations of the form

$$\langle form, con, \rho \rangle$$

useful where form is the formal begin matched, *con* is the actual value and $\rho$ is the matching produced so far. A typical rule could be

$$\langle x : \tau, con\rho\rangle \longrightarrow \rho \cup \{x = con\}$$

This is all for the applicative case; what about the imperative one? Investigate dynamic errors, allowing constants and repeated variables in the formals (dynamic error = matching failure).

**6.** In the phrase **rec** $d$ all identifiers in $R = \mathrm{FV}(d) \cap \mathrm{DV}(d)$ are taken to be recursively defined. Investigate the alternative **rec** $x_1, \ldots, x_n \cdot d$ where $\{x_1, \ldots x_n \subseteq R\}$.

**7.** In some treatments of recursion to evaluate an expression of the form

$$\begin{array}{l}\textbf{let rec } (f(x) = \ldots f \ldots g \ldots \text{ and } g(x) = \text{—}\ f\ \text{—}\ g\ \text{—})\\ \textbf{in } f(5)\end{array}$$

one evaluates $f(5)$ in the environment

$$\rho = \{f(x) = \ldots f \ldots g \ldots, g(x) = \text{—}\ f\ \text{—}\ g\ \text{—}\}$$

(ignoring free variables) and uses the simple transition:

$$\rho \vdash f(5) \longrightarrow \textbf{let } x = 5 \textbf{ in } \ldots f \ldots g \ldots$$

I could not see how to make this simple and nice idea (leave the recursively defined variables free) work in the present setting where one has nested definitions and binary operations on declarations. Can you make it work?

**8.** Try some examples of the form

$$\textbf{let rec } (f(x) = \ldots f \ldots g \ldots \& g(x) = fg) \textbf{in } e$$

where & is any of ;, **and** or **in**

**9.** Consider the following recursive definitions of constants:

a) **rec** $x : \mathrm{int} = 1$
b) **rec** $(y : \mathrm{int} = 1\textbf{and } x : \mathrm{int} = y)$
c) **rec** $(x : \mathrm{int} = y\textbf{and } y : \mathrm{int} = 1)$
d) **rec** $(x : \mathrm{int} = x)$
e) **rec** $(x : \mathrm{int} = y\textbf{and } y : \mathrm{int} = x)$

How are these treated using the above static and dynamic semantics? What do you think should happen? Specify suitable static and dynamic semantics with any needed error rules. Justify your decisions, considering how your ideas will extend to imperative languages with side-effects (which might result in non-determinism).

**10** Find definitions $d_0$ and $d_1$ to make different as many as possible of the following definitions:

a) (**rec** $d_0$; $d_1$))
b) (**rec**) (**rec** $d_0$; $d_1$)
c) (**rec**) ($d_0$ **rec** $d_1$)
d) (**rec**) (**rec** $d_0$ **rec** $d_1$)

where (**rec**)$d$ indicates the two possibilities with and without **rec**.

**11.** Check that the first alternative for type-checking recursive definitions would work in the sense that

$\alpha \vdash_V d : \beta \equiv \vdash_v d : \beta$ and $\alpha \vdash_V d$

**12.** Programming languages like PASCAL often adopt the following idea for function definition:

**function** $f(form) : \tau$
**begin**
    $c$
**end**

where within $c$ the identifier $f$ as well as possibly denoting a function also denotes a location, created on function entry and destroyed on exit; the result of a function call is the final value of this location on exit. For example the following is an obscure definition of the identity function:

**rec function** $f(x : \text{int}) : \text{int}$
  **begin**
    $f := 1$;
    **if** $x = 0$ **then** $f := 0$
    **else** $f := f + f(x - 1)$
  **end**

Give this idea a semantics.

**13.** *Call-by-need.* In applicative languages this is a "delayed evaluation" version of call-by-name. As in call=-by-name the formal is bound to the unevaluated actual, with the local environment bound in. However, when it is necessary for the first time to evaluate the actual, the formal is then bound to the result of the evaluation. Give this idea a semantics. One possibility is to put (some of) th environment into the configurations, treating it like a store. Another is to bind the actual to a new location and make the actual the value of that location in a store. Prove call-by-need equivalent to call-by-name. Consider delayed evaluation variants of parameter mechanisms found in imperative languages.

**14.** *Call-by-name.* Consider (minimalist & maximalist) versions of call-by-name in imperative programming languages. Look out for the dangers inherent in

> **procedure** $f(x : \text{int}\mathbf{name}) =$
> **begin**
> $\quad\vdots$
> $\quad\vdots$
> $\quad x :=..$
> $\quad\vdots$
> $\quad\vdots$
> **end**

**15.** Discover the official ALGOL 60 definition of call-by-name (it works via a substitution process); give a semantics following the idea and prove it equivalent to one following the idea in these notes (substitution = binding a closure).

**16.** *Call-by-text.* Give a semantics for call-by-text where the formal is bound to the actual (not binding in the current environment); when a value is desired the actual is evaluated in the then current environment. Consider also more "concrete" languages in which the abstract syntax (of the text) is available to the programmer, or even the concrete syntax: does the latter possibility lead to any alteration of the current framework?

**17.** *Call-by-reference.*Give a maximalist discussion of call-by-reference, still only allowing actual reference parameters to be variables. Extend this to allow a wider class of expressions which (must) evaluate to a reference. Extend that in turn to allow *any* expression as an actual; if ti does not evaluate to a reference the formal should be bound to a new reference and that should have the value of the actual.

**18.** *Call-by-result.*Discuss this mechanism where first the actual is evaluated to a reference, $I$; second the formal is bound to a new reference $I'$ (not initialized), third, *after* computation of the body of the abstract, the value of $I$ is set to the value of $I'$ in the then current store. Discuss too a variant where the actual is not evaluated at all until after the body for the abstract [**Hint**, user declaration finalization].

**19.** *Call-by-value-result.*Discuss this mechanism where first the actual is evaluated to a reference $I$; second the formal is bound to a new reference $I'$ which is initialized to the current value of $I$; third, *after* the computation of the abstract of the body, the value of $I$ is set to the value of $I'$ in the then current store.

**20.** Discuss **selectors** which are really just functions returning references. A suitable syntax might be

> **selector** $f(form) : \tau = e$

which means that $f$ returns a reference to a $\tau$ value. First consider the case where all lifetimes are semi-infinite (extending beyond block execution). Second consider the case where lifetimes do not persist beyond the block where they were created; in this case interesting questions arise in the static semantics.

**21.** Consider higher-order functions in programming languages which may return abstracts such as functions or procedures. Thus we add the syntax:

$$e ::= \lambda form.\ e \ \mid\ \lambda form.\ c$$

The issues that arise include those of lifetime addressed in exercise 20.

**22.** Here is a version of the typed $\lambda$-calculus

$$\tau ::= \mathrm{int}\ mid\ \mathrm{bool}\ mid\ \tau \longrightarrow \tau$$
$$e ::= m\ mid\ t\ mid\ x\ mid\ e\ bop\ \ mid\ \mathbf{if}\ e\mathbf{then}\ e\mathbf{else}\ e\ mid$$
$$\mathbf{let}\ x : \tau = e\mathbf{in}\ e\ mid\ (e)\ mid\ \lambda x : \tau.\ e$$

Give a static semantics and two dynamic semantics where the first one is a standard one using environments and where the second one is for closed expressions only and uses substitutions as discussed in the exercises of Chapter 3. Prove these equivalent. Add a recursion operator expression

$$e ::= Y$$

with the static semantics $\alpha \vdash_V Y : (\tau \longrightarrow \tau) \longrightarrow \tau(\tau \neq \mathrm{int}, \mathrm{bool})$ and a rule something like $\rho \vdash Ye_0 \longrightarrow e_0(Ye_0)$. What does this imply about formals which are of functional type and their evaluation, and why is that important?

# Appendix A

# A GUIDE TO THE NOTATION

| | | |
|---|---|---|
| **Syntactic Categories** | | |
| **Truthvalues** | $t \in \mathrm{T}$ | |
| **Numbers** | $m, n \in \mathrm{N}$ | |
| **Constants** | $con \in \mathrm{Con}$ | |
| **Actual Constants** | $acon \in \mathrm{ACon}$ | |
| **Unary Operations** | $upop \in \mathrm{Uop}$ | |
| **Binary Operations** | $bop \in \mathrm{Bop}$ | |
| | | |
| **Variables** | $v, f \in \mathrm{Var}$ | $V \subseteq_{\mathrm{fin}} \mathrm{Var}$ |
| **Identifiers** | $x, f, p, m, ci \in \mathrm{Id}$ | $I \subseteq_{\mathrm{fin}} \mathrm{Id}$ |
| | | |
| **Expressions** | $e \in \mathrm{Exp}$ | |
| **Boolean** | $b \in \mathrm{BExp}$ | |
| **Actual** | $ae \in \mathrm{AExp}$ | |
| **Variable** | $ve \in \mathrm{VExp}$ | |
| **Function** | $fe \in \mathrm{FExp}$ | |
| **Procedure** | $pe \in \mathrm{PExp}$ | |
| **Module** | $me \in \mathrm{MExp}$ | |
| **Class** | $cle \in \mathrm{CExp}$ | |
| | | |
| **Commands (=Statements)** | $c \in \mathrm{Com}$ | |
| | | |
| **Definitions/ Declarations** | $d \in \mathrm{Def/Dec}$ | |
| | | |
| **Formals** | $form \in \mathrm{Forms}$ | |
| **Types** | $\tau \in \mathrm{Types}$ | |
| **Expression** | $et \in \mathrm{ETypes}$ | |
| **Actual Expr.** | $aet \in \mathrm{AETyes}$ | |

| | |
|---|---|
| **Denotable Type Spec.** | $dts \in \text{DTSpecs}$ |
| **Declaration Type Spec.** | $dects \in \text{Dec TSpecs}$ |

**Static Semantics**

| | |
|---|---|
| **Free Variables/ Identifiers** | $\text{FV/I}(e), \text{FI}(c), \text{FV/I}(d)$ etc. |
| **Defining Variables/ Identifiers** | $\text{EV/I}(d)\ \ \text{DV/I}(form)$ |
| **Denotable Types** | $dt \in \text{DTypes}$ |
| **Type Environments** | $\alpha, \beta \in \text{TEnv}$ (e.g. $= Id \longrightarrow_{\text{fin}} \text{DTypes}$) |
| **Example Formulae** | $\alpha \vdash_V e : et\ \alpha \vdash_I c\ \alpha \vdash_I d : \beta$ <br> $form : \beta\ \ \text{T}(form) = aet$ |

**Dynamic Semantics**

| | |
|---|---|
| **Denotable Values** | $dval \in \text{DVal}$ |
| **Environments** | $\rho \in \text{Env}$ (e.g $= \text{Id} \longrightarrow_{\text{fin}} \text{DVal}$) |
| **Storeable Types** | $st \in \text{STypes}$ |
| **Locations** | $I \in \text{Loc} = \sum_{st} \text{Loc}_{st}\ \ L \subseteq_{\text{fin}} \text{Loc}$ |
| **Storeable Values** | $sval \in \text{SVal} = \sum_{st} \text{Val}_{st}$ |
| **Stores** | $\sigma \in Stores$ (e.g. $= \{\sigma \in \text{Loc} \longrightarrow_{fin} \text{SVal} \mid$ <br> $\forall st \in \text{STypes}.\ \sigma(\text{Loc}_{st}). \subseteq \text{SVal}_{st}\}$) |
| **Evaluation Modes** | $\mu \in \text{Modes}$ |
| **Transition Systems** | $\langle \Gamma, T, \rightarrow \rangle \qquad \gamma \in \Gamma$ <br> where $\Gamma$ is the set of configurations <br> $T \subseteq \Gamma$ is the set of *final* configurations <br> $\gamma \longrightarrow \gamma'$ is the *transition* relation |
| **Example Configuration** | $\langle e, \sigma \rangle;\ \langle c, \sigma \rangle, \sigma;\ \langle d, \sigma \rangle$ |
| **Example Final Configuration** | $\langle Con, \sigma \rangle; \sigma; \langle \rho, \sigma \rangle$ |
| **Example Transition Relations** | $\rho \vdash_{I,\mu} \langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ <br> $\rho \vdash_I \langle c, \sigma \rangle \longrightarrow \langle c', \sigma' \rangle / \sigma'$ <br> $\rho \vdash_I \langle d, \sigma \rangle \longrightarrow \langle d', \sigma' \rangle$ |

# Appendix B

# NOTES ON SETS

We use several relations over and operations on sets as well as the (very) standard ones. For example $X \subseteq_{\text{fin}} Y$ means $X$ is finite and a subset of $Y$.

Definition: Let $Op(X_1, \ldots, X_n)$ be an operation on sets. It is *monotonic* if whenever $X_1 \subseteq X_1', \ldots, X_n \subseteq X_n'$ we have $Op(X_1, \ldots, X_n \subseteq Op(X_1', \ldots, X_n')$. It is *continuous* if whenever $X_1^1 \subseteq X_1^2 \ldots \subseteq X_1^n \subseteq \ldots$ is an increasing sequence and $\ldots$ and $X_n^1 \subseteq X_n^2 \subseteq \ldots \subseteq X_n^m \subseteq \ldots$ is an increasing sequence then

$$(*)\quad Op(\bigcup_m X_1^m, \ldots, \bigcup_m X_n^m) = \bigcup_m Op(X_1^m, \ldots, X_n^m)$$

**Note C**: ontinuity implies monotonicity. Conversely to prove continuity, first prove monotonicity. This establishes the $'' \supseteq''$ half of $(*)$; then prove the "$\subseteq''$ half.

**Example**:

- **Cartesian Product:**

  $$X_1 \times \ldots \times X_n = \{\langle x_1, \ldots, x_n\rangle \mid x_1 \in X_1 \text{ and } \ldots \text{ and } x_n \in X_n\}$$

  is monotonic and continuous. Prove this yourself.

- **Disjoint Sum:**

  $$X_1 + \ldots + X_n \stackrel{def}{=} (\{1\} \times X_1) \cup \ldots \cup (\{n\} \times X_n)$$
  $$\textstyle\sum_{i \in A} X_i \stackrel{def}{=} \bigcup_{i \in A} \{i\} \times X_i$$

  Show that the finite sum operation is continuous. (Finite Sum is just union, but forced to be disjoint).

- **Finite Functions:** The Class of *finite functions* form $X$ to $Y$ is

  $$X \to_{fin} Y = \sum_{A \subseteq_{fin} X} A \to Y$$

  Note that the union is necessarily disjoint. Show that $\to_{fin}$ *iscontinuous.*

For $A \subseteq_{\text{fin}} X$ if $f \in A \to Y \subseteq X \to_{\text{fin}} Y$ (we identify $f$ with $\langle A, f\rangle$) we write $f : A$. This is used for environments (including type environments) and stores. There are two other useful operations.

For $f : A \;\; g : B$ in $X \to_{fin} Y$ we define $f[g] : A \cup B$ by

$$f[g](c) = \begin{cases} g(c) \; (c \in B) \\ f(c) \; (c \in A \backslash B) \end{cases}$$

and in case $A \cap B = \emptyset$ we define $f, g : A \cup B$ (also written $f \cup g$ by:

$$f, g(c) = \begin{cases} f(c) \; (c \in A) \\ g(c) \; (c \in B) \end{cases}$$

Note this is a special case of the first definition, but it is very useful and worth separate mention.

**The Importance of Continuity**

Suppose $Op(X)$ is continuous and we want to find an $X$ solving the equation

$$X = Op(X)$$

Put $X^O = \emptyset$ and $X^{m+1} = Op(X^m)$. Then (by induction on $m$) we have for all $m$ $X^m \subseteq X^{m+1}$ and putting $X = \overset{U}{m}\, X^m$

$$\begin{aligned} Op(X) &= Op(\bigcup X^m) \\ &= \bigcup Op(X^m) \text{ (by continuity)} \\ &= \bigcup^m X^{m+1} \\ &= X^m \end{aligned}$$

And one can show (do so!) that $X$ is the least solution – that is if $Y$ is any other than $X \subseteq Y$. Indeed $X$ is even the least set such that $Op(X) \subseteq X$.

This can be generalized, suppose $Op_1(X_1, \ldots, X_n), \ldots, Op_n(X_1, \ldots, X_n)$ are all continuous and we want to solve the $n$ equations

$$\begin{aligned} X_1 &= Op_1(X_1, \ldots, X_n) \\ &\vdots \\ X_n &= Op_n(X_1, \ldots, X_n) \end{aligned}$$

Put $X_1^O = \emptyset$ for $i = 1, n$ and define

$$X_i^{m+1} = Op_i(X_1^m, \ldots, X_n^m)$$

Then for all $m$ and $i$ $X_i^m \subseteq X_i^{m+1}$ (prove this ) and putting

$$X_i = \bigcup_m X_i^m$$

we obtain the least solutions to the equations – if $Y_i$ are also solutions then for all $i$, $X_i \subseteq Y_i$. Indeed the $X_i$ are even the least sets such that $Op_i(X_i, \ldots, X_n) \subseteq X_i (i = 1, n)$. This is used in the example below. Prove this.

**Example**: Suppose we are given sets Num, Id, Bop and wish to define sets Exp and Com by the abstract syntax

$$e ::= m \mid x \mid e_0 \text{ bop } e_1$$
$$e ::= x := e \mid c_0;\ c_1 \mid |; \textbf{if } e_0 = e_1 \textbf{then } c_0 \textbf{else } c_1 \mid \textbf{while } e_0 = e_1 \textbf{do } c$$

Then we regard this definition as giving us set equations

$$\text{Exp} = \text{Num} + \text{Id} + (\text{Exp} \times \text{BOp} \times \text{Exp})$$
$$Com = (Id \times Exp) + \text{Com} \times \text{Com} + (\text{Exp} \times \text{Exp} \times \text{Com} \times \text{Com}) + (\text{Exp} \times \text{Exp} \times \text{Com})$$

and also giving us a notation for working with the solution to the equations. First $m$ is identified with $\langle 1, m\rangle \in$ Exp and $x$ is identified with $\langle 2, x\rangle$ in Exp Next

$$\begin{aligned}
e_0 \text{ bop } e_1 &= \langle 3, \langle e_0,\ \text{bop } e_1\rangle\rangle \\
x := e &= \langle 1, \langle x, e\rangle\rangle \\
c_0;\ c_1 &= \langle 2, \langle c_0, c_1\rangle\rangle \\
\textbf{if } e_0 = e_1 \textbf{ then } c_0 \textbf{ else } c_1 &= \langle 3, \langle e_0, e_1, c_0, c1\rangle\rangle \\
\textbf{while } e_0 = e_1 \textbf{ do } c_0 &= \langle 4, \langle\rangle e_0, e_1, c_0\rangle
\end{aligned}$$

Now the set equations are easily solved using the above techniques as they are in the form

$$Exp = Op_1(\text{Exp}, \text{Com})$$
$$Com = Op_2(\text{Exp}, \text{Com})$$

where $Op_1(\text{Exp}, \text{Com}) = \text{Num} + \text{Id} + (\text{Exp} \times \text{BOp} \times \text{Exp})$ and $Op_2$ is defined similarly. Clearly $Op_1$ and $Op_2$ are continuous as they are built up out of (conmposed from) the continuous disjoint sum and product operations (prove they are continuous). Therefore we can apply the above techiniques to find a least solution Exp, Com. Note that Exp and Com are therefore the least sets such that

1. Num $\subseteq$ Exp and Id $\subseteq$ Exp (using the above identifications).
2. if $e_0, e_1$ are in Exp and *bop* is in Bop then $e_0$ bop $e_1$ is in Exp.
3. if $x$ is in $Id$ and $e$ is in Exp then $x := e$ is in Com.
4. . . . $\vdots$
5. . . . $\vdots$
6. if $e_0, e_1$ are in Exp and $c$ is in Com then **while** $e_0 = e_1$ **do** $c$ is in Com

At some points in the text environments (and similar things) were mutually recursively defined with commands and so on. This is justified using our apparatus of continuous set operators employing, in particular, the finite function operator.